Colored-Object Programming: Mixin and Derivation, Two Conjoint Concepts for a Rigorous Handling of Independent Supplementary Behaviours

Henry J. Borron

► To cite this version:

Henry J. Borron. Colored-Object Programming: Mixin and Derivation, Two Conjoint Concepts for a Rigorous Handling of Independent Supplementary Behaviours. [Research Report] RR-2877, INRIA. 1996. inria-00073814

HAL Id: inria-00073814
https://inria.hal.science/inria-00073814

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Colored-Object Programming : Mixin and Derivation, Two Conjoint Concepts for a Rigorous Handling of Independent Supplementary Behaviours

Henry J. Borron

N° 2877
Avril 1996

THÈME 2

Les rapports de recherche de l'INRIA
sont disponibles en format postscript sous
ftp.inria.fr (192.93.2.54)

si vous n'avez pas d'accès ftp
la forme papier peut être commandée par mail :
e-mail : dif.gesdif@inria.fr
(n'oubliez pas de mentionner votre adresse postale).

par courrier :
Centre de Diffusion
INRIA
BP 105 - 78153 Le Chesnay Cedex (FRANCE)

INRIA research reports
are available in postscript format
ftp.inria.fr (192.93.2.54)

if you haven't access by ftp
we recommend ordering them by e-mail :
e-mail : dif.gesdif@inria.fr
(don't forget to mention your postal address).

by mail :
Centre de Diffusion
INRIA
BP 105 - 78153 Le Chesnay Cedex (FRANCE)

Colored-Object Programming : mixin and derivation, two conjoint concepts for a rigorous handling of independent supplementary behaviours.

Henry J. Borron *

Programme 2 — Génie Logiciel et Calcul Symbolique
Action LeTool

Rapport de Recherche N° 2877 — Avril 1996 — 18 pages

Abstract. This paper is about colored object programming. It focuses on the rigorous handling of independent supplementary behaviours. Two constructs are elaborated: the first one, termed "mixin", to capture one such behaviour in a well defined and reusable way ; the second one, termed "derivation", to abstract the assembly of a basic behaviour with a mixin one. For the purpose of reusability, specific attachment constraints are specified at the mixin level . A practical proposal is made that enables the specification of mixins to be simplified, without compromising the efficiency of a derivation. In spite of our specific visual formalism (grounded on connectedness), it should be noted that therigorous proposal made here can be incorporated in state-transition formalisms based on insideness instead of connectedness (i.e. formalisms derived from statecharts).

Keywords. Language design, visual formalism, object oriented programming, state, transition, colored objects, mixin, 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 : mixin et dérivation, deux concepts conjoints pour une appréhension rigoureuse des comportements supplémentaires indépendants.

Résumé. Ce papier a trait à la programmation par objets colorés. Il est centré sur le traitement rigoureux des suppléments de comportement indépendants. Deux constructions sont élaborées : la première, appelée "mixin", pour capturer d'une manière bien définie et réutilisable un tel comportement ; la seconde, nommée "dérivation", pour abstraire l'assemblage d'un comportement de base et d'un comportement mixin. Par souci de réutilisabilité, des contraintes d'attachement sont spécifiées avec chaque mixin. Une proposition pratique est également faite pour simplifier la spécification des mixins sans compromettre l'efficacité d'une dérivation. Bien qu'explicitée dans notre notre formalisme visuel (fondé sur la connexion), la proposition rigoureuse faite ici peut être incorporée aux formalismes visuels par états et transitions fondés sur l'inclusion au lieu de la connexion.

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

Colored-Object Programming : mixin and derivation, two conjoint concepts for a rigorous handling of independent supplementary behaviours.

1. INTRODUCTION

1.1 Preamble

This paper is the second of a sery of three papers about Colored Object Programming (COP, for short)[1]. It elaborates in a same move the concepts of **mixin** and **derivation** constructs. It does so using the essential constructs presented in the first paper [Borron, 1995d]. Likewise, the two new constucts will in turn be taken advantage of in the third companion paper [Borron, 1996e] (which describes class inheritance).

Like the first companion paper, this one is only about the EXTERNAL BEHAVIOUR of class instances. It does not describe how methods and memory representations[2] –which are second citizens in COP– implement a mixin behaviour (this is part of the next companion paper).

1.2 Overview

The derivation and mixin constructs are conjoint concepts, like glue and scissors Their existence enables a distinction between the basic behaviour of a class of objects and the behaviour(s) that supplement it. A **mixin** captures an independent supplementary behaviour in a well-defined and reusable way. A **derivation** abstracts the assembly of a basic behaviour with mixin ones. The third companion paper will take advantage of this to elaborate mixin classes as part of the mechanism of class inheritance.

In the first paper, we termed **color graph** the abstract structure capturing the WHOLE behaviour of the instances of a class in a N-dimension space regardless of its implemen(ation (be it in a flat manner or thanks to a complex class hierarchy). A color graph expresses the possible instance states (using nodes) and the effect of messages on these states (using solid arrows). More precisely, a node represents a possible instance state as such (a **color**[3]) or a contribution to it (a **pigment**) ; a solid arrow, a **regular transition**, i.e. the possible occurring of an external event (message or generic function call) in a certain instance state and its impact on the instance state (next possible states). Messages that are not acceptable in a given state produce an error. To build color graphs, three essential **constructs** were defined : the selection, the decomposition, and the conjunction. The **selection** is meant to express a choice between several possible destination states once a transition has occurred ; the **decomposition** is used to explode a color (1-dimension) into pigments (N-dimension) ; conversely, a **conjunction** recursively groups up to N pigments.

Using these terms, a **derivation** is going to abstract a decomposition with specific constraints ; a **mixin behaviour**, a reusable independent selection.

1.3 Plan

The present paper is organized as follows. First, color graphs are quickly presented together with two simple examples (section 2). Then, the problem of describing and using an independent mixin behaviour is posed and progresssively refined until a final proposal is made (section 3). This proposal is then used for expressing and analysing an example of mixin behaviour and its use in a derivation (section 4). Finally, the related work is described (section 5) and a conclusion is drawn (section 6).

The main section of the paper (section 3) is itself structured as follows. First, hypotheses are made for modularity purposes : the examples of section 2 are used for guiding the intuition. Then, constraints are expressed : first, about the skeleton of a mixin (the structure made by its nodes and constructs) ; and, second, about the regular transitions in a mixin color graph. Practical rules are proposed ; a visual notation is suggested.

In the related work section, a strong connection is made with a work on objectcharts [Gregor-Dyer, 1983].

Because color graphs are essentially equivalent to higraphs/statecharts as a visual formalism (see companion paper n°1), the conclusion stresses an important consequence, well beyond the compared ergonomy of the two formalisms : our proposal about the specification of mixin color graphs and derivations can be imported in higraphs/statecharts.[4]

Important : this paper (as well as its two companion papers) presents **concepts** and NOT a programming environment. A forthcoming paper will show how a set of friendly tools may support the proposed notation

2. COLOR GRAPHS : OVERVIEW

Subsection 2.1 is meant for people that haven't read the first companion paper : it .summarizes the terms and concepts defined in it. The interested reader may well prefer to start with the examples (subsection 2.3) : these are commented in a way that progressively introduces the essential vocabulary ; no backward reference is made to assure independence. These examples are referenced in the rest of the paper.

2.1 Main concepts

[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]

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

[4] On the opposite, color graphs are currently restricted to transformational systems while statecharts cope with reactive systems : we haven't adapted the mechanisms proposed in statecharts, but one may want to do it.

In COP, a class is abstracted as a whole in the form of a **color graph**. Possible (reachable) instance states (or contributions to them) are depicted by nodes ; the existence and effect of external events, by **regular transitions**. Nodes are organized into **essential constructs**. The **skeleton** of a color graph is the structure made by all its constructs (nodes included), i.e. without the regular transitions.

1) skeleton

• Possible instance states are described according to one or several **dimensions**. In the first case, the simplest one, each state is represented by a single node (a **color**) ; the corresponding color graph is termed a **c-graph**. In the second case, each state is basically represented by N nodes, one per dimension (each such node is termed a **pigment**) ; the corresponding color graph is termed a **p-graph**. The set of pigments along one same dimension is termed a **scale** (N scales exist). A p-graph may also exhibit **blends** : each one is a node that recursively groups two or more pigments of different scales. As clearly expressed in companion paper n° 1, blends are strictly optional. We term **p-chroma** a pigment or a blend ; we term **chroma** (or, more loosely, node) a color or a p-chroma.

• Each node in a color graph is given a **condition** (evaluated by sending a side-effect free message to the instance) and/or results from an essential construct. Each node has also an **id**.

• Essential constructs (**selection, decomposition, conjunction**) are one-level trees of nodes, either diverging (the first two constructs) or converging (the last one), their root node being either of type OR (first construct) or AND (last two). The source node of a selection is qualified as being **ephemere**. One or several destination nodes of a selection may be chosen as **INIT** nodes[5]. The destination node of a conjuction is a blend. In any essential construct, a source node and a destination node are linked by a **reflex transition**.

• A reflex transition is **armed** when its source node participates to the current instance state. An armed reflex transition **fires** when the condition attached to its destination node gets true.

2) regular transitions

• **Regular transitions** fire on an external event (a message or a generic function call : in the first case, a single color graph instance is considered ; in the second case, several ones may be involved).

• Regular transitions are **inherited** along reflex transitions (except along those flowing from a decomposition)[6].

