Colored-Object Programming : Color Graphs, a Visual Formalism for Synthesizing the Behaviour of Objects

Henry J. Borron

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Colored-Object Programming : Color Graphs, a Visual Formalism for Synthesizing the Behaviour of Objects

Henry J. Borron

N° 2876
Avril 1996

THÈME 2

Colored-Object Programming : color graphs, a visual formalism for synthesizing the behaviour of objects.

Henry J. Borron *

Programme 2 — Génie Logiciel et Calcul Symbolique
Action LeTool

Rapport de Recherche N° 2876 — Avril 1996 — 18 pages

Abstract. This paper is about colored object programming, a refinement of object oriented programming. In this approach, the answer of an object to an external event (message or generic function call) depends not only on its class, but also on its current state.

The paper presents a visual formalism (a notation, not a programming environment) for describing the whole behaviour of objects of a same class, i.e. the behaviour resulting from each behaviour increment defined in this class and in each of its superclasses.

The formalism describes states and transitions between them (external events) in a N-dimension space. It is grounded on connectedness whereas related visual formalisms are grounded on insideness (these derive from statecharts).

Keywords. Language design, visual formalism, object oriented programming, state, transition, colored objects, abstraction, modularity, cleanness.

(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 : les graphes de couleurs, un formalisme visuel pour synthétiser le comportement des objets.

Résumé. Ce papier a trait à la programmation par objets colorés, un raffinement de la programmation par objets. Dans cette approche, la réponse d'un objet à un événement externe (message ou appel de fonction générique) dépend non seulement de la classe de l'objet, mais aussi de l'état courant de celui-ci.

Le papier présente un formalisme visuel (une notation et non pas un environnement de programmation) pour décrire l'ensemble du comportement des objets d'une même classe, i.e. le comportement résultant de chaque incrément de comportement défini dans cette classe et dans chacune de ses superclasses.

Le formalisme décrit les états et les transitions entre ceux-ci (événements externes) dans un espace à N-dimension. Il est basé sur la connexion alors que les formalismes les plus proches sont basés sur l'inclusion.

Mots-clés. Conception de langage, formalisme visuel, programmation par objets, état, transition, objets colorés, abstraction, modularité, propreté.

Colored-Object Programming: color graphs, a visual formalism for synthesizing the behaviour of objects.

"(...) diviser chacune des difficultés (...) en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre" René Descartes (Discours de la Méthode, 1637)

1. INTRODUCTION

This paper is the first one of a sery of three companion papers about a refinement of Object Oriented Programming (OOP)[1].

This first paper is devoted to the external behaviour of class instances using a new formalism ; the second one [Borron, 1996d] elaborates the handling of independent supplementary behaviours (mixins) ; the third one [Borron, 1995c] is about class inheritance.

This sery specifies our proposal (results). Other articles will describe the rationale behind such results, either from a computer science point of view or from a cognitive/ergonomical point of view.

1.1 Goal of the paper

This article presents a formalism for describing the WHOLE behaviour of objects of a same class, i.e. the behaviour resulting from each behaviour increment defined in this class and in each of its superclasses. In other words, the formalism is invariant for a given class of objects, the behaviour of these objects being implemented in a flat manner or using a complex hierarchy of superclasses : the same programming interface will be exhibited whatever the case. The proposed formalism is preferably used in its visual form.

We term **color graph** the interface of a class, i.e. the structure capturing the whole behaviour of its instances. (The term behaviour is used to reflect the abstraction level.) A color graph is made of nodes and transitions. A node depicts a possible instance state or a contribution to it ; a transition, the possibility for a message to occur in a certain state and its effect on the instance state. Methods and memory representations[2] are really second citizens in this formalism : they are defined only at the implementation level

1.2 Overall approach

In this subsection, we sketch the links of this paper with the other two of the sery. As a matter of fact, the way we introduce and (re)define (class) inheritance is a key point of our approach.

a) A form of inheritance is defined in one given color graph, i.e. considering the whole behaviour of a single class : termed "LOCAL INHERITANCE", this form is initially specified for transitions (specification level), then adapted to methods and memory representations (implementation level)..

b) Local inheritance is then generalized to CLASS INHERITANCE, i.e. considering a hierarchy of color graphs, each one being attached to one class : this results in a redefinition of class inheritance which, in our view, clarifies this concept and brings a number of practical advantages, for example a purely declarative combination mechanism. The generalization is first done for transitions (specification level) then for methods and memory representations (implementation level).

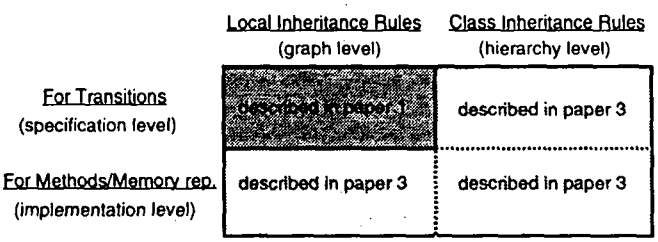


Figure 1.

To be more precise, this first paper is devoted to the external behaviour of class instances : it describes how color graphs are built, defining three essential constructs (selection, decomposition, conjunction). LOCAL inheritance for transitions is defined in this paper.

The third paper is devoted to CLASS inheritance from both an abstract point of view and an implementation one. First it shows that the decomposition and the derivation (a specialized form of decomposition construct devised for handling mixins) are central for abstractly composing a class out of superclasses. In this part, LOCAL inheritance rules for transitions are thus generalized to CLASS inheritance rules for transitions. Second, it describes how concrete implementations (methods and memory representations) can be attached to a color graph and can be inherited LOCALLY (i.e. inside one color graph) thanks to carefully chosen rules ; the paper then shows how these local inheritance rules for implementations can be generalized for CLASS inheritance (involving a hierarchy of color graphs).

While the third paper takes for granted the derivation and mixin constructs, the second paper focuses on their design, elaborating them on top of the essential constructs. The derivation, a powerful specialized decomposition, abstracts the assembly of a basic behaviour with a mixin one (specification level).

Although better read in a row, each paper is self contained. This supposes some repetitions : we tried to reduce them to a minimum.

Important : all these papers present **concepts** and NOT a programming environment. A forthcoming paper will show how a set of interactive tools may support the proposed notation in a quite friendly way.

---

[1] The original version of this paper was written in February 1996. Only surface modifications were made since then.

[2] instance variables [in Smalltalk] or slots [in CLOS]

1.3 Plan of the paper

This first paper is organized as follows : next section presents a formal definition of color graphs (**section 2**) ; **section 3** rapidly describes their visual aspect ; **section 4** illustrates all this using one example ; **section 5** presents an additional concept for refining a color graph specification ; **section 6** relates the color graph formalism to other works, notably predicates classes and higraphs ; a **conclusion** stresses the consequences of these relationships.

Depending on his/her personal tastes and habits, the interested reader may well prefer to start with the example part (the *Person* color graph in section 4) instead of the concepts one (section 2) : this example is commented in a way that progressively introduces the terms and concepts defined in the formal part ; no backward reference is made to ensure independence.

2. COLOR GRAPHS : SPECIFYING INSTANCE BEHAVIORS

In COP, the behaviour supported by a class (including its ancestors)is abstracted as a whole in the form of a color graph describing possible (reachable) instance states and transitions between them[3]. For the sake of precision, a vocabulary has been coined consistently using the theme of colors as a metaphor.

2.1 States

States are described according to one or several **dimensions**. Several dimensions are useful from a description point of view ; they also enable to fight against the combinatorial explosion effect encountered otherwise (this aspect is shown in length in [Borron, 1996b]).

C-graph

In a single dimension space, a reachable instance state is termed a **color**[4]. (The corresponding graph is termed a **c-graph**). A color may be further qualified as being either **basic** or **ephemere**. A basic color corresponds to one point on the space unique axis ; an ephemere color recursively corresponds to a cloud of basic colors.

P-graph

In a N dimensions space (N≥2), an elementary contribution to a reachable instance state is termed a **pigment**. (The corresponding graph is termed a **p-graph**). All the pigments defined along one same dimension form a **scale**. N scales of pigments, one per dimension, constitute a **palette**[5]. Several palettes may be used in a same p-graph, but only one is valid at a given time (during execution). In a given scale, a distinction is made between basic and ephemere pigments : a basic pigment corresponds to one possible value of a coordinate ; an ephemere pigment recursively corresponds to a cloud of basic pigments.

A p-graph may also feature basic and ephemere **blends**. Basic blends are recursively obtained by composing two or more pigments of different scales of a same palette. The **degree** of a basic blend is the number of these pigments ($2 \leq d \leq N$). In a N-dimension space, a basic blend of degree N corresponds to one point. Ephemere blends are less frequent : they recursively correspond to clouds of basic blends. All basic blends involved in a same ephemere blend have the same degree ; this degree is the degree of the ephemere blend.

The degree of a pigment is always one. The degree of a color graph is the number of scales in one palette (N). Pigments and blends are collectively called **p-chromas**. In a c-graph, a reachable instance state is thus represented by k p-chromas ($1 \leq k \leq N$), the degrees of which sum to N.

Chromas

Colors, pigments and blends are collectively called **chromas**. Blends of degree N may also be termed colors.

2.2 Regular transitions

Regular transitions model acceptable messages and their effect. They should not be confused with reflex transitions : these model automatic changes due to internal conditions. Regular transitions are described first, just below. The adjective "regular" may be omitted when the context is sufficiently clear.

State transitions

A regular **state transition** models the possibility for a message to be received in a certain state (the **source** state) and its effect in changing the **dictionary** of the messages accepted by the instance (this is modelled by the **destination** of the transition). A regular state transition has a name (or a **selector**, to use a Smalltalk word).

Graph transitions

The regular transitions that appear in c-graphs and p-graphs are termed regular **graph transitions**. They expressed regular state transitions. As such, they also have a name (selector). For a c-graph, the transposition is immediate : states are directly represented by colors ; and state transitions, by graph transitions.

In a p-graph, a state corresponds to N pigments. A state transition is thus basically represented by N **micro-transitions**, one per dimension *(decomposition step).* Each micro-transition flows from one pigment of the source state to one pigment of the destination state (the source and the destination pigments belong to the same scale). Each such micro-transition is **constrained** by an elementary **clause** mentioning the source pigments of all the other (N-1) micro-transitions. Usually, most of these micro-transitions, say k1, are side-effect free : side-effect free transitions are circular (destination = source) and, by convention, not shown (except one, when k1 equals N). Applying this convention yields two cases : when a single micro-transition subsists, it is termed a **mono-micro-transition** ; when the result consists in a group of several micro-transitions, it is termed a **multi-micro-transition**.

---

[3] The reader is invited to note that we haven't extended the color graph formalism to reactive systems (this is not due to an intrinsic limit of the formalism itself : see § 6.2.1.3.)

