Beyond OOP (1) : Multidimensional Object Behavior Modeling

Henry J. Borron

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Beyond OOP (1) : Multidimensional Object Behavior Modeling

Henry J. Borron

N° 3157
Avril 1997

THÈME 2

INRIA
SOPHIA ANTIPOLIS

Beyond OOP : (1) Multidimensional object behavior modeling

Henry J. Borron *

Programme 2 — Génie Logiciel et Calcul Symbolique
Action LeTool

Rapport de Recherche N°3157 — Avril 1997 — 26 pages

Abstract. The work summarized in this synthesis proposes a multidimensional modeling of objects using a state-transition formalism.

The answer of a multidimensional object to an external event (message or generic function call) depends not only on its class, but also on its state. A clear distinction is made between the specification and the implementation of a class. Concerning the specification, the fundamental concepts are those of transition and (discrete) state : to each class is associated a graph describing the behaviour of its instances in a N-dimension state space. A transition expresses the possibility for an external event to validly occur in a certain state (source state) ; it also expresses the effect of this event on the object state (destination state). The concepts of slot and method only belong to the implementation level : the class graph may be augmented by the specification of memory representations and micro-methods (a pre-method at a source state ; a post-method at a destination state).

In cartesian coordinates, the state of an object corresponds to a point in the space, a point which is identified as such, or using its coordinates along the different axes (dimensions). Yet, for a human being, a N-dimension space is difficult to represent when N is greater than 3. This is why a visual formalism has been designed for displaying any class graph on paper or on a screen. This formalism is connectedness based while other formalisms are insideness based.

The mixin concept, known in traditional OOP, has been extended and refined. Thanks to it, a behavioral supplement may be isolated in a reusable graph, which graph may be attached to one node of a class graph. An attachment constraint may be specified. A key point in the modeling is the type of transitions (CIRcular, pure OUTcoming or IMPure outcoming) at the hook node of the base graph. Efficient parameterized mixins are supported too.

A concept of potential has been elaborated. Related to the notion of invariant, it is defined in a modular way for each dimension. It is valid for classes as well as for mixins (an initialization function is used in this later case). Thanks to this concept, an abstract body may be specified for each method and for each node condition. It paves the way to a more reliable form of programming.

Keywords. Object modeling, abstract model, state, transition, multidimensional space, specification, abstract implementation, concrete implementation, constructs, selection, decomposition, conjunction, reflex transition, local inheritance, class composition, high quality code, language design, visual formalism.

(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

Au-delà de la POO :
(1) modélisation par objets multidimensionnels.

Résumé. Le travail récapitulé dans ce rapport de synthèse propose une modélisation multidimensionnelle des objets à l'aide d'un formalisme d'états-transitions.
La réponse d'un objet multidimensionnel à un événement externe (message ou appel de fonction générique) dépend non seulement de sa classe, mais aussi de son état. Une distinction claire est faite entre la spécification et l'implémentation d'une classe. Concernant la spécification, les concepts fondamentaux sont ceux d'états (discrets) et de transitions : à chaque classe est associée un graphe décrivant le comportement des instances dans un espace d'état à N dimensions. Une transition exprime la possibilité pour qu'un événement externe soit valide dans un certain état (état de départ) ; elle exprime aussi l'effet de cet événement sur l'état dè l'objet (état de destination). Les concepts de méthodes et de champs n'appartiennent qu'au niveau implémentation : le graphe de classe peut être augmenté par la spécification de représentations en mémoire et de micro-méthodes (une pré-méthode attachée à l'état de départ ; une post-méthode attachée à l'état d'arrivée).
En coordonnées cartésiennes, l'état d'un objet correspond à un point de l'espace repéré en tant que tel ou par ses coordonnées selon les différents axes (dimensions). Mais, pour l'esprit humain, un espace à N-dimensions en coordonnées cartésiennes est difficile à se représenter quand N dépasse 3. C'est pourquoi un formalisme visuel a été élaboré qui permet de coucher sur le papier ou sur un écran un graphe de classe quelconque. Ce formalisme est basé sur la connexion alors que d'autre formalismes sont basés sur l'inclusion.
Le concept de mixin, connu en programmation par objets traditionnelle, a été étendu et raffiné. Il permet d'isoler un supplément de comportement dans un graphe réutilisable, lequel peut être attaché au graphe d'une classe en un de ses noeuds. Une contrainte d'attachement peut être spécifiée. D'autres raffinements existent (ex. : mixins paramétrés). Un point clef de la modélisation est le type de transitions (circulaire, sortante pure ou sortante impure) au point d'attache du graphe de base. Des mixins paramétrés efficaces sont aussi supportés.
Un concept de potentiel a été élaboré. Lié à la notion d'invariant, il est défini d'une manière modulaire pour chaque dimension. Il est valide pour les classes aussi bien que pour les mixins (une fonction d'initialisation est utilisée en ce cas). Ce concept permet de spécifier un code abstrait pour chaque méthode et pour la condition attachée à chaque noeud. Il ouvre la voie à une programmation plus fiable.

Mots-clés. Modélisation par objets, modélisation abstraite, état, transition, espace multidimensionnel, spécification, implémentation abstraite, implémentation concrète, constructions, sélection, décomposition, conjonction, transition réflexe, héritage local, composition de classes, code de qualité, conception de langage, formalisme visuel.

Beyond OOP

(1) multidimensional object behavior modeling

H. J. Borron†

1. INTRODUCTION

Started in 1994, the reported work concerns a new form of programming which derives from OOP, but which is about as far from traditional OOP than was OOP from structured programming. It models the behavior of an object using states and transitions in a multidimensional space. This work converges with the objective of making Harel's statecharts the heart of an object-oriented system (cf. [19])[1].

1.1) Motivations

Our work was motivated by at least three categories of converging reasons.

— The first category is structural : traditional OOP does not offer a mean for abstractly modeling the behavior of objects. Influenced by the concept of typestate [24], we early proposed to incorporate this idea in a typed OOP language developed for an ESPRIT project [3]. Soon embarked in a second ESPRIT project, the experience we gained in designing quite complex an application (a multimedia draft editor supporting idea elaboration) confirmed this view : the OOP paradigm is certainly a progress vs. structured programming ; yet a mean for modeling the behavior of objects crucially lacks. And this point is connected with the fact that traditional OOP does not make object behaviors depend on object states (but only on their classes).

— The second category is aesthetic. It concerns control structures. The move from structured programming to OOP had a consequence : the OOP classifying mechanism lessens the number of test control structure instantiations. Making object behavior depend also on states is a way to continue this tendency —possibly to an extreme, i.e. up to the elimination of all remaining instantiations of test control structures ; and, by a systematic use of recursion, of all loop control structures as well. Such a goal is aesthetic : it is not an inevitable constraint, but a mere possibility that appears worth to be experimented for cognitive evaluation.
The recent past help to figure out the involved issues. A number of OOP languages —like C++, Eiffel or the CLU-like typed OOP language we designed in the first ESPRIT project— have imperative control structures. This choice, meant for easing the learning by programmers used to ALGOL-like languages, makes the semantics heavier than if control structures participated to the OOP paradigm too (as in Smalltalk, for example). The corresponding complexity is usually not apparent, being masked by a textual interface ; but a purely visual interface unveils it (two tools are required). In our view, these two drawbacks do not compensate the above-mentioned small advantage of having imperative control structures. In particular, if a visual programming environment is to be built, the effort to be invested in the building of the tools (as well as in their use) should be measured before the final design of the considered language.

— The third category is cognitive. Since 1989-1990, a number of studies —notably due to Guindon (1990), Davies (1991), Burkhardt & Détienne (1995)— have better characterized the design process as being at least partly opportunistic. For example, [16] stresses that *"it has repeatedly been shown that users prefer opportunistic planning rather than any fixed strategy such as top down development"*[2], and shows that the *"three strong corollaries"* that follow this observation (structure perception, multi-level definitness, fluid modification) are not always well supported by traditional OOP especially when considering the inheritance hierarchy (premature commitment, viscosity) and the relationships between methods (limited role expressiveness). A number of papers also reports the difficulty and time spent to locate code in a traditional OO system (Smalltalk-V), leading *"all too often programmers find that the search process is so complex that it is easier just to reinvent a class and method"* ([2], page 4) ; this sometimes leads to *"an overall approach of comprehension avoidance"*, even when the subject *"is modifying its own code that she herself had written earlier"* ([21], page 69). In our view, these observations are coherent with the absence of abstract behavioral modeling in traditional OOP.

---

† Inria Sophia-Antipolis, 2004 route des Lucioles, 06560 Valbonne, France. (borron@chris.inria.fr)

1 Part of the original work summarized here (potential and mixin concepts) and in the companion paper (class inheritance mechanism) is proposed to be a basis for a cooperation aiming at Harel's foreseen system. Following our meeting on november 25 th 1996, this paper and its companion paper were sent to David Harel on december 5th 1996.