• A regular transition is termed **circular** when its destination node is nothing but its source node. A transition is **outcoming** vs. its source node and **incoming** vs. its destination node. A **pure** outcoming (incoming) transition is not circular.

3) mark

The current instance state is marked by a **token** in a 1-dimension color graph, or by a set of N **mini-tokens** in a N-dimension color graph (one per dimension). **Mark** is a generic term for token or mini-token(s). When it fires, a reflex transition moves the mark of its source (selection, conjunction) or part of it (decomposition) to its destination. (A decomposition explodes a token into N mini-tokens.) When it fires, a regular transition moves the mark from its source to its destination.

4) potential

A further (non mandatory) refinement of color graphs consists in attaching a value to each (mini-)token. This value, termed **potential**, can be : (1) initialized at (mini-)token creation ; (2) updated when a transition gets fired (using or discarding the arguments of the message) ; (2) tested for conditions evaluation. This gives an abstract body to regular and testing transitions, and substantiates the color graph functioning (ex. : for animation).

2.2 Visual aspect

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).

Regular graph transitions are represented with named solid thick arrows. **Reflex transitions** are depicted by unnamed thin arrows : using a dotted line for **selections**, a broken line for **conjunctions**. A **decomposition** is not represented using reflex transition arrows, but by a small bar stuck between the root node and the leaf nodes (usually, the root node is only expressed by its name). 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 out-coming of them ; and to draw all their contours in that same coloration.

Usually, one unnamed arrow ondulates from a special node to an initial color (numbered 1 by default) : it corresponds to the default **creation transition** (*make-instance*)[7]. This special node, a small segment with the class name above it, corresponds to the case an instance is not created or has been destroyed.

Circular transitions may be shown without arrows. They may be further qualified (and shown) as consulting or modifying (default) for coherency checks.

Testing transitions (i.e. regular transitions associated to side-effect free messages used for evaluating

[5] This mechanism is used for bypassing tests when entering the selection on all or designated regular transitions originating from the outside of the selection. INIT nodes are parameterized by these transitions. In addition, an INIT condition (ex. : n = 1) may be specified in an INIT node if different from the condition of this destination (ex : n > 0).

[6] A reflex transition may thus have two possible roles : one dynamic (firing, at run time) ; one static (factorization, before execution).

[7] Several creation transitions may exist : in such a case, they should be named (the default one is cancelled).

conditions), because automatically derived by the system, are usually not represented.

An **INIT destination** node of a selection is shown with a double contour.

2.3 Examples

The first example depicts a **c-graph** (a 1-dimension color graph : it is made of colors only) ; the second example, a **p-graph** (two dimensions exist : pigments are used to describe the states). Both examples relate to stacks.

2.3.1 Example 1 : the *Stack* color graph

Next figure depicts how we formalize the behaviour of a *Stack* instance.

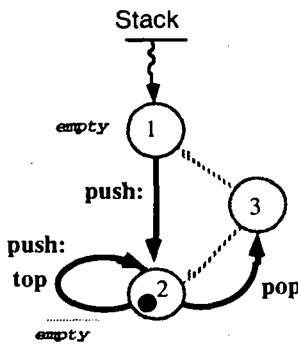

Figure 1.

a) color graph description

A *Stack* instance has two **basic** states : *empty* (**color** 1) and *not empty* (color 2). When just created, it is *empty* and can be sent only a *push:* message. Once this first *push:* has occurred, the instance is *not empty*. Hence, the **regular transition** *push:* transition from color 1 to color 2. Subsequent *push:* messages in color 2 do not change the instance color ; *top* messages do the same : thus, both *push:* and *top* transitions are **circular** in color 2. A *pop* message may also occur in color 2. Afterwards, a *Stack* instance may be either *empty* or *not empty*. To model this uncertainty, we use a **selection** construct : an **ephemere** color (color 3) is created with two **reflex transitions** flowing from it to color 1 and color 2. (Ephemere color 3 represents a cloud of two basic colors.)

Basic colors 1 and 2 are each attached a **condition**, precisely being *empty* or *not empty*. The condition at the selection source (ephemere color 3) is computed by the system : it ORs the conditions of the selection destinations. By convention, this condition is not shown (except on demand).

Due to the existence of conditions, **testing transitions** exist at the selection destinations (where they deliver a constant answer) and at the selection source. By convention, these testing transitions (here, *empty*) are not shown either.

The current instance state is represented by a **token**. Figure 1 shows it in color 2, possibly after the very first *push:* message : when triggered by a message, a regular transition moves the token from its source to its destination. When a *pop* message occurs, the token is thus moved from color 2 to color 3. From there, the token is picked up by the selection which moves it to either color 1 or color 2 : the underlying system is taken advantage of for automatically sending an *empty* message to the *Stack* instance arriving in color 3 and, given the answer to this message, routing the instance to either color 1 or color 2 : the token materializes these moves. (Note that no external request may be sent to an instance when in color 3.)

b) adding a potential

The above functioning of the token is intended to describe what occurs when an implementation (methods and memory representations) has been attached to the color graph. The same effect is obtained —at the specification level— by attaching a **potential** to the token

Here is the specification of the *Stack* potential : **(1)** its initial value is zero (at token creation-time) ; **(2)** a *push:* transition increments it by one (regardless of the argument value) ; **(3)** a *top* transition does not change it ; **(4)** a *pop* transition decrements it by one ; **(5)** the *empty* condition is true only when it is zero. Note that the condition in color 2 corresponds to a strictly positive potential value. (This can be established by inference or set by an additional declaration, possibly attached to the node, possibly attached to the potential itself or -equivalently- to the ephemere node 3.)

2.3.2 Example 2 : a *Bounded-Stack* color graph

A *Bounded-Stack* instance does not behave exactly like a *Stack* instance : an upper *bound* exists on the number of elements that may be *push*ed into. Instead of two basic colors (*empty* and *not empty*), a *Bounded-Stack* instance has three : *empty (and not full)*, *not full and not empty*, *full* (*and not empty*). Next figure depicts one way to formalize the behaviour of a *Bounded-Stack* instance in a space having *emptiness* and *fullness* as dimensions.

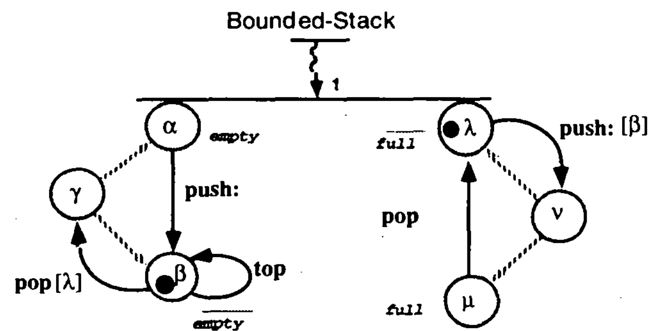

Figure 2.

a) color graph description

According to this color graph, the state of a brand new *Bounded-Stack* instance (color 1) is decomposed into two **pigments**, namely α and λ. (The horizontal bar represents the **decomposition** ; color 1 is only depicted by its name.) Pigments α, β and γ pertain to the *emptiness* **scale** ; pigments λ, μ and ν, to the *fullness*

one. Pigments α, β, λ and μ are **basic** ; pigments γ and ν are **ephemere**. (α λ), (β λ) and (β μ) respectively correspond to states *empty (and not full)*, *not full and not empty*, *full (and not empty)*.

Two regular graph transitions are **constrained** : *pop*, from β to γ ; *push:*, from λ to ν. Corresponding **clauses**, λ and β, are shown in brackets : they specify that the transitions in question are only valid when the object state is (β λ) (both the origin pigment and the pigment mentioned in the clause should belong to the instance state).

The current instance state is marked by two **mini-tokens**, one per scale. The figure shows them in (β λ), possibly after the very first *push:*message to a brand new instance. This first *push:* makes the *emptiness* mini-token flow from α to β (the *fullness* mini-token is stuck in λ since the clause on the *push:* transition in λ is not valid). When in (β λ), a *pop*, a *top* or a *push:* message may be sent to the instance. The *pop* message makes the instance move to (γ λ) and then back to (α λ). The *top* message does not modify the instance state. The *push:* makes the instance move to (β ν) and then to either (β λ) or (β μ) depending on the value of the *bound*. The *emptiness* and *fullness* mini-tokens are moved accordingly. When in (β μ), a *pop* or a *top* message may be sent to the instance, but not a *push:* one (were it, an error will be signalled). The *top* message has no effect on the instance state. The *pop* message activates the *pop* transition flowing from μ to λ (not the one flowing from β to γ since the clause of this one is invalid in (β μ)).

The constraints on transitions express that *empty and not full* may be simplified in *empty* ; *full and not empty*, in *full*. In other words, they captured the fact that *empty* implies *not full* (*empty* => *not full*). Thus states (α λ), (β λ) and (β μ) may respectively be termed without ambiguity *empty, not full and not empty, full.*

Note that the *Bounded-Stack* color graph, as depicted by figure 2, requires at least two successive *push:* messages for an instance to be *full*. This is not a problem in practise ; but, from a modelling point of view, it would be better to include the case "*bound* =1" as well.

b) adding potentials

b.1) a direct reuse of the Stack *c-graph potential*

The above *Bounded-Stack* p-graph may be expanded into a c-graph by combining pigments into colors : (α λ) will correspond to basic color 1 (*empty*) ; (β λ), to basic color 2 (*not full and not empty*) ; (β μ), to basic color 3 ; in addition, two ephemere colors will respectively correspond to conditions *not full* and *not empty*.