[4] Idioms often translate a state as a color. For example, someone may be green with envy, red with shame, etc.

[5] A palette is also termed a pigment system ; and a scale, a family : see [Borron, 1995a]. Other shifts in terminology are : reflex transition (instead of : void transition), g-circular (d-circular), i-circular (s-circular).

These mono- or multi-micro-transitions are constrained. In practice, factorization is used to simplify clauses *(unification step)* : the clauses of micro-transitions having a same source and a same destination are grouped and possibly eliminated (by convention, a clause is not shown when expresssing that its transition is valid for all pigments of all other scales in the same palette). After unification and a possible simplification, a mono-micro-transition is named a **simple transition** (it is possibly constrained) ; a multi-micro-transition, a **composite transition** (its components are all constrained).

Blends are useful for eliminating remaining clauses *(purification step)* : a constrained simple transition may theoretically be replaced by an unconstrained one (the new one flows from a blend composing the clause and the source pigment of the initial transition, to a blend composing the clause and the destination pigment of the initial transition). Similarly, a composite transition may theoretically be replaced by one unconstrained simple transition between two blends.

On top of that, we adopted a few principles for cognitive reasons. An important one is the "unique destination principle" : when similar graph transitions (same name) flow out of a same source chroma to different destination chromas, they get replaced by a unique graph transition flowing to the ephemere chroma representing all these destinations (the underlying system is taken advantage of for choosing between the destinations by automatically testing their conditions). A second important principle is the "non ubiquity" principle : according to it, a given instance state is to be represented only once in a palette. This requirement conflicts with the systematic elimination of clauses, i.e. the existence of simple unconstrained transitions only : simple constrained transitions and composite transitions may have to be reintroduced[6]. For more details about that and the principles underlying color graphs, refer to [Borron, 1996b].

In practice, a p-graph is usually built up from scratch. This is done in an intuitive way : because the p-graph dimensions are chosen to be independent or almost independent, a state transition is most often mapped to a simple graph transition (usually unconstrained) and, more rarely, to a composite transition (several interdependent constrained graph transitions). Any graph transition goes from a pigment to another pigment, or from a blend to another blend of same degree. An unconstrained simple graph transition of degree d is **valid** for any state made of the d pigments recursively composing the source chroma, the (N-d) other pigments being free ; if the transition is constrained by a clause of degree c, c other pigments are fixed[7]. Each effectively free pigment can be replaced by any basic pigment of its scale. Intuitively, the existence of free pigments means the transition can be "distributed" to all pigments equal to one possible value, or to blends mixing several of these pigments (mixed pigments should pertain to different scales). A composite transition is made of several constrained graph transitions all valid at the same time : the ANDing of the source and clause of each one systematically yields the same result (the origin of the composite transition). The destination of a composite transition is obtained by combining the destinations of its components.

Degree of a transition

In a c-graph, the **degree** of any graph transition is one. In a p-graph, the degree of an unconstrained graph transition between two p-chromas of degree d ($1 \leq d \leq N$) is d. It is $d+c$ when this transition is constrained by a clause of degree c. The degree of a simple transition is thus $d+c$. The degree of a composite transition is also $d+c$: it is the degree of any one of its component graph transitions (all have the same).

[Intuitively, the degree of a state transition is the number of dimensions that intervene in the transition (for consultation or modification). For example, the *draw* transition of a *Circle* has a degree 2. Considering p-graphs, the above definitions usually correspond to the intuition : a sufficient condition is that the uninitialized state is decomposed into pigments, a quite normal situation indeed. Considering c-graphs, the intuition is often unsatisfied : this is because instance states should better be decomposed according to several dimensions.]

Incoming, outcoming and circular transitions

With respect to a chroma C, an **incoming** graph transition is one the destination chroma of which is C ; an **outcoming** one is one the source of which is C. A **circular** graph transition is such that its source and its destination are the same chroma. A **pure** outcoming (resp. incoming) graph transition is a non circular outcoming (resp. incoming) graph transition.

Similar transitions

Given a selector S, a transition is **similar** to S if its own selector is identical to S. A transition is similar to a message if it is similar to the message selector. Two transitions are similar if their selectors are identical. Similarity is the property of being similar.

2.3 Current state

Token

In a c-graph, the current state of one instance is marked by a single **token** in one color. On each reception of a message (supposed to be not erroneous), this token migrates along the regular transition which is triggered (from the source color of this transition to its destination color) : depending on whether this graph transition is a circular or pure outcoming one, the token remains in its initial place or leaves it.

Mini-tokens

In a p-graph, the state of an instance is marked by N **mini-tokens**, one per dimension. A p-chroma of degree d groups d mini-tokens when it belongs to the instance state. On reception of a message, a simple or composite transition is normally triggered : each involved graph

---

[6] To represent a simple transition flowing to an ephemere blend without loosing in precision (i.e. without introducing destinations that were not part of this original specification), all elementary constrained transitions of a same composite transition specifying one original destination should be kept together : a qualification is thus required. A visual detail (color, number, keyword or something else) may illustrate that. (At the textual syntax level, a different **deftransition** will be used for each possible destination : this solves the problem.) The formalism is robust. In other words, less precision does not yield a wrong result : the correct destination will be found ; however, its computation will be less efficient.

[7] As a matter of fact, the existence of unreachable states may be taken advantage of for simplifying a clause (at unification time).

transition, constrained or not, moves a priori D mini-tokens if the degree of its source is D. Note the same scale(s) are involved at the source and at the destination.

Mark

The term **mark** is generic ; it refers to the possible contents of a chroma : a token in case of a color ; a mini-token in case of a pigment ; d mini-tokens in case of a blend ($2 \leq d \leq N$).

2.4 Conditions

Each chroma has an **id**, i.e. a natural integer (≥ 0) or a name (say a string made of letters and/or digits, and starting with a letter). Each one is also attached a boolean **condition** (evaluated by sending a side-effect free message to the instance : the associated **testing transition** is automatically derived by the system) and/or results from the instantiation of an AND-type or OR-type **construct**.

2.5 Constructs

Essential constructs

Three types of **essential constructs** exist : the **decomposition** that explodes a color into N pigments (one per scale in a given palette) ; the **conjunction** that groups two or more p-chromas into a blend ; the **selection** that materializes a choice between two or more chromas. The first two are encountered only in p-graphs ; the third one exists in c-graphs (selections on colors) and in p-graphs (most of the time, selections on pigments ; yet, selections on blends also exist). Let's briefly pass them in review.

The selection is a **diverging** construct : conceptually, it is a small tree, made of one source (an ephemere chroma) and several destinations (chromas, too). The condition attached to the source node is obtained by ORing the conditions attached to the destination chromas. These must form a partition : only one destination condition may be true at any given moment. [One destination may occasionally be specified as being an INIT one: when entering the selection from the outside, this node gets selected without testing.[8] The INIT condition may be precised if necessary. [9]]

The decomposition is also a diverging construct. The condition attached to its source (a color) is obtained by ANDing the conditions attached to its destinations (pigments).

The conjunction is a **converging** construct : conceptually, it is a small tree with one destination (a blend) and several sources (pigments and/or blends). The condition of its destination is obtained by ANDing the conditions of its sources.

Specialized constructs

The **derivation** construct is a special form of decomposition in two subgraphs ; the **mixin** construct, a special form of selection. Both work hand in hand for abstracting independent supplementary behaviours and their assembly with a basic behaviour. Their description is given in companion paper n° 2.

Reflex transitions

In any essential construct, a source and a destination are linked by a **reflex transition**. The semantics of all essential constructs is built on top of the semantics of a reflex transition. This one is simple : a reflex transition is **armed** when its source node has received its mark ; if armed, a reflex transition **fires** when the condition of its destination node gets true. In a selection, only one reflex transition may fire ; in a decomposition or conjunction, all reflex transitions fire at the same time.

This firing means the transfer of the mark(s) from the source chromas(s) to the destination chromas(s). More precisely, a decomposition exchanges the token of the source color with N mini-tokens in the destination pigments ; a conjunction transfers all the mini-tokens present in its source chromas into its destination chroma ; a selection transfers the mark from the ephemere source chroma to one destination chroma, the selected one (the testing is done automatically ; no external message can be sent to an instance while in an ephemere chroma)

From this, one can infer, for example, that the degree of the destination blend of a conjunction is the sum of the degree of all it sources ; or that the same scales are to be involved in each destination of a p-graph selection ; ...

2.6 Local Inheritance of transitions

Inheritance rules in Colored Object Programming (COP) include local and class inheritance rules on one hand, specifications (transitions) and concrete implementations (methods and memory representations) inheritance rules on the other hand. This subsection focuses on one part of that, namely on the LOCAL inheritance rules for transitions, i.e. only on those rules that enable the sharing of TRANSITIONS inside one SAME color graph.[10] Figure 1 illustrates that. Let's now introduce these rules in a progressive manner.

Factorization facilities

Factorization facilities proceed from the idea of sharing in one place similar regular transitions which are either circular or flowing to the same node. They apply only when transitions to be factorized all outcome from the destinations of a selection (from all of them) : the goal is not to create a new diverging topology of nodes[11], but to reuse an existing one. Because an instance temporarily in an ephemere chroma automatically executes a testing and cannot accept any external message, we can use this ephemere chroma for factorizing other transitions : this is not ambiguous.

---

[8] More generally, several INIT nodes may be specified when the destination is not unique and depends on which regular transition was actually activated among all those that flow to the root selection : these INIT nodes are parameterized by these transitions.

[9] Ex . : (n=1) when the node attached condition is (n>0).

[10] Other rules are part of the companion paper n° 3.

[11] We examined this question (factorization constructs) in a preceding paper [Borron, 1995a]. A brand new node (termed virtual) was necessary : by construction, no mark ever traverses it. (On the opposite, chromas are visited.) Abandoning this idea makes chroma and node two equivalent terms.

Three cases are distinguished :

(1) all transitions to be factorized are circular : they are all replaced by one single circular regular transition attached to the ephemere chroma. This **factorizing transition** is termed **i-circular** (i for individual) ;

(2) all transitions to be factorized have one same destination D which is different from the ephemere chroma : they are all replaced by one single regular transition originating from the ephemere chroma and flowing to the destination D. No special name is given to the factorizing transition ;

(3) all transitions to be factorized flow to the ephemere chroma : the factorizing transition thus originates from this ephemere chroma (factorization of all possible sources) and flows to it (common destination) : being circular, it is termed **g-circular** (g for group) : when applied, the destination is one of the selection destinations, any one in this group and not necessarily the source node of the effectively triggered transition (as in the i-circular case).

Circular transitions factorized in an ephemere chroma represent a relatively frequent situation. Note that an i-circular transition replaces d destination-to-destination transitions (where d is the number of destinations of the considered selection) whereas a g-circular transition replaces in fact d^2 destination-to-destination transitions. Note also that the testing transitions associated with conditions at the selection destinations may also benefit from an i-circular type of factorization : corresponding factorizing transitions unify in this case with the actual testing transitions also existing at the selection source (in constrast, other factorizing transitions are virtual).