2 The ordering of mental steps bears little relationship to the hierarchical structure of the final code. See [23].

1.2) Plan of next sections

Synthesizing the above goals, our research since 1994 takes the basic idea of OOP —give responsibility to data-structures (objects)— and pushes it to its utmost, i.e. it makes the behavior of objects depends on their states as well as their classes. By the way, it eliminates imperative control structures for tests and loops. To abstractly model the whole behavior of a class of objects, a graph is associated to each class : in such a graph, external events (messages or generic functions) are modelled using transitions. Since any graph models the global behavior of a class of objects (i.e. including the inherited behavior), a problem is to relate the various global behaviors defined for a hierarchy of classes, i.e. to re-design class inheritance. Of course, it does not suffice to offer abstract models, it is also necessary to put them into action, i.e. to implement objects.

The rest of the paper progressively introduces each point except inheritance in a class hierarchy (this is the subject of a separate companion paper) : section 2 is devoted to the modeling of objects ; section 3 shows how a hierarchy of classes is built ; a related work section and a conclusion complete the paper.

2. MULTIDIMENSIONAL OBJECT MODELING

The graph associated to each class describes all possible instance states as well as all possible transitions between these states : this constitutes the **specification level**. The graph can be augmented with methods and memory representations : this corresponds to the **implementation level**. Let's briefly introduce both levels.

2.1 Specification Level

Object states and transitions are modelled in a N-dimensions space. A specific formalism is used in this purpose. Its semantics is described in detail in reference [6]. The same report also describes its visual representation while the introductory part of reference [8] presents its textual form. For the sake of brevity, we only give simple examples here : the first one uses but one dimension ; the second and third ones combine several dimensions.

2.1.1 a one dimension example

Let's consider a *Stack* instance. Its **initial state** is *empty*. As any state, this one has a name (say, 1) : *empty* is really the **condition** attached to it. When *empty*, the instance can be sent the *push:* message which makes it become *non empty* : we say that a *push:* **transition** flows from *empty* to *not empty*, the **source** and **destination** states of the transition. When *not empty*, a *Stack* instance can be sent successive *push:* messages without changing its state. We say that the corresponding *push:* transition is **circular**. A *top* message, meant to return the last pushed element, also corresponds to a circular transition. A *pop* message may also be sent to a *not empty* instance, yet two outcomes are possible : the instance may finally be either *empty* or *not empty*. We model this uncertainty by making the *pop* transition flow to the cloud made by the *empty* and *not empty* states, the choice of the final state being determined automatically by testing the instance state vs. the conditions *empty* and *not empty*. Next figure depicts this abstract modeling along the *stack* dimension (axis) : nodes correspond to states and arcs to transitions.

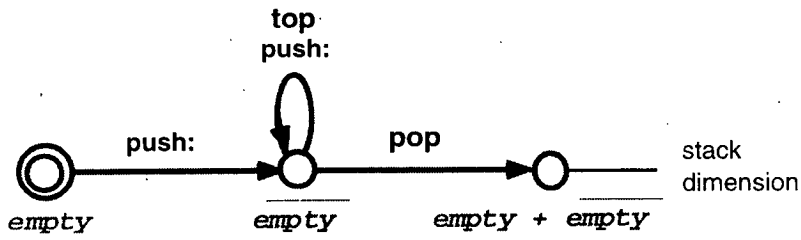
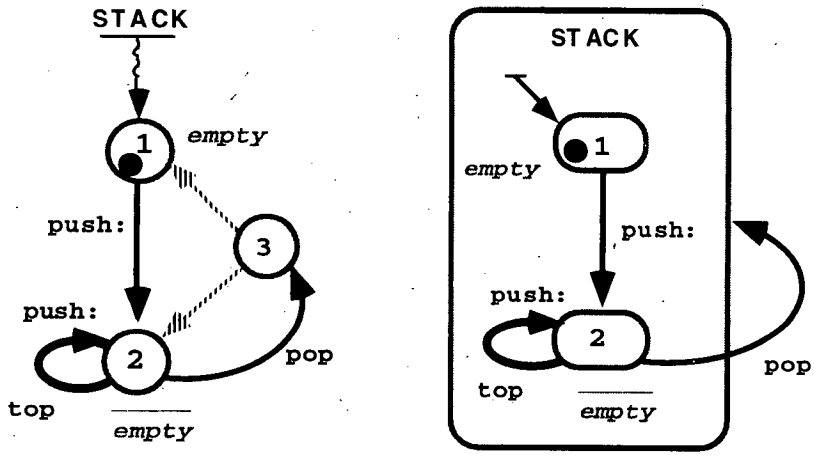


Figure 1.

Let's now propose two visual representations respectively illustrating a connectedness style and an insideness style. The first figure corresponds to our own visual formalism ; the second one, to a variant of the higraph formalism proposed by David Harel [18] notably for system modeling [17]. Note that all *empty* transitions are absent from both drawings : by convention, all transitions the existence of which can be inferred from the conditions need not be represented (these transitions are termed **testing transitions**).

A specific vocabulary has been coined for each style. In higraphs, nodes are termed **blobs** ; and arcs, **edges** (ex. : *top*, *push:*) or **hyperedges** (ex. : *pop*). In our formalism, the two small and grey arrows flowing from node 3 to nodes 1 (*empty*) and 2 (*not empty*) are named **reflex transitions** (to avoid any confusion, the other transitions are termed **regular transitions**). To mark the current instance state, we use a **token** (shown in its initial position in the figures) ; alternatives exist (shadowing the node, greying its surface,...), but we prefer a token moving from state to state as it provides an intuitive support to a concept expressed in a subsequent subsection (see 2.4).



Figures 2 & 3.

Reflex transitions play two important roles : one is static (local inheritance) and will be discussed later on (in subsection 2.3) ; the second is dynamic (firing) and is presented now. A reflex transition has a very simple semantics : it is **armed** if the token is present in its source node ; when armed, it automatically **fires** when the condition of its destination node becomes true ; this makes the token to be moved from the source node to the destination node. (This should be contrasted with the case of a regular transition : this one does not fire on an internal condition but on an external event : the sending of the named message or the call of the named generic function.)

Upon the semantics of the reflex transition is built the semantics of three **constructs.** A construct is a small group (one level tree) of nodes connected by reflex transitions with a simple relation between all the conditions. Reflex transitions are not to exist outside of constructs. Our example illustrates one type of construct, a diverging one termed a **selection** : node 3 is the source of the selection ; nodes 1 and 2 are the destinations of the selection. The condition of the selection source node ORs the conditions of the destination nodes ; these must form a partition of the source node (in terms of conditions). Hence, when the token arrives in node 3, it instantly moves to either node 1 or node 2 after an automatic test of the destination conditions. The token never stays in a selection source node (also termed an **ephemere** node) and no external event may occur when the token is in it.

The same semantics underlies both drawings (a textual syntax has also been defined : see [Borron, 1996c]). This means in particular that the second figure is not a statechart : the single arrow pointing to blob 1 means state 1 is an INITial state, not a default one ; the *pop* arrow has also a different meaning than in statecharts. Our formalism models passive objects and not reactive systems (no concept of time, concurrency, step (like staying one step in the blob encircling blobs 1 and 2), ...).

2.1.2 multiple dimensions example

For each class, we model the behavior of its instances by a system of (discrete) states and transitions. In the previous example, the behavior of a *Stack* instance was modelled according to a single dimension. Yet, quite often, more than one dimension is used. We introduce this point here with a simple example. More complex examples will be worked out latter on : a first one to illustrate **mixin** definition and use (in subsection 3.2) ; a second one to illustrate **masking** and **renaming** (in the companion paper).

Let's consider the *Circle* class. Two dimensions may be considered for modeling the states and behavior of its instances : the *radius* and *center* dimensions. Along the *radius* dimension, two points (modeling substates) exist : one corresponds to *radius-uninitialized* (*not r*); one to *radius-initialized* (*r*). Idem along the *center* dimension (*not c* ; *c*). These two pairs of two points along each orthogonal axis determine four points, the four reachable states of a *Circle*. For example, a fully initialized instance (state 4) is represented by the couple (*r c*). Next figure illustrates that.

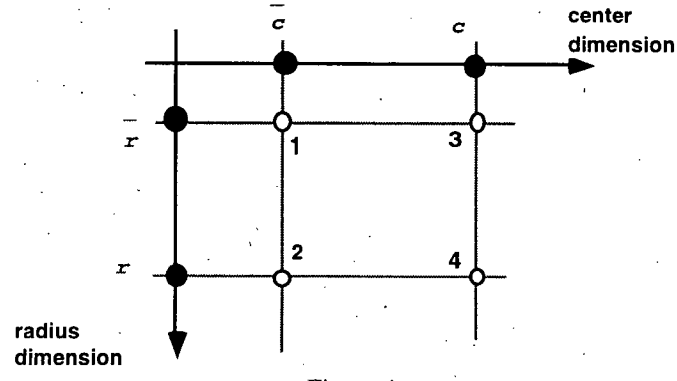


Figure 4.

Let's now consider the (regular) transitions that relate the states of a *Circle* instance. The *radius:* transition, which corresponds to giving a value to the *radius*, is valid in state 1 and in state 3 ; it is independent of the *center* dimension. For this reason, it is shown as a simple transition along the *radius* axis (from *not r* to *r*). Similar remarks apply to other transitions except the *draw* transition which is only valid in state 4. (See next figure.)