Once done, a single token will mark the current instance. The c-graph functioning may be explicited –at the color graph level– by adding a sixth specification item to those already expressing the functioning of a *Stack* token. Here, it is : **(6)** the *full* condition gets true when the potential gets equal to the *bound*.

b.2) a specification of the Bounded-Stack *p-graph potentials*

In a p-graph, a potential is attached to each mini-token. Each potential is meant to be independently computed in its own subgraph (using all the transitions that exist in this one) and to be used for independently computing the conditions that appear in this subgraph. We express this in saying that a potential is modular vs. the transitions and vs. the conditions.

Concerning the p-graph of figure 2, a potential will be attached to each mini-token : *Pe* for the *emptiness* one ; *Pf* for the *fullness* one. Both will be initialized to 0 (at mini-token creation-time), incremented (resp. decremented) by 1 when moved by a *push:* (resp. *pop*) transition, unchanged when moved by a *top* transition. The *empty* and *full* conditions are written (*Pe*= 0) for *empty* ; (*Pf* = *bound*) for *full*.[8]

The transitions to be taken into account for computing a given potential are both the explicit transitions (the *push:*, *pop* and *top* transitions that are effectively drawn in figure 2) and the implicit circular transitions (not shown as a consequence of a convention for a simplified drawing). Next figure shows them all.

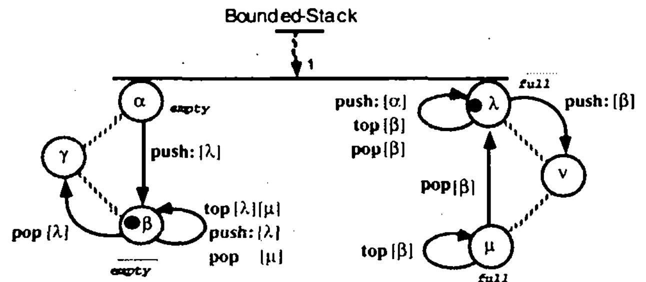

Figure 2.b.

When, for example, the very first *push:* message occurs, it moves the *emptiness* mini-token from α to β ; but il also moves the *fullness* one from λ to λ. Taking into account the implicit transition makes *Pf* to be incremented. Such computations are done automatically.

3. THE DERIVATION AND MIXIN CONSTRUCTS : DESIGN

3.1 Motivation

Besides examplifying the main concepts of color graphs, notably constrained transitions and two essential constructs, the previous example was deliberately chosen to illustrate the problem of mixin behaviours.

A *Bounded-Stack* instance behaves almost like a *Stack* instance. This behavioural proximity should be reflected in their color graphs. However, this is not the case in figure 2 : the *Stack* color graph itself does not show as such (at least, apparently). What we get is :

(1) a left subgraph that has something to do with the *Stack* color graph ;

[8] A single shown transition fires each time. Thus, considering but the shown transitions for computing *Pe* and *Pf*, we may use *Pe+Pf* for testing (this is not modular vs. the conditions).

(2) a right subgraph that has probably something to do with an hypothetical *Bounded* mixin color graph depicting the behaviour common to all (upper) bounded objects ;

(3) constraints on some regular transitions in both subgraphs, these constraints being expressed –on one side– by the root pigment of the hypothetical mixin color graph, and –on the other side– by the pigment corresponding to the state to be refined (*empty* is to be differentiated into *full* and *not full*) .

3.2 Hypotheses

3.2.1 Main hypothesis

We propose to hypothesize an abstract construct (termed the **derivation**) that can attach a **mixin** color graph to one of the nodes (the **hook**) of a **base** color graph. The base color graph should be instantiable ; the mixin color graph is not. The color graph resulting from the attachment of the mixin color graph to the base color graph is termed the **derived color graph**.

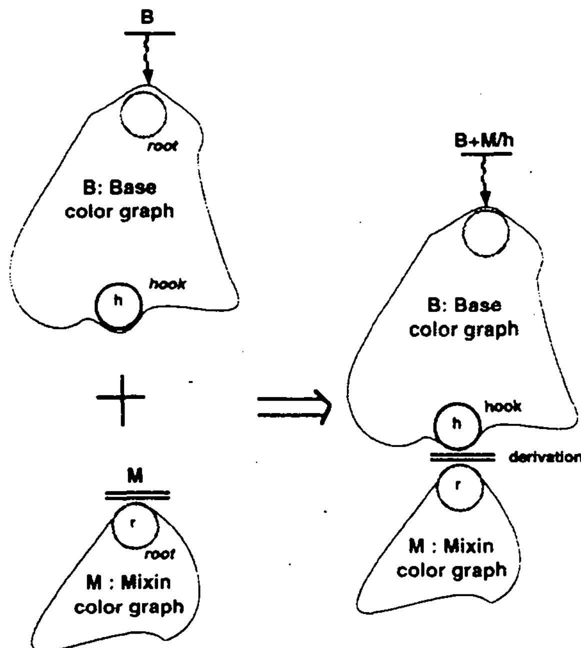

Figure 3.

The construct is supposed to do this using a decomposition and a systematic setting of constraints on regular transitions.

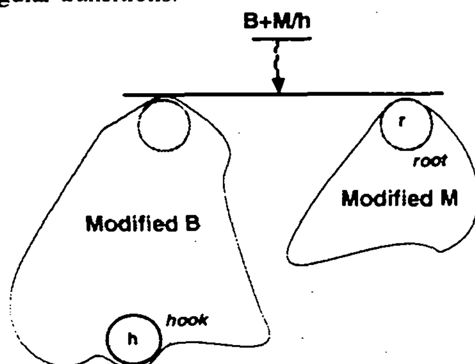

Figure 4.

Modularity is the objective that underlies our main hypothesis :

(α) the behaviour of an instance in the base color graph should not be modified (by the added mixin) in a state different from the hook (if this one is a color) or in the states in which the hook does not participate (if it is a pigment or a blend) ;

(β) the behaviour of an instance during its stay in the mixin should depend (a) on this instance initial state when entering the hook, and (b) on the external events in which this instance gets involved during its stay.

To depict this objective more precisely, let's express how it may model the mini-tokens activity and the computations involving potentials.

3.2.2 Token level hypothesis

1) Proposal

We now focus on the external behaviour of the mini-tokens. We are interested in having a simple one that reflects well modularity.

Let's term **base mark** the mark in the base color graph ; and **mixin mark**, the mark in the mixin color graph. Intuitively, we propose the following :

(1) the mixin mark is not supposed to exist as long as the hook is not marked. This derives from hypothesis α. In our example, this corresponds to having a *Stack*-like instance behaviour till the *Bounded-Stack* instance gets marked *not empty* ;

(2) when the base mark arrives in the hook, the mixin mark gets created in the root node of the mixin color graph. This reflects hypothesis β.a. In our example, this corresponds to the creation of the mixin mark when arriving in the *not empty* state after a *push:* message ;

(3) as long as the mixin mark is moved by messages, the base mark may be thought as remaining in the hook. This is a consequence of hypothesis α. This is very similar to what we observed in figure 2 : the *emptiness* mini-token was stuck in *not empty* while the *fullness* mini-token was moving in the *fullness* dimension ;

(4) when the base mark leaves the hook, the mixin mark vanishes. Again, this derives from hypothesis α. In our example, a *pop* message is supposed to have this effect when the *Bounded-Stack* instance may become *empty* again.

2) Remarks

Even if not sufficiently constraining for specifying the internals of the derivation construct, this external specification is interesting because it forces us to concentrate on some important implications of our model. Several problems may be pointed out :

(a) *Imprecision*

What is the relationship between the base mark and the mixin mark ? If we interpret our current model in terms of dimensions, the base mark refers to a point in the base space. The mixin adds new dimensions to the base space, yet not in every point : in a single one (if the hook is a color) ; in several ones (if the hook is a

pigment or a blend). The hook is like a door on this extended space. The mixin mark is to be thought as different from the base mark since new dimensions may exist in the mixin space. However, do we have to think the mixin mark as being made only of a few additional mini-tokens or as collecting the base mark too ? As a matter of fact, we may consider these two points of view as being complementary : the first one is analytical (it corresponds to a p-graph interpretation, in particular considering the proposed implementation of a derivation by a decomposition : see figure 4) ; the second one is synthetical (it corresponds to a c-graph interpretation).

Note that an alternative may have been proposed for rule 2, namely that the base mark disappears when entering the hook (the mixin mark shows up), and reappears when leaving it (the mixin mark just vanished) : the position of the base mark in the hook may effectively be considered as implicit in such a case ; the mixin mark –at least, at the external level– is thus interpreted as entering the extended space (possibly, with added dimensions), moving in, and then quitting it. However, we feel more comfortable with the initial proposal : because the base mark is left in place, it better corresponds to the idea of extension a mixin is supposed to convey.

Another imprecision may be noted : the above rule 2 does not say what happens to the mixin mark just after its creation : does it stay in the root node or does it move elsewhere ? In the first case, what about arriving in the hook with a different background ? Several initial states may exist. They may be handled by a selection or by mixin transitions relaying the involved base transitions... Both solutions seem possible.