Local inheritance rule

In practise, it is temptating to invent rules for avoiding to repeat transitions having a same name in a color graph. Factorization facilities proceed from this idea. One can observe that a decomposition into pigments also yields nice factorizations when dimensions are naturally orthogonal. (As an example, compare the c-graph and p-graph of *Circle* instances : in the c-graph, the *surface* transition -for instance- is attached to two colors ; in the p-graph, it is attached to one pigment.)

A common base sustains all this : regular transitions are inherited along reflex transitions (except when they originate from the source node of a decomposition). This is a natural property of a conjunction (because the destination condition ANDs all the source conditions) ; for the selection, the property results from a choice : the factorization facilities (applied to testing or non-testing transitions). This property is called the **local inheritance rule** for transitions. In a given chroma, transitions that apply are thus either **local** (outcoming from that chroma) or **inherited**.

Ancestor tree

Given a chroma in a color graph, we term its **ancestor tree** (or, loosely, its **ancestor graph**) the structure made by all the reflex transitions recursively converging to that chroma (except when they conceptually originate from the source node of a decomposition[12]) : this ancestor tree includes the source nodes of all these reflex transitions as well as the considered chroma (called the **root**).

[Ancestor trees are not graphs : no cycle is possible and DAGs cannot exist since (1) partitioning is required in selections ; (3) the source and destination nodes of a selection pertain to the same family /ies ; (3) blends mix only (yet recursively) pigments of different families.]

In a c-graph, the ancestor tree is also called the **c-ancestor-tree** (or c-ancestor-graph).

In a p-graph, the ancestor tree may be **simple** (if the root is a pigment) or **multiple** (if the root is a blend of degree d) : we term **p-ancestor-tree** (or p-ancestor-graph) the restriction of the ancestor tree to the nodes participating to one given scale (i.e. which may hold the mini-token dedicated to that scale).

2.7 Valid regular graph transitions and their effect

Computing the dictionary of accepted messages

In a given class, a **dictionary** of accepted messages (DAM, for short) is associated to each possible state[13]. (Note we are not considering here implementations but specifications.) Given a state S, the DAM is obtained by collecting all valid graph transitions that outcome from the considered state (**punctually defined** transitions) plus all those that are **inherited** : the DAM organizes them according to their similarity (one entry per different selector). In a c-graph, since each possible state is directly represented by a color, the transitions to be considered are all those that outcome from the colors belonging to the ancestor tree rooted at the color representing the state S. In a p-graph, transitions to be considered are all valid transitions outcoming from the p-chromas belonging to the p-ancestor-trees rooted at the k p-chromas representing the state S ($1 \leq k \leq N$).

Computing the next state

When a message occurs, it triggers all valid graph transitions that are similar to it (these transitions are stored in the DAM associated to the current instance state at the entry corresponding to the message selector). If no transition is valid (no DAM entry for the message selector), the message is invalid : an error is detected and the user is warned ("message not understood : ...").

The external effect of a valid message is to possibly change the instance DAM ; in other words, the instance state. This depends on the destination(s) of the triggered transition(s). Basically (i.e. not considering factorization facilities), a circular graph transition has no effect on the instance state ; on the opposite, a pure outcoming transition may induce a new state.

To take care of a factorization facility, we consider the **elected** graph transition among all those the facility factorizes, i.e. the one that would effectively be triggered if the facility was not used. Two cases are to be considered depending on the actual form of the factorization facility. If i-circular, the elected transition

---

12 In the visual representation of color graphs (outlined in section 3), these reflex transitions are not shown (a bar is drawn instead). The ancestor tree is thus easy to see : it is composed by all the visible reflex transitions recursively converging to the considered chroma.

13 Being only interested in external messages, we do not consider here ephemere chromas.

is circular : hence, no effect on the instance state (destination chroma = source chroma). Otherwise (i.e. g-circular or pure outcoming), the elected transition is a pure outcoming one : its effect is materialized by its destination which is identical to the (g-circular or pure outcoming) transition factorizing it.

The new state is thus computed as follows (taking care of selections, conjunctions and factorization facilities) :

- in a c-graph, a single graph transition may fire. Two steps are distinguished. **Move step** : the token stays in its position if the transition is i-circular (final position) ; otherwise, the token is moved to the destination of the transition. **Propagation step** : reflex transitions are activated : a number of them (selections) may then fire, hence the new instance state ;

- in a p-graph, several valid graph transitions may be triggered by a message ; each one may concern one or several mini-tokens. Three steps are distinguished. **Move step** : for each non i-circular triggered transition, the involved mini-tokens are moved to the transition destination. **Backtracking step** : moves may have depleted a number of blends ; the mini-tokens that remain in them are moved back to their original pigments. **Propagation step** : reflex transitions are activated : a number of them (selections and conjunctions) may fire, hence the new instance state.

This algorithm presupposes the color graph in question satisfies the "unique destination" principle. If this principle is disobeyed in a c-graph, the token is duplicated in each source of the transitions in question. When tests are done, a unique destination is selected. A single token subsists. (In a p-graph, this algorithm is generalized to the affected mini-tokens.)

2.8 System-defined transitions

Testing transitions

A recurrent situation concerns the **testing transitions** of a selection : they a priori exist at the ephemere source chroma (where the testing is made) as well as at each destination chroma (where they deliver a constant answer). Due to the local inheritance rule, the transitions at the destinations are inherited from the ones at the ephemere source chroma. The underlying system automatically generates these ones from the conditions attached to the destinations.

More generally, when a chroma is attached a condition, the corresponding transition exists at this chroma (where it delivers a constant true answer) as well as at its **siblings** (where it delivers a constant false answer), the siblings being defined as chromas that may get the same mark and are equally initialized : in a p-graph, the testing transition is attached at the source of each ephemere pigment in each p-ancestor-tree of the considered chroma[14] : in a c-graph, the testing transition is attached at the source of each ephemere color of the c-ancestor-tree of the considered color. (When several ephemere chromas have been found in a same p-ancestor-tree or in the c-ancestor-tree, a comparison is made between the destinations of the different selections so as to capture the considered chroma and all its siblings using the least possible number of these selections.)

Creation transitions

A special node, termed **pseudo-color**, represents the case where an instance is still not created (or has been destroyed). One or several **creation transitions** may outcome of it and flow to one or several different initial colors[15]. A single default creation transition is provided (termed *make-instance* for CLOS compatibility) ; a default initial color too (color 1). Other creation transitions are explicit (must be named). Any explicit creation transition cancels the default one.

Objects may exist that cannot be created nor destroyed by the user. These objects are termed **immutable**. Integers constitute one such example. In these cases, a **base color** is defined (the pseudo-color does not exist).

Transformation transitions

If several palettes are used, **transformation transitions** may be defined by the user to explicitely pass from one palette to another one (corresponding changes of representation are automatically implemented by the underlying system).

2.9 Multi-transitions

In this paper, we almost systematically use the term "message" (cf. : Smalltalk) because it appears more intuitive than "generic function" ; however, the proposed formalism is valid also for expressing the behaviour of objects acted on by generic function calls (cf. CLOS). These generalize message sendings in considering several specializers (i.e. instances of known classes) instead of a single receiver.

COP naturally supports such a view : several color graphs may be involved instead of a single one. A transition that is attached to the chromas of several specializers is termed a **multi-transition**. Multi-transitions may be composite with respect to an involved color graph. (Multi-transitions are not examplified in this paper.)

2.10 Other features

Given its subject, this paper does not describes all details about color graphs. Two other interesting features appear in companion paper n° 2 : constrained subtrees ; and unwilling transitions.

3. COLOR GRAPHS : THEIR VISUAL ASPECT

An attractive aspect of COP is its visual interface, the basis of a future visual programming environment. Let's give some indications about it.

A node is pictured as a small bubble : inside the bubble, a name or an integer identifies the node : close to the bubble, the condition attached to this node is named (inferred conditions are usually not displayed).

---

14 When the considered chroma is a blend, the testing transition takes a priori the form of a composite transition for symmetry purposes, yet it could be simplified as a constrained simple transition.

15 Initial chromas may be defined at other levels. This feature is much less used except in mixins (see companion paper 2)

Regular graph transitions are represented with named solid thick arrows (the name, i.e. selector, is underlined for g-circular transitions). **Reflex transitions** are represented by unnamed thin arrows : using a broken line for **conjunctions**, a dotted line for **selections**. A **decomposition** is not represented using reflex transition arrows, but by a bar stuck between the leaf nodes and the root node (shown by its name only, most usually).

These distinctions may be intensified when the medium enables colored drawings : it appears interesting to attribute a same coloration to pigments of a same scale and to reflex transitions outcoming of them ; and to draw all their contours the same way.

Circular transitions may be shown without arrows. They may be further qualified (and shown) as modifying or consulting (default) for coherency checks. **Testing transitions** (i.e. regular transitions associated to side-effect free messages used for evaluating conditions), being automatically derived by the system, are usually not shown. An **INIT destination** chroma of a selection is shown with a double contour.

The **pseudo-color** is represented by a small segment with the class name above it[16]. Usually, a single unnamed arrow ondulates from it : this one corresponds to the default **creation transition** (*make-instance*). In case several creation transitions exist, the pseudo-color may be duplicated. For immutable objects like integers, the **base color** is shown as a square.

4. COLOR GRAPHS : AN EXAMPLE

This example illustrates the decomposition, selection and conjunction constructs. It is described in quite a detailed way to let the interested reader starts his/her reading using it ; it also shows how much information can be crystallised in a single color graph (often implicitely).

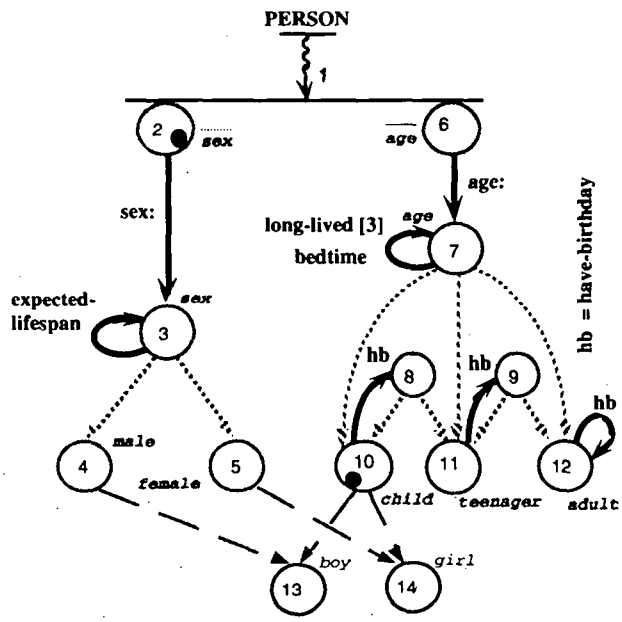


Figure 2.