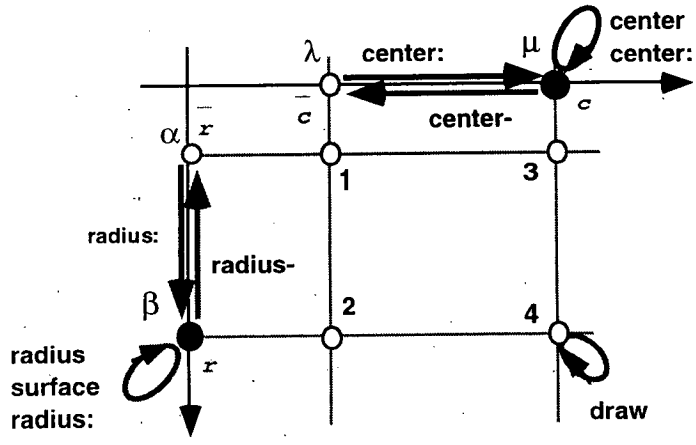
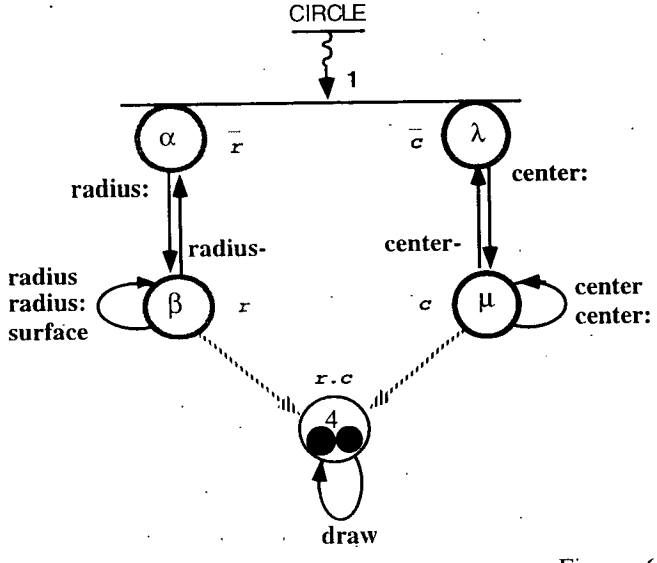


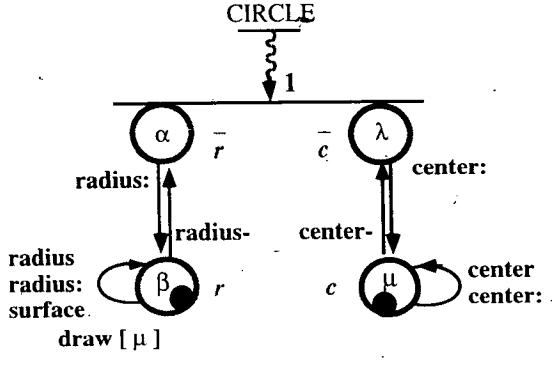
Figure 5.

More generally, the representation we consider as a reference from a theoretical point of view is a cartesian coordinate system in a N-dimensions space (N≥1). In such a space, each dimension is materialized by one axis. Each state is described as a point in this space, i.e. by N coordinate points, its projections onto each axis. Any transition between two states can itself be decomposed into its projections onto the coordinate axes : the result, a list of **constrained transitions**, is termed a **composite transition**. (In our example, the *draw* transition can also be represented by its projections onto the axes : this a priori leads to a couple of circular constrained transitions respectively attached to *r* and *c*, the constraint -termed **clause**- being respectively *c* and *r*.) As a matter of fact, a composite transition can often be simplified, a simple convention enabling the removal of all constrained circular transitions as long as one constrained transition is kept —be it circular or not circular. (*draw* can be simplified that way into a single circular transition.)

The cartesian representation is useful for interpreting the decomposition of states and transitions according to N-dimensions. However, it is rarely practical : human beings having difficulties to vizualize a cartesian space when N exceeds 3! Hence, the necessity of adapted visual representations. Once again the connectedness and insideness styles may be used, as shown by the next three figures. The decomposition into multiple dimensions is shown differently according to the style: a small horizontal bar in the connectedness style (cf. the next two figures) ; a separation line in the middle of a blob in the insideness style (cf. the third figure below). Note that two tokens (one per dimension) are now used for marking the current state.

Like the cartesian space, the connectedness-based formalism enables the easy representation of points as such (ex. : state 4), and —more generally— of sets of points (subspaces) that are parallel to one or several axes. For example, state 4 can be shown as such. Consequently, the *draw* transition can be directly attached to it. When used appropriately, this possibility appears interesting for a cognitive purpose. An alternative representation exists in which state 4 does not appear as such, the *draw* transition being itself expressed in the form of a single constrained transition as explained above. Next two figures show both possibilities (in the second one, the clause appears between brackets).





Figures 6 & 7.

Since insideness is tightly related to hierarchical relationships, the representation of state 4 as such is less natural to obtain in a purely insideness-based formalism : the *draw* transition consequently appears in constrained form.

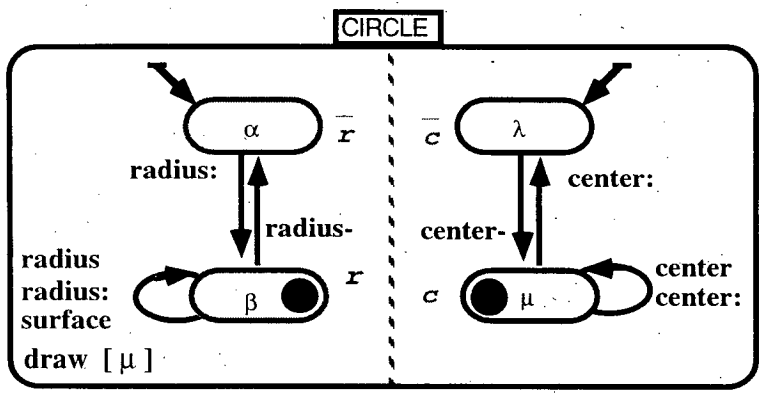


Figure 8.

This example enables us to describe two more constructs, the decomposition and the conjunction. Like the derivation, they are both defined in terms of reflex transitions. Even if represented by a small horizontal bar in our visual formalism, the **decomposition** is really made of a source node with reflex transitions flowing to two or more destination nodes (see next figure) : it is a diverging construct. It is also a AND construct : the condition of its source node ANDs the conditions of its destination nodes. This implies the simultaneous firing of all its reflex transitions. Visually, it is distinguished from the selection (an OR diverging construct) to avoid a confusion ; and, quite usually, its source node is not represented by a bubble, but simply by its name (a number quite often : cf. figures 6 & 7). The **conjunction** is also a AND construct, but it is a converging one : the condition of its destination node ANDs the conditions of its source nodes. Consequently, all the reflex transitions of a conjunction also fire at once. As examplified by figures 6 & 7, the graph attached to a class can always be transformed so that no conjunction appears in it (yet, constrained transitions are likely to appear).

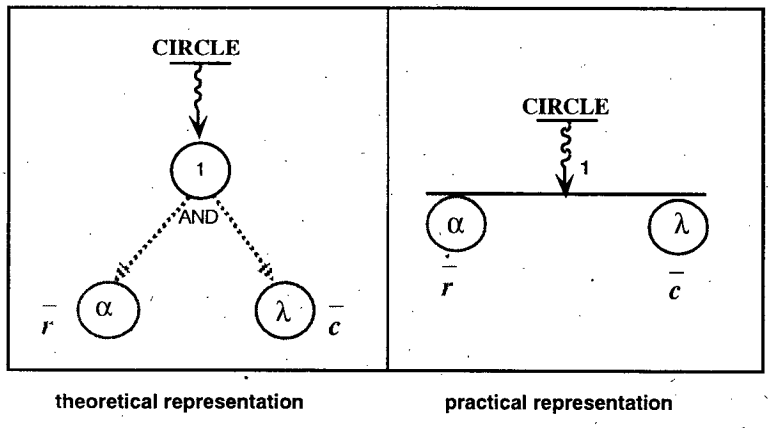


Figure 9.

2.2 Implementation Level

The graph attached to a class represents the complete interface of this class (i.e. including the behavior inherited from its ancestor classes : see section 3). We term **concrete implementation** the attachment of **implementation items** (i.e. methods and memory representations) to a class graph. The result is termed an **augmented** graph.

Two novelties distinguish our approach from OOP at this level : the possible attachment of several memory representations to a class ; the attachment of two methods (termed "micro-methods") to a transition.

2.2.1 Memory representation

A memory **representation** may be attached to each node which corresponds to a single and non abstract dimension. In other words, the node in question should not result from a conjunction (like node 4 in the *Circle* example) and its associated dimension should not be (explicitly or implicitly) declared abstract (like the *object* dimension of class *Object*). A memory representation is made of a number of **cells** (slots in CLOS wording ; instance variables in Smalltalk wording). Changes of representations (between two nodes) and combinations of representations (ex. : to get the representation of node 4 in the *Circle* example) are done automatically. Note a change of representations (**change-rep**) between two nodes is similar to a *change-class* in CLOS.