(b) *Ambiguousness*

Rules 3 and 4 may be conflicting. What should occur if a message moves both the mixin and the base marks out of their positions ?
The best is probably to define constraints that a priori rule out such a conflict. Intuitively, a mixin transition is not supposed to move by itself the base mark out of the hook. Consequently, a base transition takes precedence over a mixin transition : if a message is supposed to move both the mixin and the base marks out of their positions, the base mark should be moved ; and the foreseen move of the mixin mark ignored (otherwise, this is like allowing the mixin transition to move back the base mark). This constraint is also valid for entirely new transitions (transitions that do not correspond to those existing in the base color graph).
Conversely, no constraint on mixin transitions should be induced by base transitions that are not valid in the hook (this derives from hypothesis α) : these transitions may only have been triggered in the past history of the instance, thus impacting its initial state when arriving in the mixin ; they may well impact its future, but not its stay in the mixin.

(c) *Degree of efficiency*

Because the mixin mark may vanish (rule 4) and possibly reappear (rule 2) immediately in the same mixin position, the mechanism consisting precisely in quitting the mixin context and retrieving it (or, worse, creating a new identical one) may be costly. Thus, features compatible with our model and increasing its efficiency will be welcome. (A proposal will be made [in §3.4.3 to be precise].) An example to consider in this respect exists for *Bounded-Stack* instances : after the *pop* message has fired, an instance which was *not full* may be still *not full*.

3) Mixin attachment constraints

Next figure formalizes a bit this discussion in adding to the mixin external representation a **virtual hook** with its condition and its set of outcoming transitions, the only ones that do constrain the mixin transitions (cf. point b). As a reminder of point c, a distinction is made between the transitions that systematically move the state out of the hook, those that systematically move it back to the hook, and those that are uncertain. These transitions will respectively be qualified (vs. the hook) as pure outcoming (**OUT**), circular (**CIR**) and impure outcoming (**IMP**).

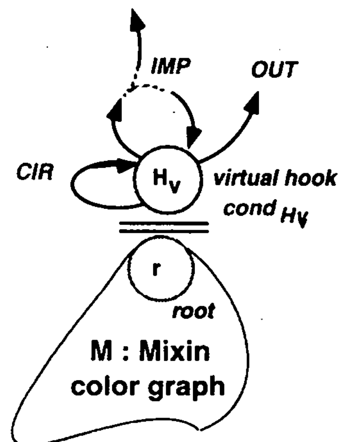

Figure 5.

Because the mixin is supposed to be reusable, the condition and the transitions flowing out of the virtual hook are abstracted from a particular use (names may be changed). They express constraints to be met for an attachment to be possible : a base candidate should observe them. All constraints to be met (including these ones) are part of the **mixin attachment constraints** (MAC, for short).

3.2.3 Potential level hypothesis

1) Goals

Concerning potentials, we want them to be modular both vs. the conditions and vs. the transitions. In other words, we want : (1) each condition in the mixin (resp. base) color graph to depend but on the mixin (resp. base) potential ; (2) the mixin (resp. base) potential to be computed considering but the regular transitions expressed in the mixin (resp. base) color graph.

2) Constraints already observed

For the base color graph, the above constraints mean the added behaviour should change neither the way conditions are computed, nor the way the potential itself is computed. This is coherent with hypothesis α and with the constraint previously expressed about mixin transitions. In particular, entirely new mixin transitions should not modify the base color graph potential.

Considering the mixin color graph, the constraints have different implications since : (1) the mixin potential is to be initialized when created ; (2) all transitions necessary to its computation (even if not changing the instance position inside the color graph mixin) have to be known.
This second point is already met : the external transitions that may impact the mixin potential computations are listed in the MAC. What remains to be done is the initialization of the mixin potential when created[9].

2) Mixin potential initialization

This initialization is to be made using the base potential. As a matter of fact, the information carried on by the base potential should be sufficiently rich to compute the initial value of the mixin potential. Because the mixin is supposed to be reusable, the necessary informations are part of the MAC : they describe what is expected from a base color graph. When the mixin is attached to a base, this base (and not the mixin) is in charge of computing the initial value of the mixin potential as required by the mixin or –more conceptually– the value of the virtual base potential in the virtual hook. (In practice, the initialization function to be run is often the *identity* function.) This computation is hidden to the mixin. From the mixin point of view, all happens as if the base was using the same potential as this mixin does (the virtual base potential in the virtual hook).

Example : suppose a *Stack* potential holds a list of all the messages *push:* and *pop* it has received. Further suppose an attachment of *Bounded* to *Stack*. *Stack* is responsible for computing the number of elements it effectively holds, and to pass this result to *Bounded* so as to conform to the *Bounded* MAC.

3.3 Constraints on the mixin skeleton

3.3.1 Skeleton level hypothesis

The external model we just sketched is unsufficiently constrained. So, let's look at the *Bounded-Stack* example once again (figure 2). The goal is to discover possible constraints on the mixin skeleton.

(1) First of all, a *Bounded-Stack* instance is supposed to have two states, *full* and *not full* which are really substates of *not empty*. (When a given instance is *not empty*, then it is either *full* or *not full*). If a *Bounded* mixin can be defined, it should obey this constraint. To be more general, we say that the states of a mixin color graph form a partition vs. the condition attached to the base hook (be $cond_h$ this condition). In terms of COP constructs, this implies the mixin root node to be the root of a selection, possibly a multi-level one (hence, possibly integrating more primitive mixins in the form of embedded selections...). Note a selection was one of the two hypotheses formulated above as a remark in 3.2.2.2.a.

(2) We also notice the mixin should be systematically entered in *not full*. We thus suspect the existence of a mixin initial node.

(By generalization, the existence of several such nodes may be imagined : as evoked above in remark 3.2.2.2.a, the initial state may well depend on some historical fact, i.e. on the followed path. For the moment, we nevertheless hypothesize but one mixin initial node, since the way many do intervene is not fully clear yet. This point is going to be refined quickly, in 3.3.3.1 to be precise.)

We term **INIT** this basic node and **cond0** the condition attached to it (this condition evaluates to true when the base mark enters the hook)[10].

(3) Another point worth to be noticed is that the condition of the mixin initial state, here *not full*, also evaluates to true when the mark is outside the hook (*not full* is also true when *empty* is true). (See next figure.) We thus may be tempted to generalize that in stating that *not* $cond_h$ implies *cond0*.

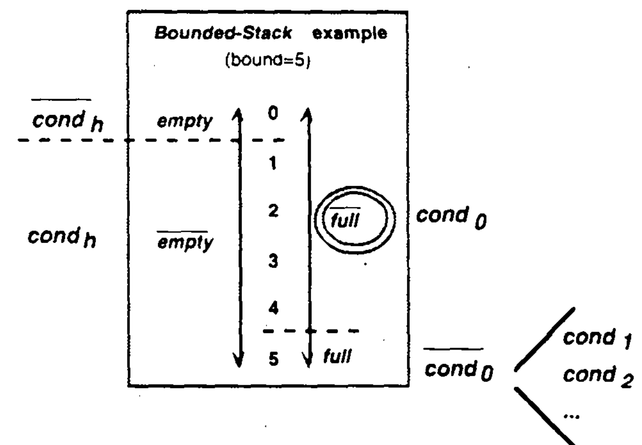

Figure 6.

However, this is a misleading point. What is general is the existence of states in the base color graph that meet the condition *not* $cond_h$ (otherwise, the substates of the mixin color graph are valid in each point of the base color graph : all corresponding derivations thus factorize in a simple decomposition).

For example, if we think about a *Person* color graph, we may well imagine that a *height* dimension exists depending on the *age* of the *Person* instance : a *child* will be supposed to be *small*, an *adult* to be *tall*, etc. Clearly, this dimension wil not be defined in case the *age* category of the *Person* instance is unknown.

3.3.2 Proposal

9 No termination function is required to transfer information from the mixin potential to the base potential since this one is computed separately (see above).

10 Note that an INIT node in a selection construct is a mere facility : strictly not necessary for the working of the selection, it allows the underlying system to bypass one or several tests that would have been necessary otherwise. At the programmer's level, this facility captures some invariant relationship of interest ; yet, it is dangerous : the user's specification of the INIT node is obeyed silently (except in debugging mode), be this specification correct or incorrect.

1) Description

To summarize, we presently refine the model of a mixin to be a (possibly multi-level) selection, valid when the base mark is in the hook, with an INIT basic node (or possibly several ones).

The proposed model thus differs significantly from what we observed in the *Bounded-Stack* p-graph shown in figure 2 : (a) at present, the mixin root is the root of a selection ; (b) the mixin mark does not appear at instance creation-time.

Next figure refines figure 4 : in particular, the mixin skeleton is now depicted as a multi-level selection.

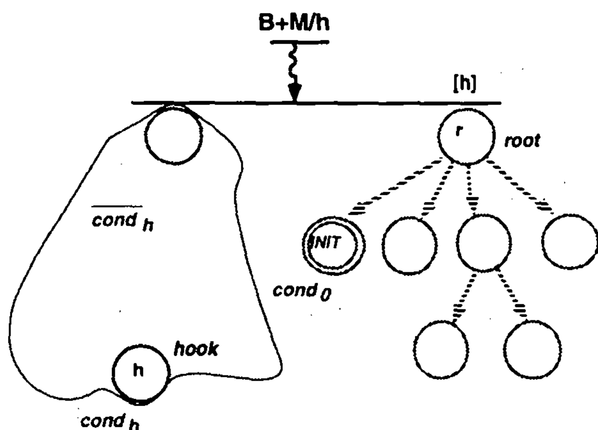

Figure 7.

The validity of the mixin behaviour is restricted : it shows up only when the hook state is entered in the base color graph. Because figure 7 refines the implementation part of figure 4, it also shows how we take into account and implement this important constraint : the mixin behaviour is added (i.e. composed with the base color graph) in the form of a **constrained subtree.**

2) Constrained subtrees

A constrained subtree is a fairly general useful abstraction. The subtree is displayed under the decomposition bar and, above it, a clause (between brackets) is shown.