16 It is noted _ (underscore) in textual syntax.

Figure 2 is derived from the source code of an example given in section 4.2 of reference [Chambers, 1993][17]. *Person* instances are differentiated into *male* and *female* categories according to the *sex* ; into *child*, *teenager* and *adult* categories, according to the *age*. In addition, both criteria are used to distinguish *boy* and *girl* categories.

4.1 Static description

4.1.1 Color graph skeleton

a) Dimensions ; decomposition ; pigments

In terms of COP, two **dimensions** are used : *sex* and *age*. (The **degree** of the color graph, symbolized by N in this paper, is two.) First dimension corresponds to nodes numbered 2 to 5 ; second one, to nodes numbered 6 to 12. Such nodes specify the state of a subpart of a *Person* instance : they are termed **pigments**. The whole set of pigments on a same dimension form a **scale**. A pigment of each scale is required to describe an instance state.

For example, when just created, a *Person* instance is uninitialized : it is represented by the pigment list (2, 6). Pigment 2 (resp. 6) means *sex* (resp. *age*) has not received a value yet. (This is the meaning of the **condition** attached to these pigments, respectively *not sex* and *not age*). In this case, pigments 2 and 6 result from a **decomposition** of **color** 1, the initial state. (Color 1 is not represented by a bubble, but uniquely by its id above the decomposition bar.)

b) Selection ; ephemere pigments

Pigments 2, 4 and 5 (resp. 6, 10, 11, 12) are termed **basic** pigments in the first (resp. second) scale ; pigments 3, 7, 8, 9 are termed **ephemere** pigments. Each ephemere pigment is the source of a **selection** on (basic or ephemere) pigments of a same scale. Ephemere pigment 7 is the source of a selection on three pigments : this is somewhat unusual since selections are quite often based on a couple of values, notably booleans. In the considered case, this means that once the *age* part is given a value, the state of this part evolves automatically towards 10, 11 or 12, depending on the *age* value, according to the partition defined by the conditions attached to these destination nodes. [The cited reference supposes *age* less or equal to 12 for a *child* (pigment 10); *age* between 13 and 19 for a *teenager* (pigment 11) ; *age* strictly greater than 19 for an *adult* (pigment 12).] Ephemere pigments 8 and 9 are used when the *age* is incremented.

c) Conjunction ; blends

Besides already mentioned nodes, the color graph exhibits also nodes 13 and 14. These nodes are termed **blends**. Each one results from a **conjunction** : blend 13 is equivalent to the pigment list (4, 10) and blend 14 to the pigment list (5, 10). These blends have a **degree** 2 : they group a pigment of two different scales. In our example, because the color graph degree is also two, they in fact happen to group a pigment of each scale : for this reason, these two blends may also be termed **colors**.

17 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 figure 2.

More generally, a blend has a degree d ($2 \leq d \leq N$) : a pigment has a degree 1.

Blends can always be removed without changing the semantics of a given color graph : a number of details (here, the *boy* and *girl* conditions) are to be expressed differently (this will be shown below). Here, blends are used since the useful ones (13 and 14) are few and allow a clearer expression than otherwise. This is not always the case : when many pigments exist in two or more scales, the number of possible pigments inflate (combinatorial explosion) ; if the number of useful blends is high, the user may reasonably prefer to do without them. Let's retain that blends may be useful in certain situations.

A color graph exhibiting pigments (and thus possibly blends) is termed a **p-graph** : it has several dimensions ($N \geq 2$). A color graph having but one dimension ($N=1$) is termed a **c-graph** : all its chromas are colors.

d) Reflex transitions

In the above figure, two types of unnamed thin arrows exist : in dotted lines for selections (ex.: 3-4, 3-5), in broken lines for conjunctions (ex. : 4-13, 10-13). Each thin arrow represents a **reflex transition**. This reflects that the semantics of a selection or conjunction can be expressed in terms of the simpler semantics of a reflex transition. This semantics is really simple : when armed, a reflex transition fires if the destination conditions gets true. (A reflex transition is armed if its source node belongs to the current instance state.) Yet not apparent, the semantics of a decomposition may also be expressed in terms of reflex transitions : however, a decomposition is not drawn using thin arrows so as to avoid a possible confusion with the selection (a diverging consruct too).

(In the following, we may well use condition names to refer to a chroma instead of its number : for example, pigment 7 will also be termed *age*.)

4.1.2 Regular transitions

Regular transitions between two chromas are used to specify the possibility for a message to be sent to an instance[18] and its effect on the instance state : the **source** and **destination** chromas respectively capture these informations (**simple** transition). Instead of two chromas, two sets of chromas may be necessary when blends are not used (**composite** transition).

a) User defined transitions

As it usually happens when using an appropriate decomposition, all transitions shown in the example are simple ones.

The *age:* message is used for initialization. It modifies a part of the instance state. This is reflected by the *age:* transition between the source pigment 6 and the destination pigment 7. The state change can be checked by testing the conditions *age* or *not age* : this testing is obtained by sending the *age* message to the instance, in others words by activating the **testing transition** *age*.

[18] or, more generally : ... for a generic function call to occur...

Similar comments can be made for the *sex:* transition.

Sending an *expected-lifespan* to one instance does not change its state : *expected-lifespan* is thus represented as a **circular** transition. *Expected-lifespan* cannot be sent when the *sex* is unknown (the value it returns depends on whether the instance is *male* or *female*) : to model this situation, one possibility is to attach an *expected-lifespan* circular transition to both *male* and *female* pigments. In fact, it is equivalent and simpler to factorize these two circular transitions in their father node, the ephemere *sex* pigment : the list of acceptable messages in *sex* is unchanged since only testing transitions can be sent in an ephemere pigment (these sendings are done automatically). The factorizing transition is termed **i-circular** : each individual transition it factorizes is itself circular.

For similar reasons, *bedtime* is represented by as an i-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 we can take 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 have two or three father nodes.
The proposed modelling of *have-birthday* with three transitions is quite precise (hence, the highest possible efficiency at run-time : only a single test to do in 8 or 9, and none in 12). A simpler, but less precise (and thus less efficient) one is possible : basically, it consists in adopting the ephemere node 7 as the destination of the three transitions : each such transition has now three possible outcomes (nodes 10, 11 and 12) instead of one or two : two tests will be done at run-time, but the correct destination will be found (robustness) ; the advantage is that these three transitions will in fact be factorized in a single one, a circular transition attached to node 7. This factorizing transition is not i-circular, but **g-circular** (g stands for group : the actual destination is to be chosen among the group of destinations of the selection). This examplifies a certain liberty in design.

Long-lived (which tells if a given instance has gone farther its *expected-lifespan*) is represented by a **constrained** i-circular transition attached to the *age* pigment (the constraint, termed a **clause**, is shown betwen brackets : it is the id of a blend or pigment). This happens 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 (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).

The underlying system may provide coherency checks using the selector syntax and additional declarations[19].

[19] By default, it may consider that circular transitions are for consultation only. In our example, this is actually the case. This also matches the selector syntax. Were *have-birthday* be factorized in pigment 7, it should be declared as modifying.

b) Testing transitions

Testing transitions are automtically derived from conditions. Normally, testing transitions are not shown in a color graph. (They may be on demand.)

In the example, testing transitions are *sex*, *male*, *female* (for the first scale) ; *age*, *child*, *teenager*, *adult* (for the second scale) ; *boy* and *girl* (valid not only for blends *boy* and *girl*, but for all fully initialized states). Testing transitions may exist in factorized form : being side-effect free, they are i-circular. Due to this property, *sex* is attached to pigments 2 and 3 ; *male* and *female*, to pigment 3 ; *age*, to pigments 6 and 7 ; *child*, *teenager* and *adult*, to pigment 7 ; *boy* and *girl*, to pigments 3 and/or 7 (both are required under some form).

Boy and *girl* are solely names for encapsulating the conditions at the destinations of the two conjunctions. As indicated above, suppressing the *boy* and *girl* blends is possible : in this case, a *boy* (resp. *girl*) transition should explicitely appear either as a pair of i-circular transitions attached to pigments 3 and 7, each one being constrained by the other pigment (composite transition) ; or by only one of these constrained i-circular transitions (constrained simple transition) : being circular, all but one may disappear. This illustrates the necessary adaptation of regular transitions (not necessarily testing ones, nor even circular ones) that originate from and flow to suppressed blends. (When suppressed blends are ephemere, the adaptation is a bit more complex.) The transition declarations described here is to be done by the user when blends do not exist : when blends exist, the same work is automatically and silently done by the underlying system (here, *boy* and *girl* will be installed as a constrained i-circular transition (either in 3 or 7)).

Testing messages *male* and/or*female* will automatically be sent in ephemere pigment 3 for choosing between pigments 4 and 5 ; *child*, *teenager* and *adult*, in ephemere pigment 7 for choosing between pigments 10, 11 and 12 ; *child* and *teenager*, in ephemere pigment 8 for choosing between pigments 10 and 11 ; *teenager* and *adult*, in ephemere pigment 9 for choosing between pigments 11 and 12.[20]

Other testing transitions are **constant transitions**, i.e. transitions that deliver a constant answer in a given chroma : for example, *sex* can be (externally) asked in pigments 2, 4 and 5, yielding constant answers (resp. false, true, true). Same remark for *male* and *female* in pigments 4 and 5 ; *age* in pigments 6, 10, 11, 12 ; *child*, *teenager*, *adult* in pigments 10, 11, 12. Of course, deriving from the constant answers obtained in the above-mentioned pigments, constant answers are also got for these messages in *boy* and *girl* blends (as well as for other fully initialized instances.) *Boy* and *girl* can be sent to any fully initialized instance, yielding constant answers : *boy* returns true only in blend 13 ; *girl*, only in blend 14.

[20] One test is normally avoided each time. However, if the program is not to be trusted, the debug mode will be chosen : no optimzation will be done ; all tests will be run so as to detect an erroneous case (partition condition unsatisfied).

c) Creation transition

Only one **creation transition** exists in the color graph, the default one (name : *make-instance* ; initial state : 1).

4.2 Dynamic description

4.2.1 Current state

The *Person* color graph describes the possible states of *Person* instances. To mark the current state of one instance, we use two **mini-tokens**, one for each dimension. (We reserve the term **token** for color graphs having but one dimension.) The figure shows a possible configuration with one mini-token (the *sex* one) in 2, and another one (the *age* one) in 10. Each mini-token may be either in one pigment of its dedicated scale or in a blend : in the latter case, the two mini-tokens must be grouped. (A blend of degree d is a place holder for d mini-tokens.)