As an example of different memory representations in a same graph, one may well consider the following choice for a *Stack* instance : (a) no cells at all in the *empty* node ; (b) an *elements* list in the other two nodes. The figure below shows this : a specific pattern is used for each different memory representation ; a small bar represents an automatic *change-rep*.

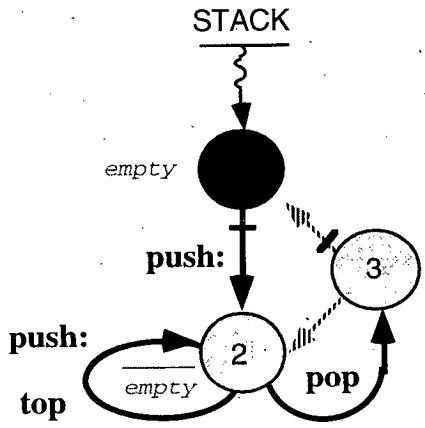


Figure 10.

2.2.2 Micro-methods

Two methods may in principle be attached to one transition : a **pre-method** at the transition source ; a **post-method** at the transition destination[3]. Next figure illustrates this using two small circles : a white one at the source ; a black one at the destination.

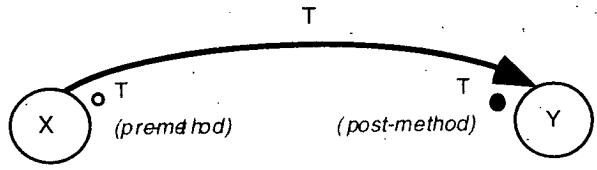
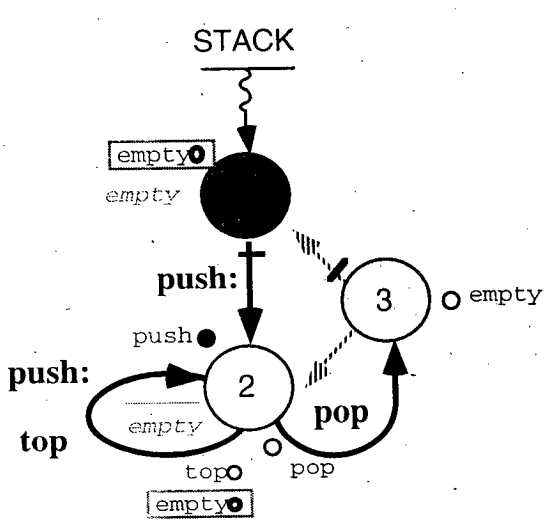
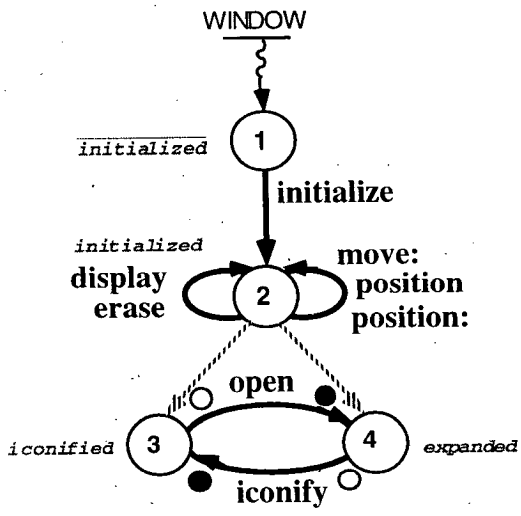


Figure 11.

We use the term **micro-method** (or, more loosely, method) when we do not want to make a distinction between the pre- and post-methods ; or when this is not necessary, the context being unambiguous.

Frequently enough, a pre-method alone convenes : this matches the OOP case. Sometimes, a post-method alone is a better choice ; other times, a couple of pre- and post-methods offers a better modeling. The next two figures illustrate all possibilities.





Figures 12 & 13.

[3] Qualified methods (cf. *:before*, *:after*, *:around*, ... methods in CLOS) constitute a different, orthogonal issue.

The first two cases are examplified with the *Stack* class again (first figure). Using the *Stack* representations described above, a pre-method alone is used for *pop*, *top* and *empty* (the two *empty* pre-methods inside grey bordered rectangles are constant methods : they are generated automatically). A lonesome *push:* post-method is used to implement the *push:* transition in the *empty* state (a pre-method appears absolutely useless since a cell is needed to hold the pushed element, but this cell does not exist in the *empty* node) ; this *push:* post-method also implements the *push:* transition in the *not empty* state.

The second figure illustrates the third case (pre-method + post-method). Initialized instances of a *Window* class are supposed to be either *iconified* or *opened*[4]. The *open* transition from the *iconified* node to the *opened* node is obtained with a pre-method for erasing the icon and a post-method for displaying the window itself (the *iconify* transition does the opposite using also a couple of pre- and post-methods). Such a neat modeling cannot be rivalled by traditional OOP.

2.3 Local inheritance

Inheritance along the reflex transitions of a SINGLE graph is termed **local inheritance**. It concerns transitions (specification level) as well as implementation items (implementation level). This property enables thus **factorization** inside one graph.

2.3.1 Local inheritance for transitions (specification level)

Rule : regular transitions are **inherited** along all reflex transitions (except along those flowing from a decomposition)[5]
The next figure illustrates the basic principle : the *T* transition flowing from node *s* to node *d* is inherited in node *a* ; in other words, a transition **similar** to *T* (same name) conceptually flows from *a* ... to *d*, the destination of *T*. This is the **inherited transition**. Were node *a* the source of another reflex transition, the principle would be applied recursively. *T* in *s* is termed the **factorized transition**.

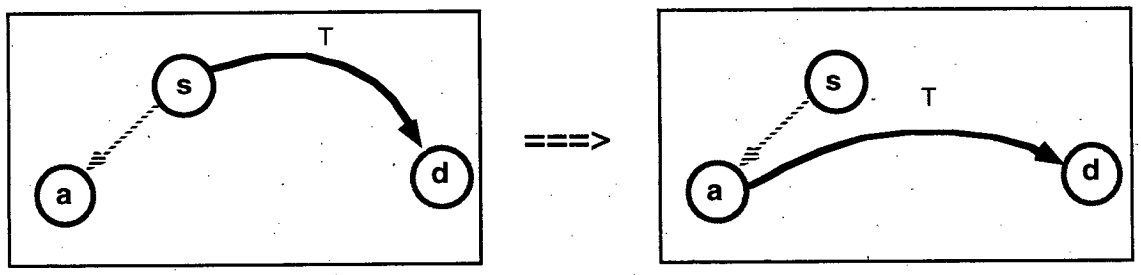


Figure 14.

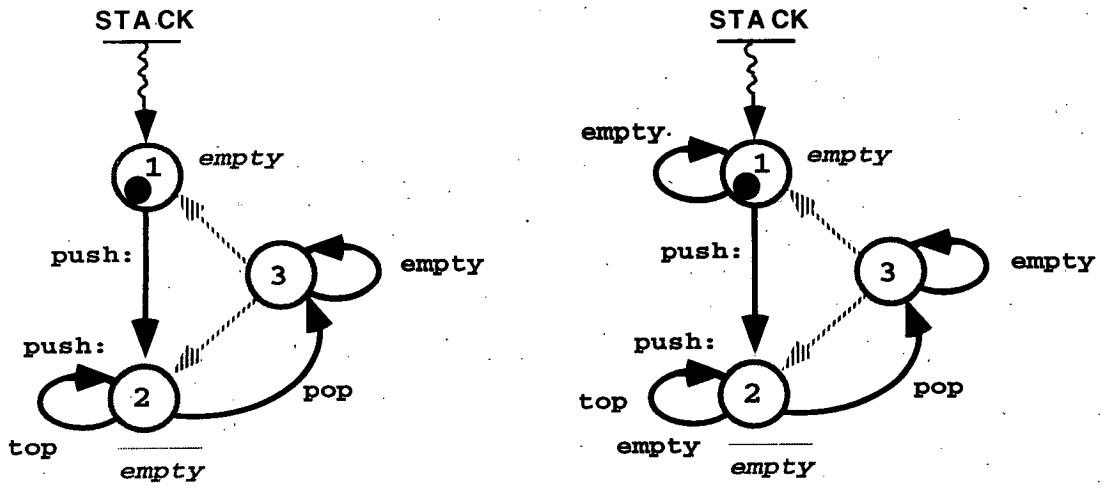
A quite frequent case corresponds to a transition flowing out of an ephemere node and leaving the state of the instance unchanged : under these circumstances, the factorized transition is circular and the inherited transitions are also circular. Let's give two examples of that. Were the default rule of subsection 2.1.1 about the representation of testing transitions not obeyed, the *empty* transition of the *Stack* graph would be shown in node 3 as done in the next figure : being inherited along the reflex transition (3 1) and (3 2), the *empty* transitions in nodes 1 and 2 need not be specified. This **factorization** of the *empty* transition in the ephemere node 3 simplifies the *Stack* graph (figure 16 is shown for comparison). Note that the factorized transition is itself **valid** in node 3 : the instance automatically sends itself the *empty* message when its token arrives in node 3 so as to choose between node 1 and node 2. Similar remarks could be made about the *Window* graph of figure 13 for the *iconified* and *expanded* testing transitions ; concerning all the regular transitions attached to the ephemere node 2, the situation is a bit different : these are factorized transitions and as such they are inherited in the *iconified* and *expanded* nodes, yet these factorized transitions are not valid in node 2 : for example, the *display* circular transition is inherited in states 3 and 4 (the inherited transitions are also circular). Yet, *display* cannot be sent to the instance while in the ephemere node 2.