This clause specifies when the subtree is valid : its contents is a chroma (hook) –or, more generally, a list of chromas (hooks). When the instance state meets this chroma –or, one of them, the mini-tokens that mark the instance in this chroma appear in the root of the constrained subtree.

This constrained subtree is really a displaced subtree (or several identical ones, if a list of several chromas appear in the clause). A constrained subtree is not meant to be drawn twice (in its regular position and as a constrained subtree).

In practice, this notation allows to zoom on subparts of a color graph. Its reciprocal is the possibility to abbreviate a subtree as a single node (holophrasting). Examples will be given soon (figures 8.b-c-d).

3) Example

Let's consider again the *Bounded-Stack* example. The *Bounded* mixin appears as a selection on two basic nodes *full* and *not full* (initial node). When attached, the clause above this selection in the decomposition equivalent is "[*not empty*]" or "[2]", meaning that the two basic nodes in question are substates of *not empty* (color[11] in figure 1).

3.3.3 Degree of generality

Let's examine the problem of attaching a mixin to an ephemere node or to a blend, thus keeping our attention on structural level problems.

1) Mixins in ephemere nodes

Interestingly enough, the proposed model for mixins appears quite modular. This will appear here for several aspects.

• Above, we evoked a *Person* color graph[12] with a *Height* mixin behaviour to be added to *child, teenager* and *adult*, the three destinations of a selection on *age*. Here is its skeleton of this mixin.

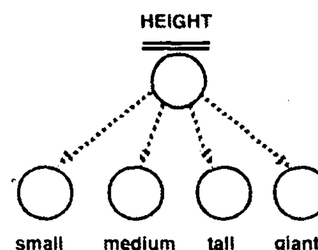

Figure 8.a

Next figure shows the (derived) *Person* color graph.

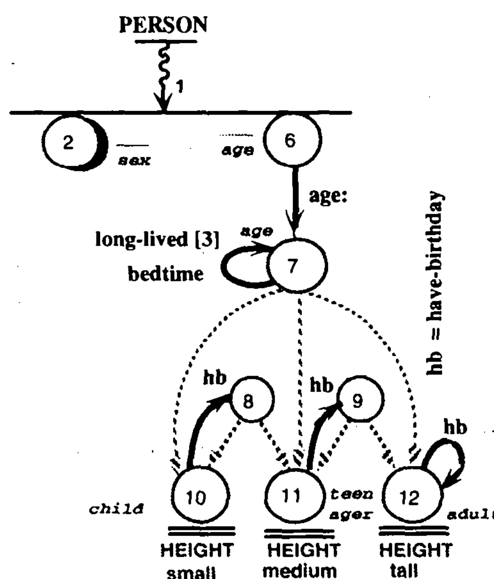

Figure 8.b

In the preceding figure, the three *Height* derivations are shown together with their respective initial values. In

---

11 To be able to consider, as in figure 2, the explicit condition *empty and not full, not full* should be interpreted as the single substate of *empty* in the form of another mixin, a (degenerated) selection constrained by color 1 (i.e. with clause "[1]").

12 The *Person* color graph is shown and explained in companion paper n°1.

addition, node 2 (*not sex*) has been abbreviated : this is evidenced by the shadow underneath.

It is clear that these three *Height* derivations could be **factorized** into a single one, attached to the selection source (ephemere node *age*). It is also clear that a different INIT node is to be provided for each possible case of the selection. This is a simple extension in fact : a selection easily supports such a parameterization. The only thing to do is to provide some syntactic sugar for associating a specific basic node of the mixin to one of the virtual (factorized) hooks.

In figure 8.c, the three *Height* derivations have been factorized into a single one (attached to node 7) and the *Person* color graph has been abbreviated once again (node 7 is now abbreviated too).

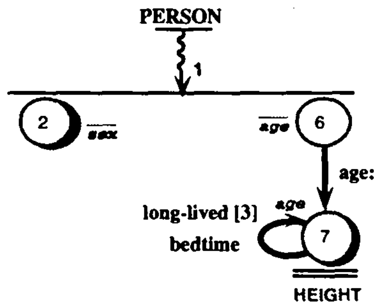

Figure 8.c.

Figure 8.d examplifies a further step. Node 6 itself has been abbreviated ; nevertheless, node 7 may be shown : it is represented in the form of a constrained subtree, used here for abbreviation.

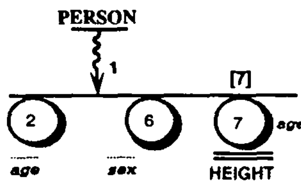

Figure 8.d.

Finally, in figure 8.e, the displayed information is most concise.

Figure 8.e.

(The purpose of the paper is not to describe a possible programming environment. Yet, as shown above, some of the dynamic uses of the proposed notations partly indicate the power it may have.)

2) Mixins in blends

A mixin behaviour may be added in a blend. An example about the *Person* color graph consists precisely in deciding that the *Height* supplement of behaviour depends also on *sex* : a *girl* would be expected to be a little bit less high than a *boy*.

If this behaviour is to be specified for all fully initialized instances, one way to do (and show) it consists in using a constrained subtree[13]. Hence, a notation similar to figure 8.d : in place of "[7]", the clause becomes "[(7 3)]" (3 is the ephemere pigment that ORs the *male* and *female* basic pigments).

3) Parameterized mixins

The clause notation for constrained subtrees offers a way to take into account **parameterized mixins**.

Let's take an example. The *Bounded* mixin depends on the value of a *bound*. We require a general solution :

— a first one is to design a unique color graph that works even when the *bound* is equal to one. However, this solution will be inefficient because of the presence of selections handling the particular case "*bound = 1*" (cf. the very first *push:* in *empty* and the very first *pop* in *full*) : these selections imply run-time testings ;

— a better solution is to specify two mixins with a parameterized clause *[bound=1]* and *[bound>1]*, and to consider both mixins (their ORing to be precise). They will be both considered when the *Bounded* mixin is created. However, only one will become effective because of their disjoint clauses. No run-time testing will be incurred.

4) Convergence of notations

The clause notation for constrained subtrees remarkably expresses the fact that a derivation on each basic pigment of one same scale is equivalent to a simple decomposition (as noted in §3.3.1.3).

For example, if the *Height* dimension was added to pigment 6 (regardless of its semantics), the clause attached to the mixin in the equivalent decomposition would become "[7+6]", or "[age ∨ *not age*]" : this involves all the pigments belonging to the *age* dimension and can thus be automatically replaced by nothing (the double bar and the node above it disappear accordingly). The *Height* subtree gets unconstrained : being attached to the decomposition bar, it is thus part of a simple decomposition (which is correct). This convergence of notation can also be observed at the external representation level using an appropriate convention for the decomposition construct (a simple bar between ordered classes).

3.3.4 Attachment constraint

1) Virtual hook condition

Let's consider the *Bounded-Stack* example once again. Suppose we want to attach the *Bounded* mixin color graph to the *Stack* color graph. We are to do it at color 2 (*not empty*). Doing it in color 1 (*empty*) should not be possible. This is because the two basic states of the mixin (*full* and *not full*) are only meant to be substates of *not empty*.

More generally, a condition may be specified at the virtual hook node of a mixin color graph. As noted in §3.2.2.3, this condition participates to the MAC. Be $\boldsymbol{cond_{Hv}}$ this condition.

[13] A composite transition like *long-lived* may also be expressed that way.

2) Attachment checking

$cond_{Hv}$ should match exactly the actual hook condition, $cond_h$, when the context of use of the mixin is defined ; in other words, it should yield true (resp. false) when the hook condition is true (resp. false). The checking is done at attachment-time, before any execution ("**attachment checking**").

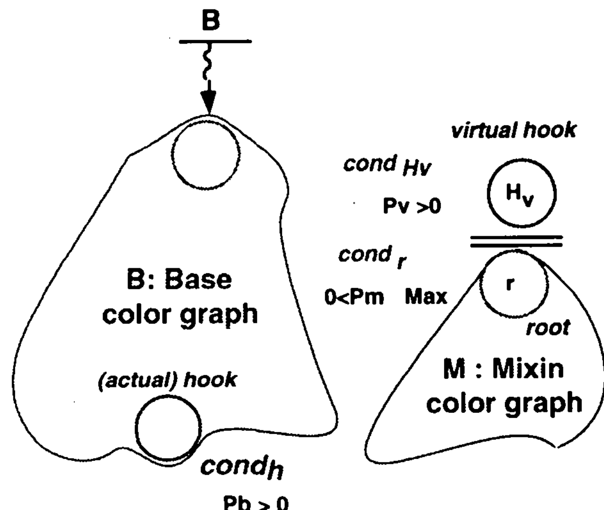

Figure 9.

The condition $cond_{Hv}$ at the virtual hook is expressed using a virtual base potential Pv while $cond_h$ is expressed using the actual one, say Pb. Considering our *Bounded-Stack* example, the following steps are run :

— in the MAC of *Bounded*, $cond_{Hv}$ is specified as being $(Pv > 0)$;

— the *Bounded* mixin color graph is attached to the *Stack* color graph in node 2 (actual hook) ;

— in this node, the condition is $(Pb>0)$: this is the actual hook condition ($cond_h$) ;

— no initialization function is specified for this attachment, thus $Pv = Pb$ (default function = *identity*) ;

— the new expression of $cond_{Hv}$ $(Pb> 0)$ is compared to the expression of the hook condition $(Pb>0)$;

— the attachment constraint is thus satisfied.