The current state is **stable** if no reflex transition may fire. An instance in such a state remains in it as long as no acceptable message occurs. For example, state (2, 10) is stable : the *sex* mini-token cannot move from 2 until a *sex:* message occurs ; the *age* mini-token cannot move either since no reflex transition outcoming from 10 (conjunctions) may fire as long as the *sex* mini-token remains in 2.

4.2.2 Acceptable messages

Given a stable state, acceptable messages are defined by all **valid** outcoming transitions, either **locally defined** or **inherited**. (A constrained transition is valid if the instance state satisfies its clause. An unconstrained transition is systematically valid. Regular transitions are inherited along the reflex transitions.) For example, considering the stable state (2, 10), transitions *sex*, *sex:* and *have-birthday* are locally defined while *age*, *bedtime*, *child*, *teenager*, *adult* are inherited[21] ; transition *long-lived [3]* is not valid.
For a fully uninitialized instance, acceptable messages are *sex:* and *age:* as well as *sex* and *age*; for a fully initialized instance (like *boy* or *girl*), they are *sex*, *male*, *female*, *expected-lifespan*, *age*, *bedtime*, *have-birthday*, *child*, *teenager*, *adult*, *long-lived*, *boy*, *girl*. A partially initialized instance accepts *age*, *bedtime*, *have-birthday*, *child*, *teenager*, *adult*, plus *sex* and *sex:* (if *age* is initialized) ; *sex*, *male*, *female*, *expected-lifespan* plus *age* and *age:* (if *sex* is initialized).

4.2.3 Computing the next state

a) Regular case

When an acceptable message is received by an instance in a stable state, the corresponding transition (possibly a composite one) fires, thus making one or several mini-tokens move to another chroma(s) ; this may in turn yield the firing of one or several constructs, i.e. the firing

[21] The adjective "inherited" is used here from a purely topological point of view : it does not mean the transition should be meaningful in the node from which it is inherited : for example, *bedtime* in 10 is inherited from 7 simply because it is factorized in 7 even if *bedtime* cannot be sent to an instance in 7.

of one or several reflex transitions, possibly in a recursive manner, possibly via the automatic sending of testing messages (automatic firing of testing transitions). For example, if a *sex:* message is sent to an instance in state (2, 10), then the *sex:* transition fires. The messages *male* and/or *female* (testing transitions) are automatically sent, making (for instance) the reflex transition 3-4 fire ; this, in turn, makes the *boy* conjunction fires (reflex transitions 4-13 and 10-13 fire) : the two mini-tokens defining the state of the *Person* instance get grouped into node 13.

b) Backtracking case

The above schema is to be tuned when a regular (inherited) transition does not move all the mini-tokens its source node contains. To be more precise, the **degree** of a regular transition is defined : if unconstrained, it is the degree of the chroma in which it is locally defined. (For example, the degree of *bedtime* (in 7) or *have-birthday* (in 10, 11, 12) is one.) When the transition is constrained, its degree results from the degree of its source chroma and of its clause. (For example, the degree of *long-lived[3]* is two.) In a p-graph, the degree of a transition defines the number of mini-tokens involved for consultation and/or modification.
Circular (i-circular) transitions do not induce any move.

In a p-graph, a transition of degree d may be inherited in a chroma (blend) of superior degree (D>d). For example, transitions *bedtime* or *have-birthday* (of degree 1) are inherited in *boy* or *girl* (degree 2). If the inherited transition is i-circular (like *bedtime*), the moved mini-tokens stay in their initial position : hence, no complication occurs. This is usually not the case for non i-circular transitions : for example, *have-birthday* moves only the *age* mini-token. To cope with such a situation, mini-tokens which are not moved by the outcoming regular transition are first placed back into their original pigments (backtracking phase), the moving ones being placed at the source(s) of the transition ; then all proceeds as explained before.

Consider, for example, a *boy* instance. Suppose an *have-birthday* message is sent. The *boy* instance may become a (*male teenager*). The new state may be computed as follows : the mini-token which is involved in the *have-birthday* inherited transition (the *age* mini-token) is first moved to the transition source chroma (here, *child*) ; this invalidates the blend *boy* : the mini-token which is not involved in the inherited transition (*sex* mini-token) is thus moved back to its original pigment (here, *male*). Then the transition gets fired : the *age* mini-token remains in *child* or moves to *teenager*. In the first case, the conjunction (*male child*) fires : *boy* defines again the instance state. In the second case, the instance state is defined by (*male teenager*). A less precise modelling of the *age-birthday* transitions, factorizing all cases in node 7 (g-circular transition) will produce the same result..

5. POTENTIAL : A CONCEPT FOR REFINED SPECIFICATIONS

5.1 Motivation

The current definition of transitions (possibly, multi-transitions involving several specializers) is a simple declaration (**deftransition**) listing besides the name of the transition (selector) and possible keywords (qualifiers), an augmented argument list : each required argument is followed by its class name and a list of chroma pairs. No body is given. On the other hand, the definition of a color graph (**defcolorgraph**) lists chromas together with the conditions (**:test** or **:test-not** conditions) and/or the way they are constructed. No body is given for evaluating the conditions. This is but when methods and memory representations are attached to the color graph (see companion paper n° 3) that a color graph may be animated.

To overcome this point, a new concept was thus imagined, the **potential**. Besides the animation of a color graph for early evaluation, this concept enables theoretical proves (in absence of a smart prover, the user has to do the job manually) : (a) a color graph may be checked for consistency ; (b) optimizations may be discovered (some tests for identifying a destination state in a selection may be avoidable) ; (c) the best specification of a mixin color graph may be elaborated given a minimal set of constraints (companion paper n°2 provides an example). Finally, a specific implementation may be monitored at run-time.

5.2 Definition

Each (mini-)token is attached a value, most usually an integer, that can be (1) initialized when the (mini-)token is created ; (2) updated when a transition gets fired (using or discarding the arguments of the message) ; (3) tested for conditions evaluation. This value is termed the (mini-)token **potential**.

Above rules correspond to state an abstract body for transitions and conditions. By abstract, we mean that the computations involved in a possible animation of a color graph have a priori nothing to do with an actual implementation. (For example, an integer may be used as a potential in a *Stack* color graph while actual *Stack* instances may be implemented, say, with a list.) Because it depends on the operations to be carried out, the type of value for a potential is not restricted a priori. (For example, knowing the longest sequence of successive *push:* sent to a *Stack* instance will not be possible using a potential of type integer : theoretically, some questions may not receive an answer without holding all the (mini-)token history, i.e. without storing a list of all received messages including their arguments).

5.3 Examples

5.3.1 *Stack* color graph

The *Stack* color graph is simple : its main construct is a selection on two basic colors, *empty* and *not empty*. The potential concept abstractly refines the specification of its transitions : (1) the initial value of the *stack* potential

is zero ; (2) a *push:* transition increments the potential by one (regardless of the argument value) ; (3) a *top* transition does not change the potential ; (4) a *pop* transition decrements its value by one ; (5) the *empty* condition is true when the potential is zero ; (6) (for a *Bounded-Stack*) the *full* condition gets true when the potential gets equal to the *bound*.

5.3.2 *Person* color graph

The comments we made about the effect of transitions in the *Person* color graph were often based on an intuitive meaning of its transitions (ex. : *have-birthday*) and conditions (ex : *child*). The concept of potential makes them more substantial.

Let's name *age* (resp. *sex*) the potential of the *age* (resp. *sex*) mini-token. At creation time, the *age* potential is given a zero value (pigment 6). The *age:n* message is used to initialize the *age* potential : it is set to the argument *n* (pigment 7). The *age* condition is specified to be (*age* > 0) ; the *child* one, (age ≤12) ; the *teenager* one, (13 ≤*age*≤19) ; the *adult* one (*age*≥20). The transition *have-birthday* will be specified as incrementing the *age* potential by one. Etc.
Likewise, at creation time, the *sex* potential is given a zero value (pigment 3). The *sex:s* message is used to initialize the *age* potential : depending on the argument value, it may be set to 1 or 2 for example (pigment 7). The *sex* condition is specified to be (*sex* > 0), the *male* one, *(age = 1)* ; the *female* one, *(age=2)*.

5.4 Enhancements

As it is transparent in these examples, a number of enhancements appear useful to maintain a clear separation between the potential concept and the actual implementation, while avoiding the useless duplication of efforts. For example, specifying the list of possible values for each potential (at the color graph level) seems useful because a color graph is meant to be more stable than a particular implementation. This implies the definition of mapping functions for obtaining a potential from actual arguments or from an actual implementation that needs to be looked after for debugging or monitoring. Conversely, as shown by the *Person* example, it might well be desirable for an actual implementation to rely much on its abstract specification so as to limit the implementation effort.

The potential concept is taken advantage of for animating one color graph. To animate the color graphs of a group of interacting objects, we have to go further and explicitely mention –at the color graph level– the relationships between objects in terms of exchanged messages and answers : this corresponds to specify in much more detail the abstract bodies of transitions. Reference [Coleman-Hayes-Bear, 1992] offers interesting tracks with the specification –at the objectchart level– of services required from another objects (input and output parameters are specified : an input parameter may only be read by the object providing the required service ; an ouput parameter is set by the providing object).

6. RELATED WORK

COP derives in direct line from OOP and happens to be related to higraphs/statecharts. These two relationships are analyzed in the following two subsections.

6.1 Object-oriented programming

6.1.1 OOP in general

As a fervent of OOP, we developed this particular formalism with the idea of pushing further the basic idea of OOP, which –in our view– is to give objects the reponsability they can handle [Borron, 1996e]. With this view in mind, it is quite natural to take into account object states when possible. Once this step is made, OOP appears somehow as having forded the river half way : the idea supporting OOP is excellent, but appears as having been not pushed to its ultimate whenever possible. Considered from the COP bank, a traditional class is like a single point (it offers a single dictionary of methods). Whereas COP distinguishes several coordinates on each dimension and thus exhibits a number of points for a given class, traditional OOP aggregates them in one. This difference accounts for the original approach of inheritance proposed in companion paper n° 3, and the progress it enables. To take a metaphor, traditional OOP considers a class like an atom whereas COP opens it and reveals an internal structure. This deeper understanding enables a better modelling : concepts get purer. In our view, this is the profound reason that enables us to get rid of the *send-super* antimodular OOP construct (termed **super** in Smalltalk and **call-next-method** in CLOS), thus enabling a pure declarative programming style, and not a mixture of imperative and declarative styles as regretted by Sonya Keene, a member of the CLOS committee [Keene, 1989]. [Borron, 1996e] develops this point of view in detail.