Another case is due to the reflex transitions of a conjunction : all regular transitions attached to its source nodes are inherited in its destination node (for example, in the *Circle* graph of figure 6, the *surface* transition, valid in substate β is also valid in state 4.)

---

4 This example is drawn from [12], subsection 3.5, page 279.

5 Are thus concerned all the reflex transitions represented by grey arrows in our visual formalism.



Figures 15 & 16.

Note : in all the examples presented here, all the factorized transitions happen to let the instance state unchanged : this is not the usual case. This type of circular transitions is termed "**i-circular**" and are distinguished from "**g-circular**" transitions which do not result in circular inherited transitions : the next figure depicts the inheritance of a g-circular transition in an ephemere node (note that this corresponds in fact to n^2 inherited transitions if n is the number of destination nodes of the selection : see figure 18). The reader interested in cognitive ergonomy will find in [9] five design principles for building up a graph : two of them (**one source principle, one destination principle**) are related to factorization and apply to frequent topologies.

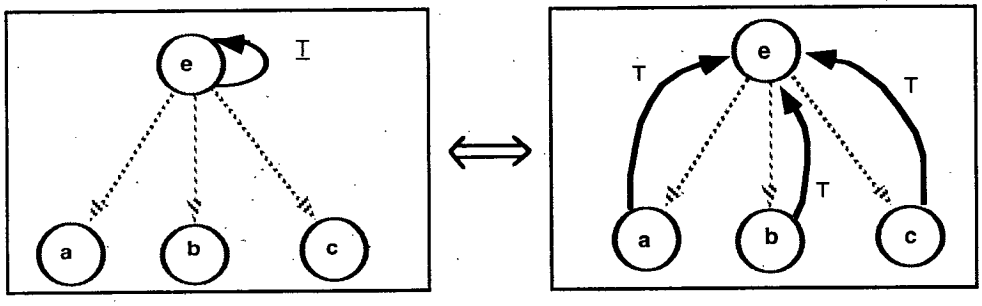


Figure 17.

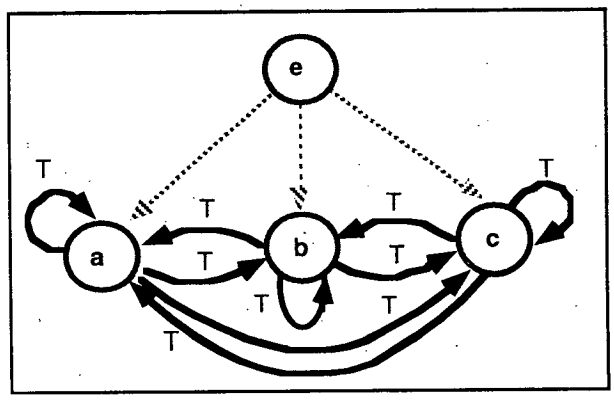


Figure 18.

As shown in figure 14, the destination of an inherited transition is the same than the destination of the factorized transition. Generally speaking, the **effective destination** is to be determined at run-time among the ultimate **descendants** of this destination node, i.e. among the leaves of the tree of reflex transitions recursively flowing from this node. Of course, if this tree is reduced to that single node, then the destination of T is that node itself and no run-time testing is necessary. Except this trivial case, the destination of T may also be determined statically when T is an i-circular transition : the inherited transition is also i-circular, thus its destination is its source. Hence, a two steps algorithm for moving the token when the graph is unidimensional. **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.[6]

[6] This algorithm presupposes the graph satisfies the "unique destination" principle.

In case of a multidimensional graph, local inheritance is dealt with in the following way :
— for each involved dimension, activate the inheritance scheme used in case of a unidimensional graph ;
— then combine the obtained results, i.e. the destinations of the activated transitions.
The determination of the effective destination state upon the reception of a message is thus a bit more complex than for a unidimensional graph : several tokens may be moved and a **backtracking step** takes place before the propagation step.

Next figure illustrates this case with the *Person* graph. This graph abstracts the source code of the example given in [12] by C. Chambers, subsection 4.1 page 286. In particular, *boy* and *girl*, the destinations of the two conjunctions, map exactly the definition of the *boy* and *girl* predicate classes. Similar remarks apply to *male*, *female*, *child*, *teenager*.[7] Except for *long-lived*, which tests if a *Person* instance is older than its *expected-lifespan*, all names are self explicit. Let's consider an instance and let's suppose its current state is *boy*. A *have-birthday* message triggers the transition flowing from node 13 to node 8 (which is inherited from the one flowing from 10 to 8). Hence, the three steps : (1) move step : the *age* token is moved from node 13 to node 8 ; (2) backtracking step : because the *sex* token cannot stays alone in *boy*, it is moved back to node 4 (*male*) ; (3) propagation step : the *age* token is moved, for example, to node 11 (*teenager*). In this case the new state is thus (*male*, *teenager*). Were the *age* token moved to node 10 (*child*), the conjunction will fire (both tokens move to node 13) and the new state will thus be *boy* again.

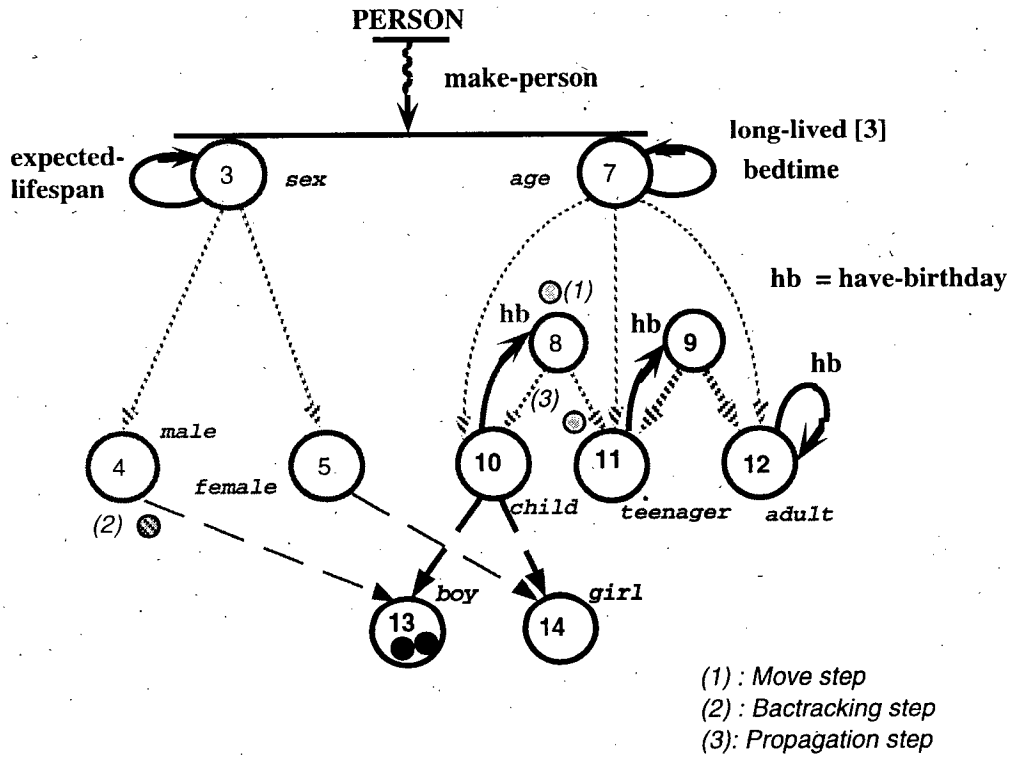


Figure 19.

For more details, notably about the handling of constrained transitions, consult [6] and [8].

2.3.2 Local inheritance for cells and methods (implementation level)

Knowing that transitions are inherited, it is very tempting to devise rules for inheriting implementation **items** (memory representations, pre- or post-methods) along the same reflex transitions.

The solution we propose is directly modelled from the handling of transitions :
— as for transitions, it consists in inspecting each dimension in turn and selecting each time one and only one valid item if it exists, the most specialized one if several exist (masking effect). (In fact, all dimensions need not be inspected systematically : when searching a pre- or post-method, are only **involved** the dimensions for which exists the transition in question ; when searching a memory representation, each non abstract dimension is involved). Each involved dimension should be satisfied once and only once. Idem for any other dimension that is **employed** (by the *m* methods) but is not involved (by the *m* transition)[8].
— as for transitions, items are searched along the reflex transitions when not found locally (local inheritance) ;

[7] The *adult* case is handled in the original source code as a default case for a *Person* instance.
[8] For example, the *print* transition is defined for the sole *object* dimension, but the *print* methods generally use all the dimensions.

— as for transition destinations, a combination is to be done when the selection by dimensions has yielded several items (multidimensional graph). (Note that the combination of dimensions has nothing to do with the CLOS-style combination since the concept of dimensions does not exist in CLOS.)