Next figure illustrates the relationship between the two actual potentials.

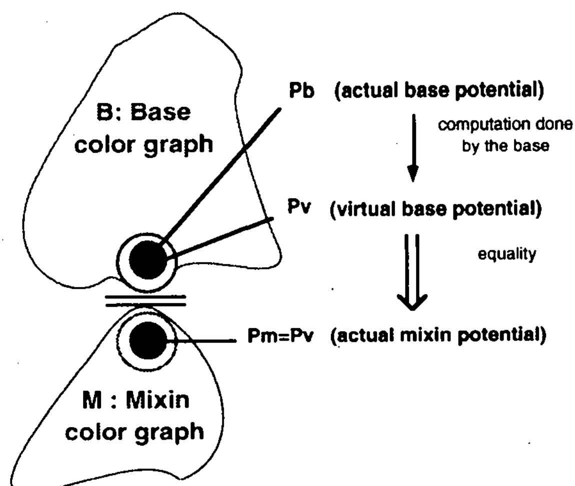

Figure 10.

3) Internal checking

a) Possible existence of a "hole"

A point is worth to be signalled at this level. The responsability of the mixin is entire vs. the specification of the virtual hook condition $(cond_{Hv})$. Since the mixin may refine and thus constrain the behaviour of instances, the condition at the root of the mixin ($cond_r$) may be more restrictive than $cond_{Hv}$. In this case, a "hole" exists.

Note that the computation of $cond_r$ is easy when the conditions at the basic nodes are known : the mixin root, being the root of a (muti-level) selection, simply ORs the partitionned conditions of its basic nodes.

In the *Bounded* mixin case, two basic nodes are defined, one by the condition $(1 \le Pm < Max)$ and the other one by $(Pm=Max)$, Max being the bound. Hence, the root condition is $cond_r = (1 \le Pm \le Max)$. A "hole" thus exists since $cond_r$ is more restrictive than $cond_{Hv}$ $(Pv>0)$. (Remember that $Pm=Pv$.) The "hole" is defined by $(Pm > Max)$.

Pv= 0 ($\overline{\text{cond}_{Hv}}$)	Pv > 0 (cond $_{Hv}$)		MAC
	⇓ Pm = Pv ⇓		
	1 Pm Max (cond $_r$)		Pm>Max
	1 Pm<Max ($\overline{\text{full}}$)	Pm=Max (full)	"hole" ??
Stack	Bounded		color graph
(BASE)	(MIXIN)		

Figure 11.

b) Checking required

If a "hole" exists, demonstration should be given that no instance may fail in that "hole". This supposes the knowledge of the whole mixin color graph (INIT nodes and regular transitions included). The checking is done modularly, supposing only that a base color graph meets the MAC. If the demonstration is not made, it is better to fill in the "hole" with a specific mixin basic node for neat error detection.

In the *Bounded* mixin case, the "hole" condition $(Pm > Max)$ cannot be satisfied by an instance since (1) incrementation of Pm is only due to a *put:* transition ; (2) on entering the mixin, the instance is *not full* (initial state) with initial condition $Pm = 1$; (3) from there, the instance may enter the *full* state after a number of *put:* transitions ; (3) the *put:* transition is disallowed when the instance gets *full* :, thus Pm cannot be incremented beyond Max..

Note that the checking of conditions in a color graph ("**internal checking**") is not specific to the derivation as it is posed in any color graph, be it a mixin or a base (this point was already evoked about the *Stack* example, at the end of §2.3.1.b) .

3.4 Constraints on regular transitions

A color graph is made of a skeleton plus regular transitions. After having uncovered the skeleton of a mixin, we now examine its transitions, and –more specifically– the constraints that they should obey. This part refines the remarks made about the token level hypothesis (subsection 3.2.2).

As previously noticed, a critical point concerns the transitions outcoming from the hook. A distinction is to be made between **pure** and **impure** ones. In the former case, the base mark has no chance to get back to the hook via the automatic firing of reflex transitions ("instantly" so to speak) : one or more external events (messages or generic function calls) are necessary. In the latter case, this is not true. The reason behind that is the "unique destination" principle[14].

This one enables the superposition of **similar** regular transitions (i.e. having a same name, or selector to use the Smalltalk wording) flowing out of a same node to different destinations : all these transitions are replaced by a single one flowing to a common new node (an ephemere node) which allows a selection among the original destinations. This principle can be applied at several levels, hence a possibly multi-level selection. When the superposition involves one or more pure outcoming transitions plus a circular one, the result is an "impure outcoming transition".

Note that an outcoming transition flowing out of the hook to an ephemere node E may be pure vs. the hook : it is impure only if a path of reflex transitions exists from E to the hook.

This distinction being made, we now specify constraints on regular transitions, focusing first on pure outcoming ones, then on circular ones, and finally on impure outcoming ones.

3.4.1 Pure outcoming transitions

Let's term OUT, a pure outcoming transition flowing out of the hook in the base color graph.

Whatever the position of the mark inside the mixin color graph before the triggering of OUT (pre-condition), $cond_h$ systematically evaluates to false once OUT has fired (post-condition). In other words, the clause of the constrained subtree implementing the mixin will be false after the firing of OUT : this subtree systematically vanishes once OUT has fired. The effect of any existing OUT transition (i.e. similar to OUT) inside the mixin color graph is systematically ignored. The rule is thus to forbid any OUT transition in the mixin color graph.

The MAC enables this rule to be checked and thus imposed in absence of any particular attachment : $cond_{H_V}$ abstracts $cond_h$; the OUT property is also specified (cf. §3.2.2.3 and figure 5).

[14] As shown in companion paper n°1, this principle is akin to the "depth idea" in higraphs [Harel et alii, 1987] (p. 55).

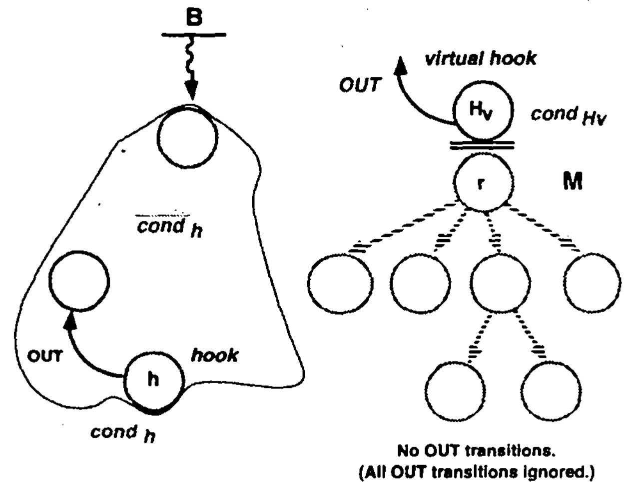

Figure 12.

3.4.2 Circular transitions

Let's name CIR, a (pure) circular transition at the hook in the base color graph. In any node of the mixin color graph, any similar transition is acceptable, be it circular or outcoming, since its pre- and post-conditions both agree with the hook condition. The MAC enables this rule to be checked and thus imposed in absence of any particular attachment : $cond_{H_V}$ abstracts $cond_h$; the CIR property is also specified.

Figure 13.a illustrates the rule. Figure 13.b shows the most general case (all possible transitions) in factorized form at the root node (a "**g-circular**" transition, to use the vocabulary defined in companion paper n° 1).

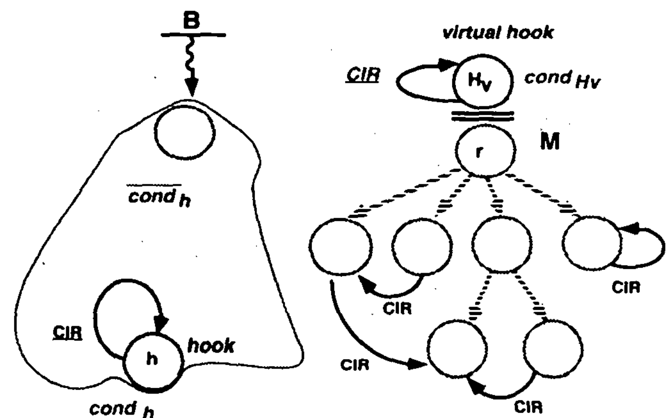

Figure 13.a.

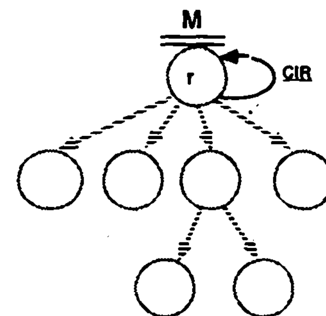

Figure 13.b.

A special but frequent case occurs when the circular transition is side effect free (*testing* transition or transition specified as *consulting*). (The MAC is supposed to record this property too.) Then, whatever the considered substate of the hook, the transition which is attached to it is circular.

Figure 14.a pictures this. Figure 14.b shows the corresponding factorized form (an "**i-circular**" transition, to use the vocabulary defined in companion paper n° 1). [15]

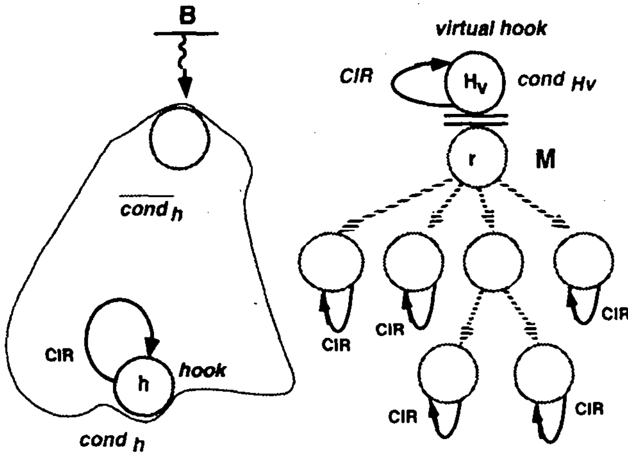

Figure 14.a.