6.1.2 Predicate classes

The *Person* color graph depicted in this paper is derived from the actual code of the *Person* example given in [Chambers, 1993]. This work, the closest to ours we know of when considering the strict OOP filiation, promotes the concept of "predicate classes".

a) Comparison

Like regular classes, "predicate classes" specify slots and methods that affect the behaviour of objects. But, different from regular classes, they are attached a predicate (what we named a condition in color graphs). An object, instance of a given regular class having "predicate classes" as subclasses, behaves as if it was an instance of one (or several) of these classes if this object meets its (their) predicate(s). In principle, a systematic dynamic testing of all possible predicates is to be done on reception of each message[22]. The author of the cited paper explains optimizations are possible.

This approach is quite interesting in that no separate mechanism –roughly speaking– is necessary : the basic

[22] Of course, the predicate of a p.c. that ands other p.c. is not to be tested directly.

concept of class is enlarged ; the inheritance hierarchy and class properties are taken advantage of. This is a very good example of reuse in design. The implementation strategy described in the cited paper consists in caching –in the instance– the result of the predicate as an internal subclass (it acts as a color) and using it to directly access the adequate dictionary. This is akin to having a dictionary per color. *"To record the outcomes of multiple independent predicates, internal combination subclasses can be constructed lazily as needed"* (p. 283). This parallels the combination of scattered dictionaries when the state of an object is expressed in a p-graph.

Let's now stress two important differences with COP. These are :

(1) the level of abstraction : color graphs specify an abstract programming interface while predicate classes consist in actual implementation code ;

(2) the (structural) specification of the next state : in color graphs, a graph transition has a destination that indicates either exactly the next state or a selection of possible states.

-> In the first case, no run-time testing needs to be done : if the state is different, a pointer change is all what is required ; if the state happens to be the same, the method dictionary is not even changed (no cost) ;
-> In the second case, the selection is the narrowest possible one if the color graph has been devised correctly[23] : to discover the actual next state, one or several condition evaluations are to be done ; these evaluations cannot be avoided.

On the opposite, predicate classes do not know the next state after a method has been executed, hence –in principle– a run-time testing to discover the current state of an instance when it receives a message : the implementation of predicate classes is thus necessarily highly optimized to avoid these run-time penalties. In the COP approach, these required optimizations are somehow directly captured into the color graphs (by the specification of transition destinations) : optimizations are thus done without effort. To summarize, a color graph provides –without sophisticated optimizations– a code which eliminates unnecessary tests.
Another practical consequence of the specification of transition destinations is visible at a more high level : one can derive a predicate class hierarchy from a color graph whereas the opposite cannot be done. Somehow, we can state that "COP = predicate-classes + destinations".

An implementation of COP using predicate classes is therefore possible provided that the destination informations are used at run-time for systematically updating the instance state. (Various solutions can be envisaged : one that preserves method modularity is to produce special **after:** methods.) In this way, the benefits of COP in terms of efficiency-without-optimization are maintained.

b) Example

The next figure visually shows the actual class hierarchy of the*Person* example as it is expressed by the code in [Chambers, 1993]. Predicates classes are shown in rounded rectangles ; methods, as tiny circles. The *make-person* method, not represented in the figure, creates a new instance and initializes it, thus encapsulating the code implementing a *make-instance* creation method followed by a *sex:* and an *age:* initialization methods (such an encapsulation is easily obtained in COP).

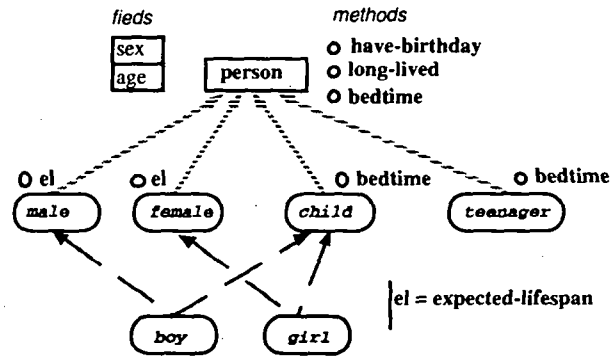


Figure 3.

From the *Person* color graph of figure 2, one can directly infer the predicate class hierarchy of figure 4.

The transposition is almost immediate. Basically, a node is mapped to a predicate class : nodes 2, 3, 6 and 7 yield direct predicate subclasses of the *Person* class ; except for nodes 8 and 9, the structure for the remaining nodes is kept unchanged. Nodes 8 and 9 can be mapped as direct predicate subclasses of the predicate subclass *age* or of class *Person* : in fact, since they do not hold methods (nor memory representations), corresponding predicate classes can be removed : this is coherent with the fact that the destination information is structurally lost in predicate classes[24]. A new predicate class is made to hold the *long-lived* method since this one corresponds to a constrained transition : the new predicate class ands the *sex* and *age* ones.

One reason for the difference between figures 3 and 4 is that the actual code of the cited reference does not take into account a more modular expression of the *age* and *sex* properties. A second reason was already mentionned : for demonstration purpose, our *Person* example was made more general than Chambers'one (our color graph does not take into account that existing *Person* instances are systemtically initialized in the original example). A third reason is apparent in figure 3 : no *adult* predicate class exists in the code of the cited paper. In COP, we systematically require a selection to describe all the possible cases : a mathematical point of view

23 The *Person* color graph illustrates this point : cf. the discussion of the *have-birthday* transition in §4.1.2.a.

24 To keep the destination informations while preserving method modularity, a solution is to encode in two *have-birthday* **after:** methods the choice restrictions these ephemere nodes represent. Strictly speaking, the information is preserved but its obviousness is degraded ... if no precaution is taken (analyzing normal method bodies for destination extraction is a priori difficult). A possibility is to maintain the color graph externally and to never attempt to extract back the destinations from method bodies. A quite different possibility is to produce the bodies of these specialized **after:** methods in a systematic way so as to be able to easily extract the destinations back, and to distinguish these methods from regular methods by a mark : a destination extractor will then be able to reconstitute the color graph from the sole methods. In this solution, no external structure is thus necessary to obtain and maintain the color graph.

underlies that. This is not to say that an implementation may not support the parent class as a default and remove one possible case.

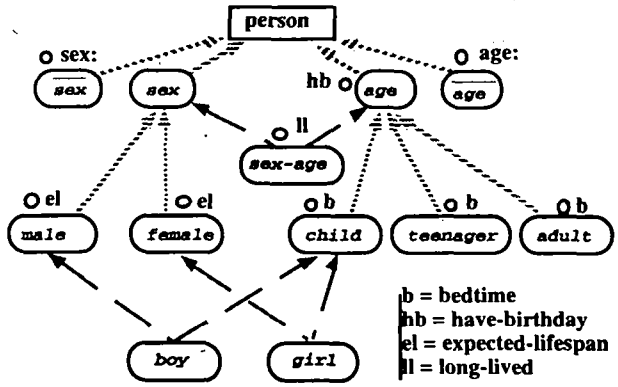


Figure 4.

Figure 3 can easily be obtained from figure 4 by suppressing a number of predicate classes :

— taking as an hypothesis that any instance is created and fully initialized by *make-person*, predicate classes corresponding to uninitialized pigments are useless ; at the same time, the predicates of the *sex*, *age* and *sex-age* p.c. are systematically true : these p.c. can be merged with the *Person* class itself ;

— the p.c. corresponding to the *adult* basic node can be removed too : the *Person* class is then considered as a default.

Since the attachment of methods and memory representations to a color graph is done in companion paper n°3, we should normally differ the completion of the comparison to this paper. As an exercise, let's prefigure its results :

— a memory representation with two cells (i.e. slots), one for holding a *sex* value and one for holding an *age* value, clearly convenes. This systematic representation is better factorized in the *Person* class (figures 3 and 4) ;

— in figure 2, the attachment of methods is not difficult to guess either. Undoubtedly, the *expected-lifespan* transition requires two different methods for *male* and *female* while three*bedtime* methods are necessary for *child*, *teenager* and *adult*. Obviously too, the *have-birthday* method can be shared in node 7. The transposition of this COP implementation to figure 4 is immediate : basically, the attachement of methods is left unchanged by the mapping except for the *long-lived* method : as already mentionned, it is attached to the new *sex-age* predicate class. Given the above mentioned suppression of predicate classes, figure 3 is easily obtained from this : the*have-birthday* and *long-lived* methods migrate to the *Person* class ; the *bedtime* method for *adult*, being considered as a default, also migrates to the *Person* class.

(Other examples from the cited paper may be considered for exercises.)

6.2 Higraphs / Statecharts

The relationship between color graphs and the visual formalism proposed by David Harel is almost a coincidence (historically, we discovered higraphs and statecharts after our design of COP). Actually, it is a mere consequence of our decision to push the OOP main idea to its ultimate. Because of our intent to make objects more responsible than in OOP, we introduced a notion of state in OOP, sublimated the concept of message into the regular transition concept and defined reflex transitions too. Hence, an approach that naturally leads to graphs and, from there, to the design of a visual formalism.

From the strict point of view of the graphical representation, this new formalism happens to be an alternative to higraphs or -more precisely- to statecharts. As explained below, our color graphs are based on connectedness instead of insideness for higraphs and statecharts : multiple ancestors are thus possible and this makes our formalism to be a bit more malleable than its alternative.

The rest of the current subsection develops these points. It is organized in the following way : first, the correspondence between color graphs and higraphs/statecharts is given ; an example follows ; then the comparison is argumented in detail.

The next subsection focuses on objectcharts, a refinement of statecharts. This formalism incorporates OOP and can thus be discussed also from a semantic point of view, including inheritance.

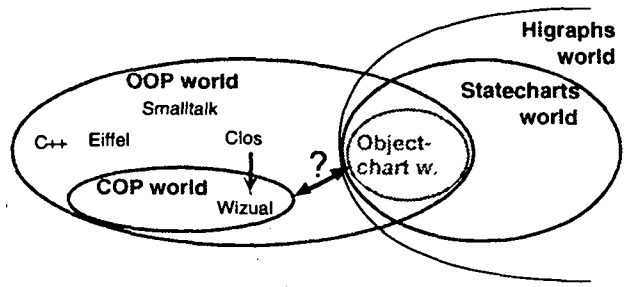


Figure 5.

6.2.1 Correspondence between color graphs and higraphs

A strong connection can be made between color graphs and higraphs. The key point concerns what we name reflex transitions : in color graphs, they are materialized using connectedness. **Replacing connectedness by insideness** remarkably leads –modulo the a priori arrangement of certain details[25]– to "higraphs" as

[25] For example, in higraphs, conditions are attached to edges (transitions), but not to blobs (chromas). In color graphs, conditions are systematically attached to chromas (as predicate to predicate classes), and sometimes to transitions too (these extra conditions are termed clauses). All transitions outcoming from a chroma thus share the condition of this chroma. This is natural given the underlying semantics of color graphs. Translating a color graph to a higraph should better keep this factorization of conditions and associates this "common condition" to each blob. Same rationale for keeping the testing transitions unshown.

exposed in [Harel, 1988] and its companion papers about "statecharts" [Harel et alii, 1987] [Harel, 1987].