As an example, let's consider the *Stack* graph of figure 12. Suppose an *empty* message is issued to an instance in state 1 (or 2). The *empty* transition is first retrieved : it is inherited from node 3 (see figure 15). The factorized transition being i-circular, the inherited transition is also circular. Now begins the actual search for the pre- and/or post-methods. Were the constant methods (inside grey bordered rectangles) not existing, then no method for *empty* would be found in node 1 (or 2). The search would thus be done along the reflex transition flowing to this node : the *empty* pre-method would be selected in node 3 ; no post-method will be found. The *empty* pre-method attached to node 3 would thus be activated, yielding a true (resp. a false) answer.

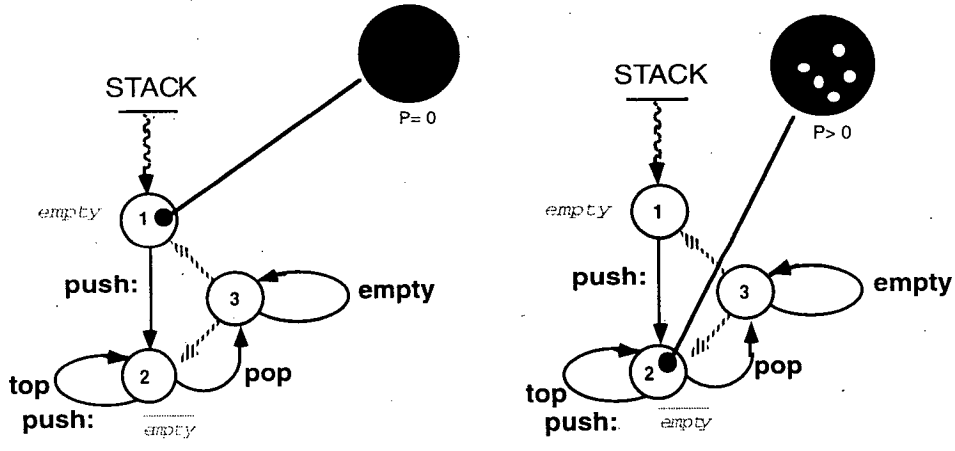
In the *Person* example, it is clear that the *have-birthday* pre-method will be factorized in node 7 (no post-method). The reader may check that, whatever the instance state (ex. : (*female*, *teenager*) or *boy*), this pre-method will be selected when a *have-birthday* message is issued.

Due to the lack of space, we also omit here a great number of details. The interested reader will find them in [8].

2.4 The concept of potential

So far, the definition of a transition is similar to the definition of a method header : no body exists. The concept of potential remedies this : it considers each token as a recipient in which a value, perhaps a complicated one, is stored. This value can be : (1) initialized at token creation ; (2) updated when a transition gets fired (using or discarding the arguments) ; (2) tested for the evaluation of conditions. This (non mandatory) refinement thus provides an abstract body to regular transitions. Hence, the possibility to substantiate the graph functioning (ex. : for animation) and to check an actual implementation vs. this abstract specification.

As an example, 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 node 2 corresponds to a strictly positive potential value. This can be established by inference.



Figures 20 & 21.

3. BUILDING UP THE CLASS HIERARCHY

The preceding section has developed the specification of a class as well as its concrete implementation. Now we come to the **abstract implementation** of a class, i.e. to its composition using pre-existing classes. These classes are termed its **superclasses**. Repeating this process leads to the building of a class hierarchy. Two operators are used for this purpose, the composition and the derivation.

3.1 Composition

Let's consider again the *Person* example. The figure below shows the graph in canonical form, hence a slight difference with figure 19 : the *make-person* **creation transition** has been replaced by its equivalent, i.e. the *make-instance* standard creation transition plus two regular transitions for initialization (*sex:* and *age:*). The figure afterwards abstracts the *Person* graph : obviously enough, two independent parts can be isolated for modularity purpose, leading to the creation of the *Sex* and *Age* classes ; the remaining part is the only one really specific to *Person*.

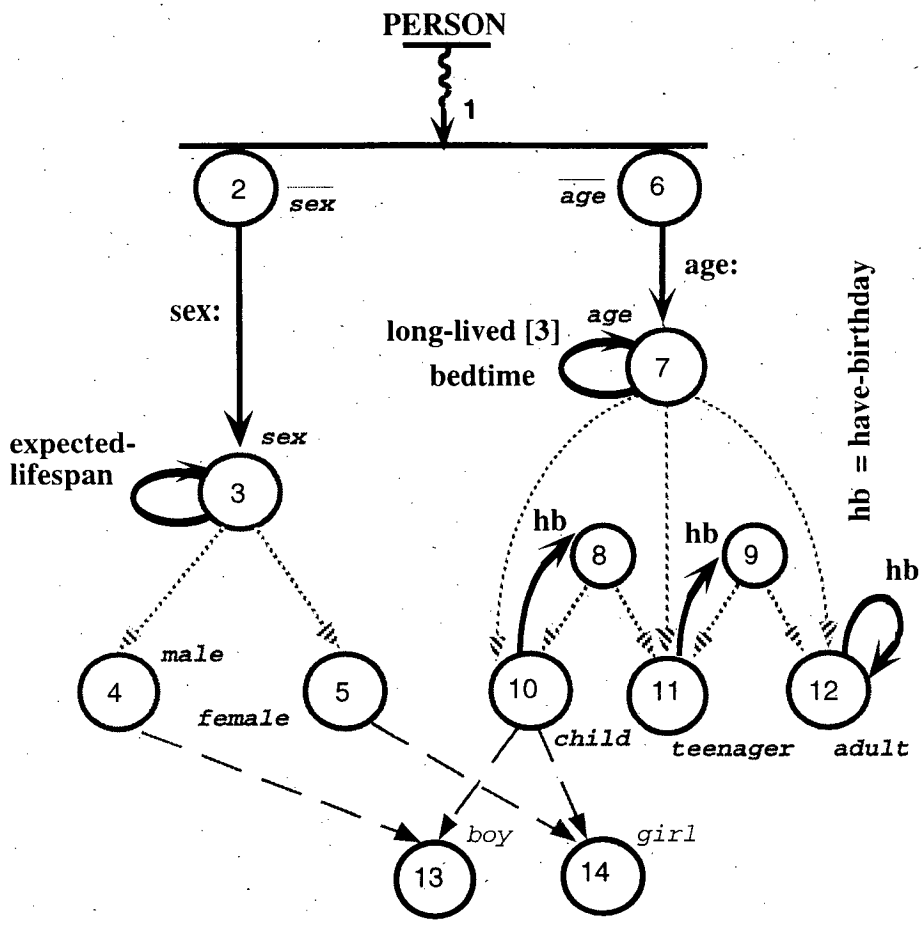


Figure 22.

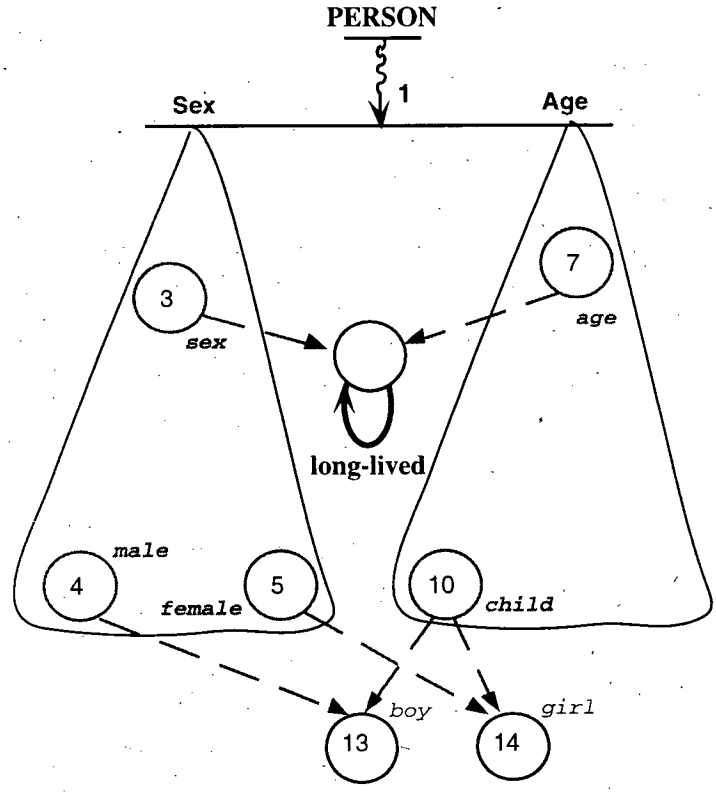


Figure 23.

The next two figures express the same idea using a cartesian representation : besides the *sex* and *age* axis, an extra axis is used for conveniently representing selections.