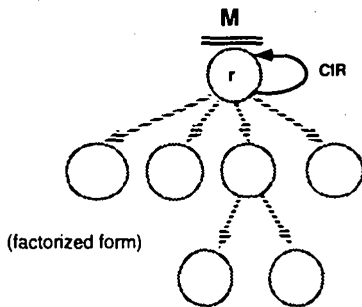

Figure 14.b.

3.4.3 Impure outcoming transitions

The rules obtained in this case are convenient in practice, yet they demand a clear understanding of their grounding. Thus, we introduce them step by step.

a) Analysis

Let's term IMP, an impure outcoming transition flowing out of the hook in the base color graph : this transition flows to a (possibly multi-level) selection involving the hook. As a theoretical step, we propose to consider the extended mixin color graph M' : this one being a selection on M, and a node A with condition *not* $cond_{Hv}$. Since the virtual hook condition of M is $cond_{Hv}$, the root condition of M' is t (true). Next figure represents M' (the virtual hook nodes of M and M' are not shown to simplify the drawing).

[15] Because *testing* and *consulting* circular transitions are quite frequent, the notation shows them without any marking. Conversely, *modifying* transitions (the default) are underlined, and hence the g-circular transitions. This makes the difference between figure 13.b and 14.b.

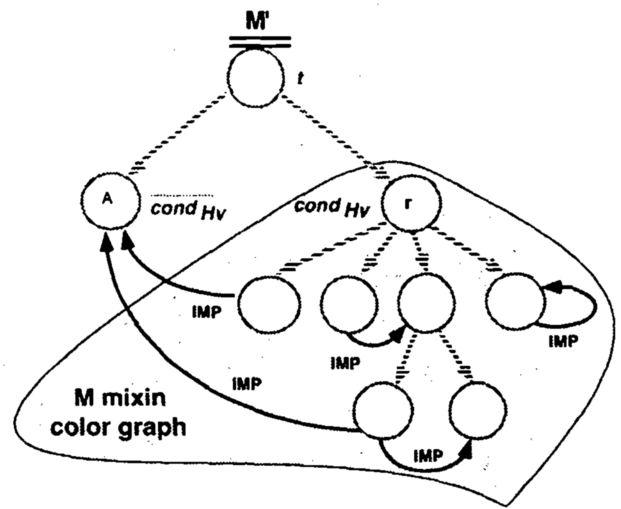

Figure 15.

Any IMP transition that exists in M' is supposed to be correctly implemented vs. the evaluation of $cond_{Hv}$ (same type of "optimistic" hypothesis than for the INIT substate(s)). These IMP transitions thus combine harmoniously with those existing in the base color graph.

Now, we consider the composition of M' with the base color graph. Let's consider a substate x of M. Zero, one or several IMP transitions may flow out of x :

(a) a single IMP transition flows to A : the next state is (K, A) where K belongs to the base graph and is different from H, the hook ;

(b) a single IMP transition flows to a different substate y of the mixin root : the next state is or belongs to (H, y) ;

(c) a single circular IMP transition exists in x : this transition is a priori compatible with both previous results, here (K, A) and (H, x) ;

(d) no transition flows out of x : the result is a priori identical to the one obtained in case of a circular transition ;

(e) one transition flows out of x to A and one to a susbstate y of the mixin root : the result depends on the testing(s) to be done in the base color graph. It may lead to the result obtained either in (a) or in (b) ;

(f) other cases are in fact more or less elaborated combinations of (a), (b), (c). The answer is identical to the one in (e).

b) Intermediate proposal

To avoid the use of the extended graph M', the notation of transitions is itself extended to be reminiscent of an IMP transition to node A (*not* $cond_{HV}$) in M'.

Two additions are made. First, a **nowhere** transition (i.e. a transition that do not flow to any node) replaces an IMP transition flowing to node A (the source node is the same, of course). Second, a **starred** transition (between two mixin nodes x and y) combines a nowhere transition with an another transition in the mixin (from x to y).

The following conventions and rules are based on the previous analysis and allow, when necessary, a more precise specification for cases (c) and (d) :

(1) if a nowhere transition flows out of x, then the next state is supposed to verify *not* $cond_{Hv}$: this result would be assumed if $cond_h$ were to be evaluated (in an ephemere node of the base color graph) ;

(2) if an unstarred transition IMP flows from x to y –be it circular or not, then the next state is supposed to meet $cond_{Hv}$ and it is y (or belongs to the set of substates denoted by y) ; $cond_{Hv}$ is not evaluated (nor $cond_h$) ;

(3) if a starred transition IMP flows from x to y –be it circular or not, then $cond_{Hv}$ will be evaluated : if it evaluates to true, then the next state is y (or belongs to the set of substates denoted by y) ; otherwise, $cond_h$ is assumed to be false (were it to be evaluated).

Note that these rules allow the implementation of the mixin to be done independently. They assume $cond_{Hv}$ to deliver the same result as $cond_h$ (this is checked at attachment time, as shown in §3.3.4). All this is obtained via the MAC ($cond_{Hv}$ abstracts $cond_h$; the IMP property is also specified).

Because an impure outcoming transition at the base hook is expected to meet the $cond_h$ or the *not* $cond_h$ condition (depending on the considered instance), both cases (1) and (2), and/or case (3) are supposed to exist in the mixin color graph for any impure transition at the base hook. This check can be done easily.

This set of rules is referenced as IMP-RULES. It may be used as such. In this case, a nowhere transition is best replaced by an absence of transition. This convention is precisely used in the next figure (derived from figure 15). The three cases are illustrated. Decisions made for substates w and z may have been different.

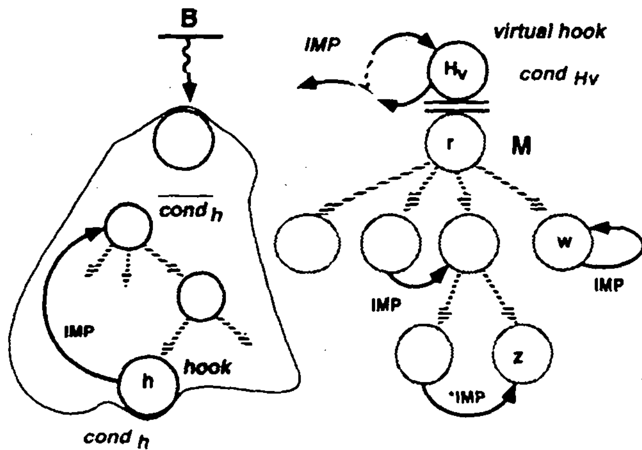

Figure 16.

Note this proposal does not take into account the existence of INIT nodes. Yet it appears already convenient, a more sophisticated set of rules is to be devised if we want to discharge the mixin color graph from such default regular transitions (and others as well), a case often encountered in practice. This attention to details is legitimated by cognitive reasons. In matter of languages, *"language designers need to be extremely careful about even the tiniest details"* [Fitter-Green, 1979], p.252. T.R.G. Green explains : *"The effect of detail is made less surprising when one considers the explosive effects of scaling up a simple algorithm (...) to something ten, a hundred, or even a thousand times bigger"* [Green, 1982], p. 34.

3.4.4 Final Proposal

The proposal is based on the idea of specifying only transitions that do not correspond to usual defaults.

• Default regular transitions at the mixin root avoid unnecessary mixin-wide specifications : they result from remarks done previously (i.e. when studying constraints on the circular and impure outcoming transitions attached to the hook).

• Default regular transitions at mixin nodes basically result from the default transitions at the mixin root (transition inheritance). In addition, when the considered node is itself a basic INIT node, it is considered as a default destination state in case of an impure circular transition that would evaluates $cond_{Hv}$ to true (for the instance implicitely considered).

• A further refinement is possible if INIT nodes are associated with an incoming transition. For example, for the *Bounded* mixin, the initial node *not full* may be specifically associated with *push:*. In a given INIT node, the preceding rule for impure transitions may then be applied only to its associated transitions, i.e. to those known for "filling" in this INIT node.

In the following, "unstarred" has been omitted for simplicity of expression.

a) Default regular transitions at the mixin root

Given a transition at the virtual hook, we term **alter ego** the similar transition attached by default at the mixin root.