1) In higraphs, colors are termed "blobs"; and pigments, too. No equivalent to blend exists.
2) An ephemere color/pigment is mapped to an encircling blob (for example, the *sex* blob encircles the *male* and *female* ones in the *Person* example).
3) Regular transitions correspond to directed "edges" or, when flowing to ephemere nodes, to "hyperedges".
4) A color graph decomposition exactly corresponds to "partitioning" in higraphs[26].

6.2.2 Example

Next figure shows how the *Person* color graph maps to a higraph or -more precisely- to what we name a "color higraph" : conventions in such a graph are the same than in color graphs except for the reflex transitions (which are represented using insideness), and for the decomposition (which is expressed as partitioning, using a vertical dashed line).

In a preliminary phase, multiple converging reflex transitions must be eliminated from the color graph. This is because insideness has a less intrinsic power of representation than connectedness. Connecteness (color graphs) allows the representation of DAGs ; insideness (higraphs) is meant to represent strictly hierarchical relationships (trees). In our example, nodes like 8, 9, 14 and 15 of the *Person* color graph (figure 2) cannot be represented in higraphs.

— Ephemere nodes 8 and 9 were meant to respect the "unique destination" principle concerning the *have-birthday* transition. This principle is the equivalent of the acclaimed "depth" one in the higraph world : we see here that this "depth" principle cannot be obeyed systematically in higraphs. Forgetting the "unique destination" principle is obviously possible concerning color graphs : we discussed this point in [Borron,1996b]. In this case, ephemere nodes 8 and 9 simply disappear from the *Person* color graph ; two *have-birthday* transitions flow out of node 10 and two of node 11. In the *Person* higraph, one cannot do otherwise. Or almost : it is only possible to keep one encircling blob, not two (if blob 8 exists, it encircles blobs 10 and 11; if blob 9 exists, it encircles blobs 11 and 12 ; blobs 8 and 9 thus intersect). If we adopt one encircling blob (8 or 9), then the structure is awfully not symmetrical vs. the *have-birthday* transitions outcoming from blobs 10 and 11. Figure 6 is thus meant to reflect the best possible choice.

— Blends 13 and 14 represent the *boy* and *girl* states in figure 2 (we can imagine one may want to possibly attach them a special behaviour).

With the higraph formalism, these states cannot be represented as such, i.e. using blobs. (Reference [Borron, 1996b] explores a number of extensions, but all seem somehow awkward, external to the insideness philosophy of higraphs, and difficult to use on a large scale.) The rationale behind such a restriction is precisely the fear of an explosion blow-up problem. However, in situations where only a few number of all the possible combinations are useful –as illustrated by the *boy* and *girl* blends in the *Person* example, this restriction is counterproductive since it plays against comprehensibility.

Given this, color graphs appear more malleable.

What we have done in the next figure is to represent the *boy* and *girl* transitions.

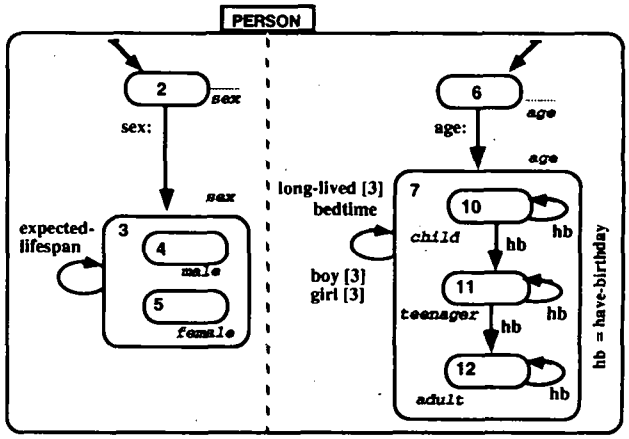


Figure 6.

Note these transitions are constrained : hence, an extra cognitive effort to identify the clause contents (here, blob 3), to correctly interpret each such transition as characterizing an interesting state and to memorize the result. (In the *Person* color graph, these transitions exist ; but –as a result of a convention– they are normally not shown except on demand : they naturally stem from the existence of the conditions *boy* and *girl* of the two blends.)

Suppose a regular transition is now to be added to the *boy* (or *girl*) state as it happens with the *draw* transition for *Circle* instances when fully initialized. This is easily done in the *Person* color graph and the result is immediate to grasp since the transition is directly attached to *boy* (or *girl*). Using a higraph, the transition cannot be attached to a *boy* (or *girl*) blob since such blobs do not exist : it should be attached to an other blob (ex. : *child*) and constrained. The resulting higraph representation thus implies an added cognitive load. For transitions flowing to an ephemere blend, the result is more complex and requires more effort.

6.2.3 Comparison between color graphs and higraphs

1) Connectedness vs. insideness : a recurrent debate

As reported by references [Fitter-Green, 1979] and [Green, 1982], the insideness vs. connectedness choice has been the subject of a debate for expressing hierarchical relationships in the early seventies : [Nassi-Schneiderman, 1973] and [Jackson, 1975] had both their adherents at that time. This choice is also very clearly expressed in [Harel, 1988] where, for example, set-theoretic relationships in entity-relationship diagrams are more elegantly and cheaply expressed by insideness than by connectedness.

[26] In both formalisms, such features are meant to avoid a combinatorial explosion of states, an important consideration indeed. This makes the parallel quite striking.

2) Good properties of higraphs are valid for color graphs

a) higraph claims

In his papers, Harel insists much partionning as well as on hyperedges. For example, [Harel et alii, 1987](p. 54) lists the reasons why *"people working on the design of really complex systems have all but given up on the use of conventional FSM's and their state diagrams"* :

(1) *"State diagrams are flat"* ;

(2) *"State diagrams are uneconomical when it comes to transitions"* ;

(3) *"State diagrams are extremely uneconomical, indeed quite infeasible, when it come to states"* ;

(4) *"State diagrams (...) do not cater naturally for concurrency"*.

Then, about the idea of depth (i.e. the hyperedges), the same paper states : *"This simple idea, when applied to large collection of states in a multi-level manner, overcome points 1 and 2 above."* (p.55). About the partitionning, it explains : *"If the orthogonality construct is used often, and on many levels, difficulties 3 and 4 above are overcome in a reasonable way"* (p. 55).

b) these claims are valid for color graphs

As noted in subsection 6.2.1, the decomposition construct in color graphs is equivalent to the partitioning in higraphs, while transitions to ephemere nodes in color graphs are equivalent to hyperedges in higraphs.
If we consider a given higraph, a color graph equivalent can be built using this correspondence. Concerning this equivalent, the above claims are obviously maintained.

Note this equivalent is particular. It is purely hierarchical. It does not exhibit multiple ancestors (blends, in particular). Let's suppose we systematically prohibit blends (as done in higraphs) and -more generally- multiple ancestors. This is possible : as already announced, blends in color graphs are strictly optional and the application of the "unique destination" principle, the other source of multiple ancestors in color graphs, may be relaxed. Then, for this restricted form of color graphs, the above acclaimed qualities of higraphs can undoubtedly be stated too.

Next subsection shows that this restriction about color graphs can be rid off in practise.

3) Color graphs are more malleable than higraphs

a) this is not dangerous in practise

What about allowing multiple ancestors ? If they result from the "unique destination" principle (the "depth" principle) in color graphs, then no danger is to be feared.

The only problem that may be encountered lies in the use of blends (combinatorial explosion problem). However, the programmer is by no way forced to use them. He/she may do so when the useful ones happen to be few (as in the *Person* example). This is like having a road in a mountain : driving too fast is known to be dangerous, and people thus drive with caution. When possible, they speed up ; otherwise, they go slowly. There is no real need do deploy state police forces all along the road for systematically imposing a low speed ! Programmers will use blends appropriately (taking the opposite point of view is like hypothesizing that programmers are fool).

b) this is in fact an advantage

The capacity of supporting multiple ancestors is an advantage for color graphs vs. higraphs.

As examplified by the *Person* class, the color graph formalism enables a systematic observance of the acclaimed "depth" principle ("unique destination" principle), which is not true for the higraph formalism. (Note that we intend to propose a flexible environment in practice, one where transitions like *have-birthday* in nodes 10 or 11 can be represented as in figure 2 or figure 7, i.e. respecting or not the "unique destination" principle.)

Concerning blends, their appropriate use a priori enables a better representation of composite transitions : a simple transition appears more immediate to interpret than one or several composite ones with their clauses. The drawback lies in the existence of these extra nodes (blends) with their reflex transitions.

Our proposal thus consists in preferring malleability to rigidity : depending on the complexity of the class to be modelled, the user is invited to choose among several possible representations. With practice, a good representation (if not the best possible one) is to be expected. As in other forms of programming, users will develop personal styles, some of which will appear better than others.

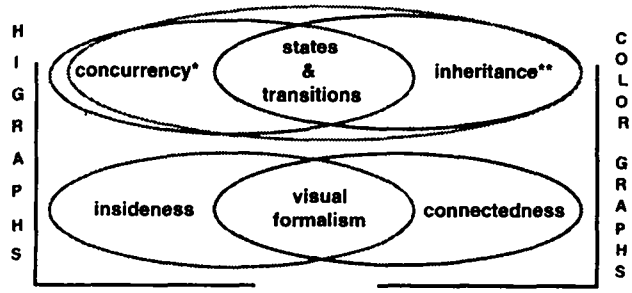
This malleability may be important in practice since, to quote [Green, 1982], (p. 34), *"The details of the notation profoundly influence its usability (...) A saving too small to be measured [on a small scale] could become a major improvement a thousand times up."*

4) Limitations

Whereas statecharts are meant for reactive systems, like operating or avionics systems, communication networks, ... (statecharts have primitives for concurrency), color graphs are currently restricted to transformational systems.

This restriction is not due to a theoretical reason : the mechanisms proposed for statecharts may in fact be incorporated in color graphs.

Like its equivalent formalisms, our proposal has a more profound limit : a finite number of states are required. Thus, if continuous values are used, this condition may not hold.



(*) Concurrency mechanisms may be adapted to color graphs.
(**) Objectcharts support (a form of) inheritance.

Figure 7.

5) Partial conclusion

The discussion of the *Person* example shows that human factors are more complex, intricated and subtle than one may think at first sight. If a question exists, it is not about the compared theoretical powers of expression of the two formalisms (due to their mathematical grounding, they are basically identical for these applications), but about their compared ergonomics[27] (including the single/multiple ancestors aspect).

The ergonomic aspects of color graphs were seriously thought about [Borron, 1996b]. They integrate practical as well as conceptual reflections (ex. : [Green et alii, 1981] [Green, 1989] [Green, 1990])). And the above comparison shows that this formalism brings a useful malleability.