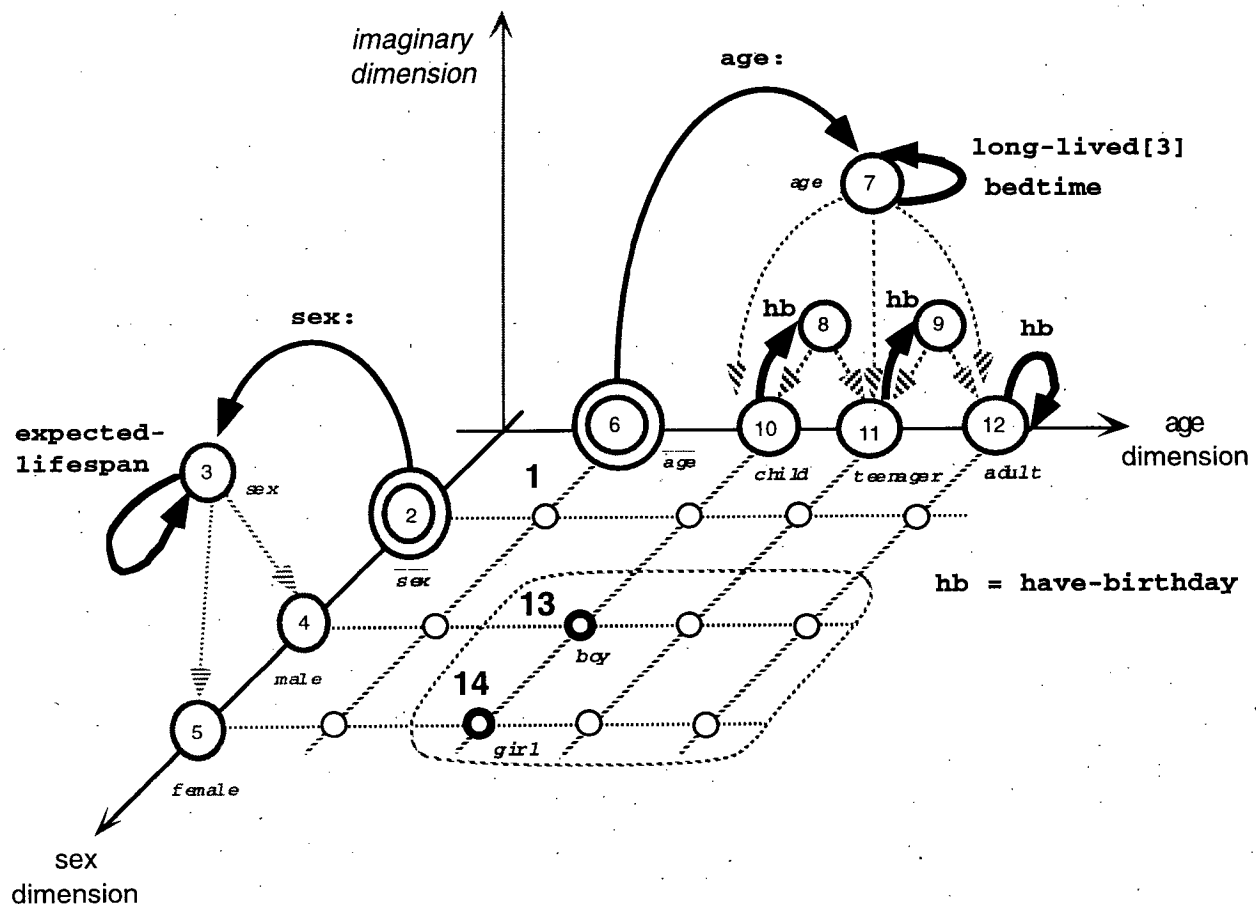


Figure 24.

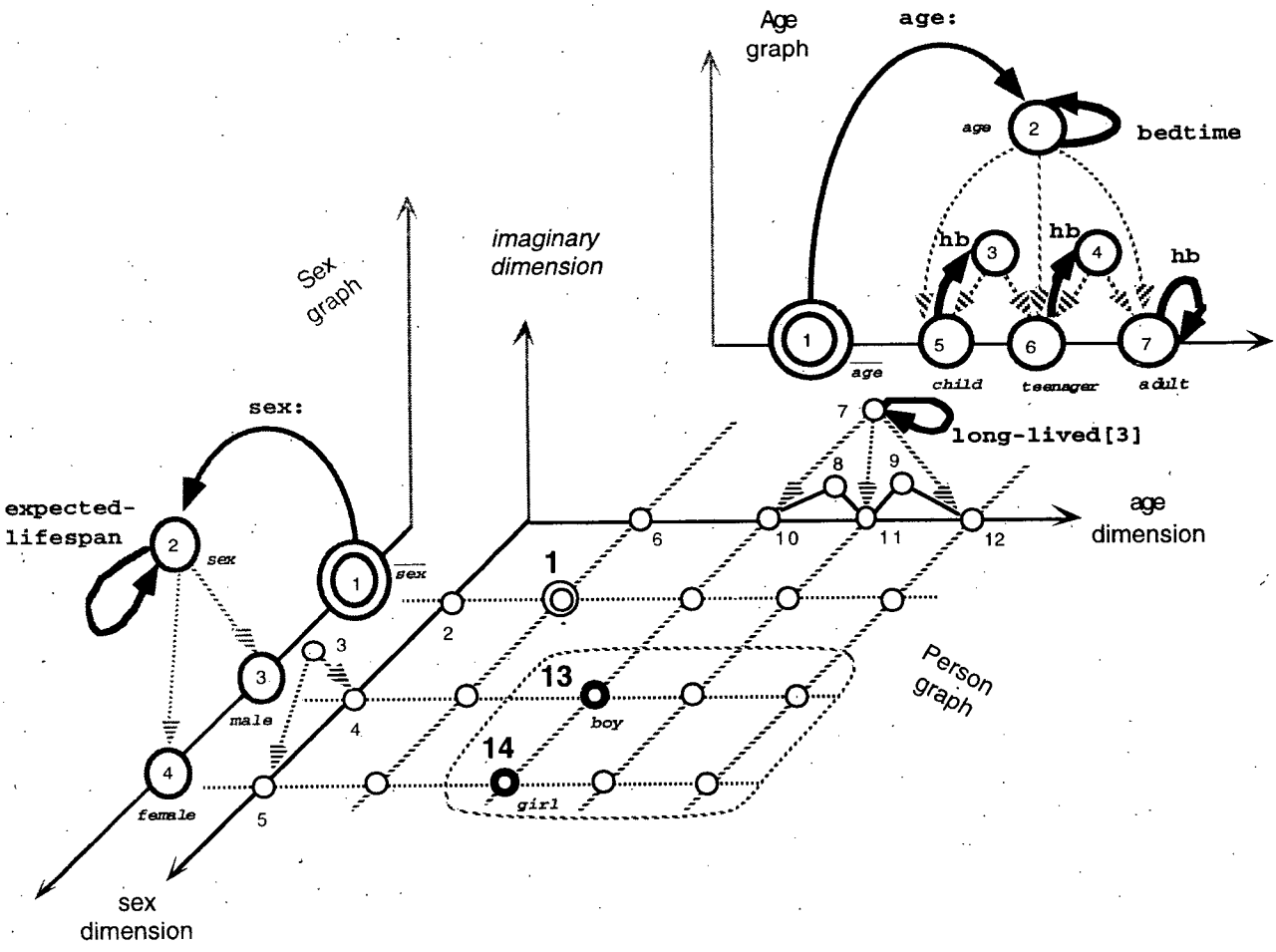


Figure 25.

Let's suppose the definitions of the *Sex*, *Age* and *Person* graphs already exist. In this case, the only thing to do is to specify the *Sex* and *Age* superclasses (in this order), relying on the **composition** operator for identifying the first subgraph of the *Person* graph with the first superclass graph (*Sex*), and the second subgraph with the second superclass graph (*Age*). Nothing else needs to be done since all remaining informations are part of the already known definitions (notably, the *boy* and *girl* nodes, as well as the *long-lived* transition). Concerning the implementation, this quite simple declaration will also imply the automatic inheritance of methods and memory representations attached to the graphs of superclasses *Sex* and *Age*.)

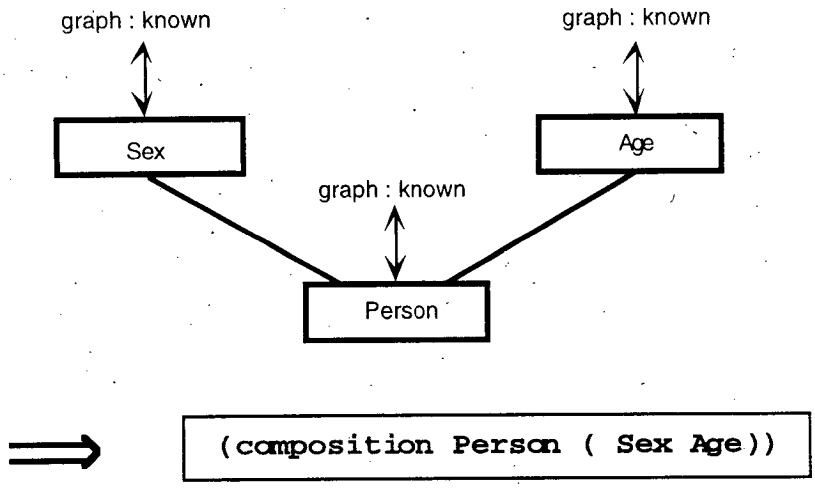


Figure 26.