(1) a circular transition, specified as *consulting* (or known as *testing*), gives rise to an i-circular alter ego (CIR) ;

(2) a circular transition, unspecified or specified as *modifying*, gives rise to a g-circular alter ego (CIR) ;

(3) an impure outcoming transition gives rise to a starred g-circular alter ego (*IMP).

b) Default regular transitions at mixin nodes

They depend on the transition at the virtual hook :

(I) circular transition, specified as *consulting* (CIR) => any node is supposed to inherit from the i-circular transition CIR at the root node, i.e. to be attached a circular transition CIR ;

(II) circular transition, unspecified or specified as *modifying* => any node is supposed to inherit from the g-circular transition CIR at the root node, i.e. to to be attached an outcoming transition CIR flowing to the mixin root ;

(III) impure outcoming transition (IMP) =>
— if the considered node is an INIT node, then it is supposed to be attached a starred circular transition *IMP (it makes $cond_{Hv}$ to be evaluated ; if it evaluates to true, the next state marks this INIT node) ;

— if the considered node is not an INIT node, then it is supposed to inherit from the starred g-circular transition *IMP at the root node, i.e. to be attached a starred transition *IMP flowing to the mixin root.

c) Specifying non default transitions at mixin nodes

These also depend on the transition at the virtual hook :

(α) circular transition, specified as *consulting* (CIR) => the default circular transition CIR may be redefined by a similar circular transition ;

(β) circular transition, unspecified or specified as *modifying* => the default transition to the mixin root may be redefined by an unstarred similar transition (more specialized) ;

(γ) impure outcoming transition (IMP) => it may be redefined according to the IMP-RULES ;

(δ) pure outcoming transition at the hook (OUT transition) => no similar transition may exist in the mixin color graph (it is ignored as explained above) ;

(ε) not to forget, entirely new transitions may be added.

d) The unwilling transition

A special case of redefinition is offered by the **unwilling** transition (visually represented by striking the transition selector). It is meant to express a transition is inhibited. However, for not breaking our rules, the transition still exists formally (a post-condition should remain the same or be more restricted) : at the implementation level, the method that implements it can be overridden (yet, not transgressing the transition specification). For example, in *Bounded*, the *push:* transition may be specified as *unwilling* when the *bound* is reached.

Because of its underlying intents, an unwilling transition may be supposed to pertub the computations of potentials. In this respect, an unwilling transition has thus a specific abstract body for specifying its effect on the potential. For example, in *Bounded*, the unwilling *push:* transition is specified as having no effect on the mixin potential (void body) whereas the regular *push:* transition incremented it.

This being admitted, a problem exists since the base and mixin potentials will not be computed in accordance. The mixin cannot hypothesize about the base of the potential : from an information point of view, the base potential may be richer than the mixin one. (For example, the *Stack* (base) potential may be a list of all the messages that were sent to the considered instance, while the *Bounded* (mixin) potential may be the mere number of elements in the instance : updating the base potential from the mixin potential when the instance leaves the mixin cannot be made without informations that not part of the mixin.)

For the sake of generality, we propose two possibilities, both based on the idea of updating the base potential each time the unwilling transition is fired. The first proposal consists in adding to the abstract body the necessary effect on the base potential. This way, the base and mixin potentials are in accordance and this is done efficiently. However, this solution is not modular vs. the base mixin. The second solution is more general and can be made efficient : it consists in specifing an uwilling transition as being starred and activating a possible "handling" transition in the base (that takes care of the base potential)..

3.5 Multiple derivations

Because mixins abstract independent supplements of behaviour (on independent dimensions), they may be composed and their composition is simple. In addition, it is commutative. A notation is thus proposed for commodity. (An example is given figure 20.)

4. THE DERIVATION AND MIXIN CONSTRUCTS : AN EXAMPLE

Next figure shows the *Bounded* mixin color graph together with its MAC[16]. It is valid for a *bound* strictly greater than 1.

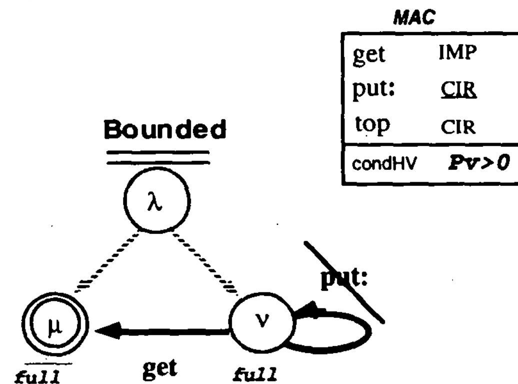

Figure 17.a.

Names have been abstracted as an intent of a large use. The correspondence is to be precised at attachment time.

Substate μ is defined as the *Bounded* mixin initial state. The simplicity of the graph comes from the existence of default transitions.

— Because *top* is specified as CIR (circular consulting) transition, its alter ego is a i-circular *top* transition at λ : this convenes.

— Since the *put:* transition is specified as CIR (circular modifying), its alter ego is a g-circular *put:* transition at λ. In μ and ν, the inherited *put:* transition thus flows to λ : this convenes in μ ; in ν, an unwilling transition is specified.

— Finally, the *get* transition is specified as IMP (impure outcoming) : its alter ego in λ is a starred *get* g-circular transition : because μ is an INIT node, the default is a starred circular transition : this convenes ; in μ, the default transition is the transition inherited from λ, i.e. a starred transition flowing to λ. This one is replaced by an unstarred transition flowing to μ.

16 We haven't shown the INIT condition of node μ nor specified the effects of *put:*, *get* and *top* on the mixin potential (this has been studied in §3.3.4.3).

Next figure takes into account the defaults and shows all the transitions (this is what should be specified if the default rules of the final proposal were not applied.)

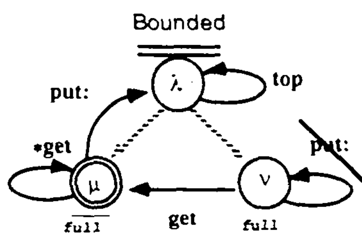

Figure 17.b.

Figure 17.c shows the derived color graph of *Bounded-Stack*. It results from the application of the rules given in subsection 3.4. The circular *push:* transition of *Stack* disappears since it is reimplemented in the added selection. Same thing for the *pop* transition.

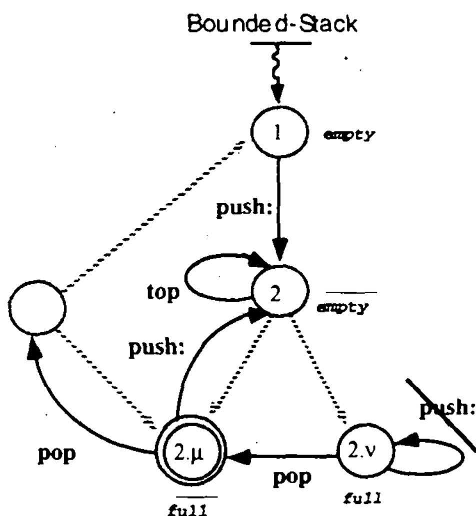

Figure 17.c.

Note this color graph is efficient : no useless tests are done. Note this efficiency can be obtained also while taking into account the case "*bound*=1" via a parameterized mixin (cf. §3.3.3.3).

5. RELATED WORK

Concerning this paper, [McGregor-Dyer, 1983] seems to be the work that, by many aspects, is the closest to ours.

Other works of interest can be related to our approach (either globally or for specific points), notably [Chambers, 1993] [Harel, 1987] [Harel, 1988] [Harel et alii, 1987]. They are discussed in the papers we made when appropriate. (See [Borron, 1995a-b-c], [Borron, 1996a-b-c-d-e-f-g]) .

Our solution to the "state partitioning anomaly" (anomaly described in [Aksit-Bergmans, 1992]) may also be strongly connected to the proposal made here for the derivation and mixin constructs ; yet, the full solution use additional mechanisms. A separate paper will describe it [Borron, 1996h].

Now, we turn to expressing the relation between our work and [McGregor-Dyer, 1983].

5.1 Generalities

The cited reference is about a work *"to better understand the relationship between a state machine representation of a class and similar representations for its subclasses"* (p. 61). It refers to the so-called "strict inheritance model" for inferring three implications (pp. 64-65) :

(1) *"A child class can not delete a state of any of its parent classes"* ;

(2) *"Any new state introduced in a child class is wholly contained in a existing state of one of the parent classes"* ;

(3) *"A child class may not delete a transition from the state machine of one of its parent classes"*.

Our paper does not focus on inheritance, but it prepares it as indicated in the introduction. The decomposition and derivation constructs are going to be used to assemble classes (orthogonal ones, mixins ones) on the basis of their color graphs (cf. companion paper n° 3). As far as the description of (derived) color graphs is concerned, our work –by construction– do observe all the above principles. It is quite striking to note that the authors basically have the same understanding as ours : they propose two *"techniques"* : *"First, a state from a base class could be decomposed into two or more substates in the derived class. The second technique was to add a new set of states that are in parallel to those that existed in the base classes"* (p. 68). This is akin to the idea of derivation and decomposition.

5.2 Bounded-Queue

Next two figures are drawn from page 67 of the cited reference : they exhibit the state diagram for a *Queue* and for a *Bounded-Queue "using a style of state diagram similar to [the] objectcharts"* of [Coleman-Hyes-Bear, 1992]. The goal is to express the inheritance relationship between a *Queue* (figure 18.a) and a *Bounded-Queue* (figure 18.b).

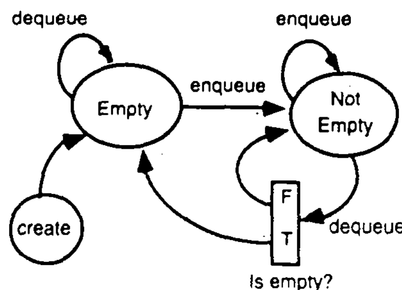

Figure 18.a.

A comparison can be made with the *Stack* color graph (figure 1) and our derived *Bounded-Stack* using the derivation construct (figure 17.c) : because the transposition from *Stack* to *Queue* is obvious, we will speak about the *Queue* of figure 1 and the *Bounded-Queue* of figure 17.c. The transition names given here are those used in the cited reference.

[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.

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