To avoid the biaises due to habits and/or personal tastes, we intend to collaborate on a careful experiment such like [Green, 1977] [Fitter-Green, 1979] [Green, 1982] [Gilmore, 1986] [Petre-Winder, 1988] [Soloway-Bonar-Ehrlich, 1988] [Scanian, 1989]. (The same pragmatic and objective attitude was taken in [Harel, 1988], p.519). It will be particularly interesting to measure how much the extra malleability of color graphs is going to be used. Enhancements are also supposed to result from the study.

6.3 Objectcharts

"Objectcharts are extended statecharts in which the effect of state transitions on [object] attributes are specified" [Coleman-Hayes-Bear, 1992], p.12. In certain states, attributes of an object may be reported on via additional services termed "observers". Hidden attributes are specified as such, but are treated as observers. Observers are named, but not represented by arcs. (We support this feature too.) Necessary input/ouput parameters are specified for each service (we already mentioned this in subsection 5.4.) Besides the service name, the initial and final state names, a transition specification also comprises a firing condition and a postcondition. The former is a predicate which may mention state names, attributes and observers ; the latter is *"a predicate on the initial and final values of attributes and observers which characterizes the effect of the transition"* (p.13).

27 The interested reader will find in [Borron, 1996b] additional arguments about the ergonomic aspects of color graphs vs. higraphs.

What is the correspondence with COP ? A firing condition implemented with attributes is like an abstract condition body implemented with potentials ; a postcondition, like an abstract transition body. Yet, potentials are abstract (they relates to an invariant specification) whereas attributes are part of objects .

Another work is reported in [McGregor-Dyer, 1993]. It uses *"a style of state diagram similar to [the] objectcharts"*. Yet, it introduces a new notation, a vertical box containing *T(rue)* and *F(alse)*. This notation *"reduces the complexity of the diagram slightly"* (p.66). As a matter of fact, this new notation corresponds to the selection construct, yet in a less general manner. The extended objectchart formalism is thus based on insideness as well as connectedness. One figure of the cited paper, the *Queue*, corresponds almost exactly to a color graph (connectedness only).

The inheritance aspect of objectcharts are discussed in companion paper no 3.

7. CONCLUSION

The proposed formalism is visual and object-oriented.

7.1) An extremely object-oriented formalism

The proposed formalism is **object-oriented**. As a matter of fact, it has been designed for pushing object orientation to its utmost : it thus incorporates the concept of instance state and models the effect of messages (or generic function calls). A color graph synthesizes the whole behaviour of a class of objects. (The term "whole" reflects the invariance of a behavioural description vs. its implementation, flat or highly hierarchical).

The formalism is abstract : this paper is not concerned with the implementation of objects in terms of methods or memory representations ; class inheritance is not considered at this level either (these points are the subject of companion paper n° 3).

The formalism is **highly structured** :

(1) instance states may be described according to one or several dimensions (hence the concepts of colors, pigments and blends) ; essential constructs exist for describing AND or OR relationships between these states (decomposition, conjunction and selection constructs) ; these constructs are themselves built upon a more basic concept, termed the reflex transition ; conversely, above the essential constructs, more abstract constructs also exists, like the mixin and the derivation ones (cf. companion paper n° 2) ;

(2) a condition (evaluated by sending a side-effect free message to the instance) is attached to each chroma (color, pigment or blend) : this condition may be defined by the user and/or is inferred by the underlying system

(on the basis of constructs) ; the associated testing transition is automatically derived by the system ;

(3) acceptable messages and their effect are modelled by regular state transitions. These are normally depicted by multiple constrained graph transitions according to the color graph dimensions : two conventions were adopted to simplify the representation of regular graph transitions and to possibly eliminate the clauses (i.e. constraints) : five other principles are also used for designing simple color graphs, among which the "unique destination" principle and the "non ubiquity" principle. Testing transitions are normally not shown. Factorization facilities have been set up for factorizing regular transitions at selection destinations : more generally, regular transition inheritance (along void transitions) can be taken advantage of for devising a simple expression of regular transitions. Given a color graph, we show how to compute the dictionary of acceptable messages for a given state ;

(4) to materialize the current state of an instance in a given color graph, a mark is used : it consists in a set of mini-tokens, one per dimension (the term token is used in a 1-dimension color graph) ; the paper describes the effect of constructs and regular graph transitions on (mini-)tokens ;

(5) in addition to the color graph formalism itself, a supplement of behavioural specification is handled via the potential concept. A potential is a value that is attached to each (mini-)token. It can be initialized (when the (mini-)token is created), modified by a triggered transition (using its arguments or discarding them) and tested (for condition evaluation). A potential is abstract : a priori, it has nothing to do with a possible implementation.

7.2) A visual formalism

7.2.1 acceptability

The proposed formalism is visual. Generally speaking, a visual formalism is not expected to satisfy everyone since some people are more "auditive" than "visual" : because their preferred way of thinking is not based on visual representations, the use of a visual formalism is far from being intuitive for them : it does require an important cognitive effort from their part. The visual formalism presented here is not expected to be an exception and hence cannot be reasonably expected to especially attract these people.

However, some other people are preferably oriented towards "visual" or "kinetic" mental representations. In 1945, a survey done about the creative methods of mathematicians reported on the mental pictures of such -a priori- abstract scientists [Hadamard, 1945]. Albert Einstein replied that *"the elements in [his] thought [were] of visual and some of muscular type"*. Jacques Hadamard wrote : *"My mental pictures are exclusively visual."* Being in charge of the survey, he conclued that, for almost all the polled mathematicians, *"The mental pictures (...) are most frequently visual, but may also be of another kind -for instance kinetic"*. [Larkin-Simon, 1987] acknow-ledges this fact and shows why it is advantageous to use diagrams. They display important informations otherwise only implicit in a textual representation and very costly to compute from it. The visual formalism presented in this paper follows this line.

7.2.2 malleability

The visual formalism we propose happens to be essentially[28] equivalent to the higraph/statecharts formalism developed by David Harel : whereas the higraph formalism is based on insideness, our formalism is based on connectedness. This makes it more malleable than its alternative (a point which may be important in practise) while not loosing any strong aspect of higraphs.

A higraph representation can always be mapped to a purely hierarchical color graph. When appropriate, this first color graph can itself be upgraded to a better color graph (same semantics, yet expressed in a simpler way). For example, in some situations, a color graph will better be expressed with blends (higraphs do not support them) while, in other situations, it will be better to avoid their use (as systematically done in higraphs).

This brings an important consequence : all the acclaimed qualities of higraphs/statecharts are valid for color graphs[29]. In particular, there is no question about how well color graphs will scale up with complex applications. Complex applications have been built using statecharts : the color graph formalism will thus undoubtedly be able to support them equally well (at least).
Another immediate consequence of the correspondence between statecharts and color graphs is worth to be noted : the primitives used in statecharts for specifically supporting reactive systems may be imported in color graphs (restricted in this presentation to transformational systems).
Conversely, this relationship implies that parts of our work can be imported into higraphs, statecharts and derived formalisms, notably two aspects that are developed in companion papers n° 2 and n° 3 :
— our proposal for handling mixins (to the best of our knowledge, our mixin and derivation constructs do not exist yet in Harel's formalism nor in the derived ones) :
— our proposal for inheritance by dimensions.

7.3) results important for the next two companion papers

Four points presented here are worth to be retained for elaborating the derivation construct in companion paper n°2 : (1) the decomposition construct (the derivation construct is to be designed as a specialized form of decomposition) ; (2) the splitting of a regular state transition into multiple constrained transitions ; (2) the concept of mini-token ; (3) the concept of potential.

Two points are important for devising class inheritance in companion paper n°3 : (1) the decomposition construct (this one will act as a pair of "scissors" for

---

28 Inheritance is not supported in higraphs/ statecharts.

29 Let's stress this point again. Telling the opposite supposes a bad use of the extra malleability of color graphs (for example, using blends when not appropriate). In higraphs, bad use is systematically avoided : higraphs are hierarchical (this is so to speak a hard-wired property) : the counterpart is that no malleability exists.

decomposing a class into subclasses) ; (2) the local inheritance of regular transitions along reflex transitions, with notably the concept of p-ancestor-tree. This form of local inheritance (i.e. inheritance inside a same color graph) will be extended first to local concrete implementations (methods and memory representations), then to class inheritance (i.e. considering a hierarchy of color graphs).

Acknowledgements

Subsection 6.2 would not have been possible without the precious help of D[r] Philippe Verdret, a former colleague of us at INRIA Sophia-Antipolis, both a computer scientist and a psychologist. Having read the first version of [Borron, 1995a] [Borron, 1995b] in March 1995, he brought to our attention the existence of [Harel, 1988], the paper devoted to visual formalisms and more precisely to higraphs. Thanks for his suggestion and for all his remarks since then. Thanks also to Gregor Kiczales for his personal encouragements.and for pointing to us [Chambers, 1993] after a presentation of our work to him in July 1994.

Bibliography

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

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

[Borron, 1996a] Henry J. Borron. *"Colored-Object Programming : Goals and Metaphors"*. January 1996.

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

[Borron, 1996c] Henry J. Borron. *"Colored-Object Programming : About the programming activity"*. (First version in January 1996.) Research Report 2880, april 1996.

[Borron, 1996d] 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, 1996e] Henry J. Borron. *"Colored-Object Programming : Rationale"*. February 1996.

[Borron, 1996f] Henry J. Borron. *"Colored-Object Programming : A solution to the state partitioning anomaly"*. In preparation.

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

[Chambers, 1993] Craig Chambers. *"Predicate classes"*. In Proceedings of ECOOP'93. pp. 268-296. Springer-Verlag. July 1993.

[Coleman-Hayes-Bear, 1992] Derek Coleman, Fiona Hayes & Stephen Bear. *"Introducing Objectcharts or How to use statecharts in Object-oriented design"*. IEEE Transactions on Software Engineering, Vol 18, no 1. January 1992.

[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.

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

[Gilmore, 1986] D.J. Gilmore *"Structural visibility and Program comprehension"*. In People and Computers : Designing for Usability (Proceedings of the second Conference of the BCS HCI SG). Harrison and Monk (eds). Cambridge University Press. pp. 527-545. September 1986.

[Green, 1977] T.R.G. Green. *"Conditional program statements and their comprehensibility to professional programmers"*. Journal of Occupational Psychology. 1977.

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

[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.

[Jackson, 1985] M. Jackson. *"Principles of program design"*. Academic Press, London. 1975.

[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.

[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.

[McGregor-Dyer, 1993] John D. McGregor & Douglas M. Dyer. *"A note on inheritance and state machines"*. Software Engineering Notes, Vol 18, no 4. October 1993.

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

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

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

[Soloway-Bonar-Ehrlich, 1988] E. Soloway, J. Bonar & K. Ehrlich. *"Cognitive strategies and looping constructs" : an empirical study"*. Communications of the ACM 26, pp. 853-860. September 1983.

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