Now, if we suppose that the definition of the *Person* graph does not already exist, then supplementary information should be provided. (See next figure).

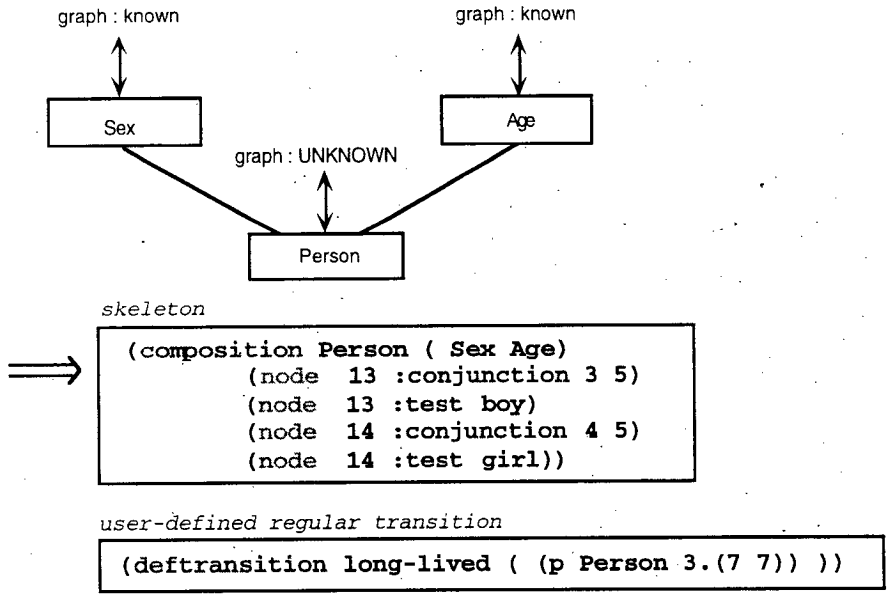


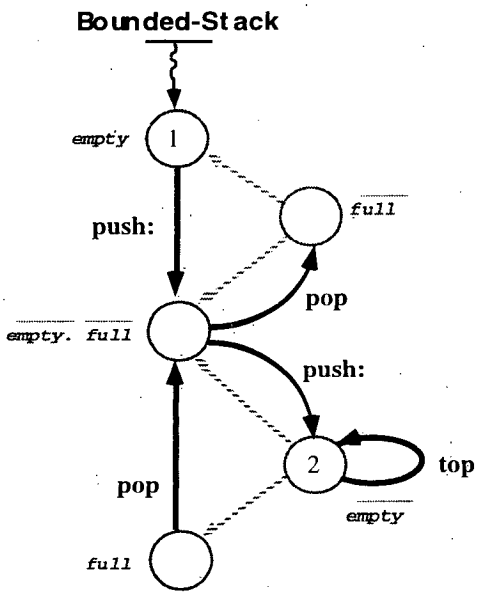
Figure 27.

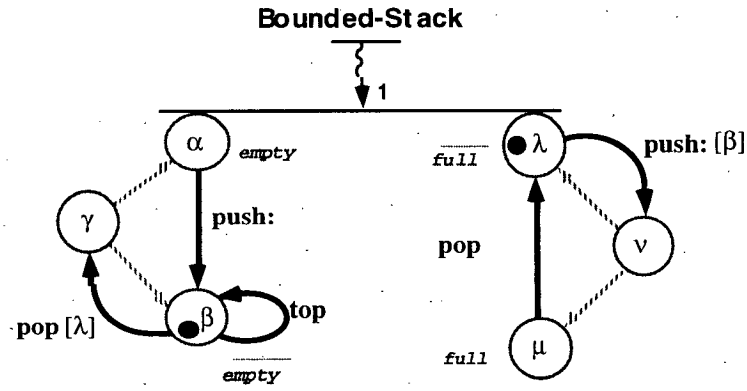
In any case, this example makes clear that the composition operator corresponds to the decomposition construct.

3.2 Derivation

3.2.1 Problem

Let's now consider the case of a *Bounded-Stack* instance. Its behavior may be modelled in a unidimensional way (see next figure), but a multidimensional model —making apparent an *emptiness* and a *fullness* dimensions— seems more natural (cf. the figure afterwards). Yet, this latter modeling is itself an ad hoc one : we would like to recognize the *Stack* graph itself and we cannot. Furthermore, because it is useful in many situations, *Bounded* should preferably be isolated as a reusable supplementary behavior. We are also interested in a solution which preserves the modularity of the potentials vs. the transitions and the conditions (the shown solution doesn't). Next subsection describes the design of the proposed solution.





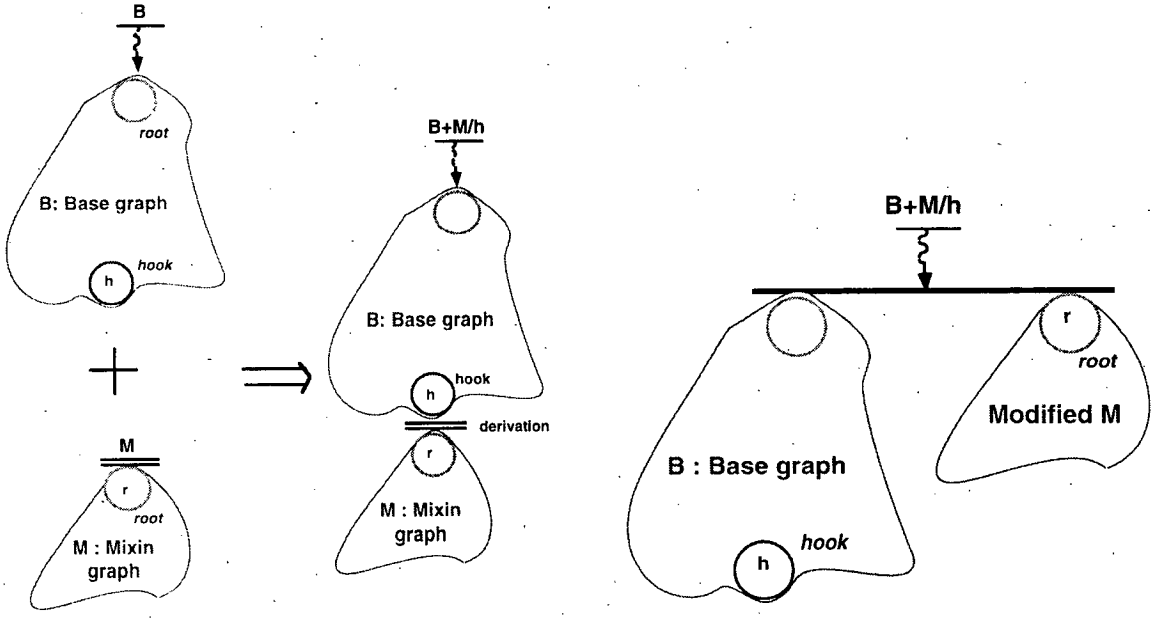
Figures 28 & 29.

3.2.2 Design

Here is the essence of our design (cf. [7]).

a) basic hypothesis

First, we hypothesize an abstract operator (termed the **derivation**) that can attach a **mixin** graph to one of the nodes (the **hook**) of a **base** graph. The base graph should be instantiable ; the mixin graph is not. The graph resulting from the attachment of the mixin graph to the base graph is termed the **derived graph** (see figure 30). The derivation is supposed to do this using a decomposition and a systematic setting of constraints on regular transitions (see figure 31).



Figures 30 & 31.

b) mixin attachment constraints

We add to the mixin external representation a **virtual hook** with three types of information (see next figure) :

1— a set of outcoming virtual transitions reflect outcoming base transitions, the only ones that do constrain the mixin transitions. 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**) ;

2— a condition of attachment $\mathbf{cond_{Hv}}$ that the instance should meet when in the actual hook (ex. : *not empty*) ;

3— a regular consulting transition **initPv** the instance should support to initialize the mixin potential (ex. : *length*).

These informations express constraints to be met for an attachment to be possible : a base candidate should observe them. Because the mixin is supposed to be reusable, all of them are abstracted from a particular use : they are given virtual names that may be changed at attachment-time.

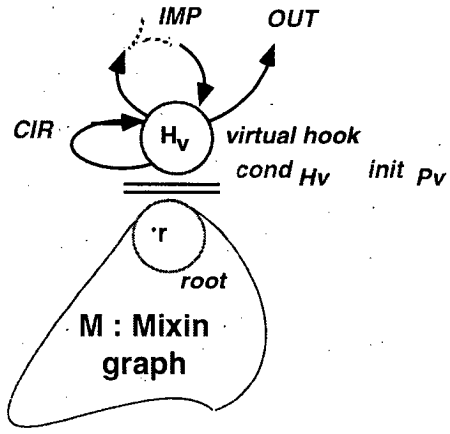


Figure 32.

The first and third constraints will be exploited below (§ c and § f).

The second constraint expresses that the mixin partitions the hook state : *condHv* should be equal to *condh*, the actual hook condition. Considering our *Bounded-Stack* example, the following steps are run for checking $cond_{Hv}$:

— in the virtual hook of *Bounded*, $cond_{Hv}$ is specified as being *not empty* ;
— the *Bounded* mixin graph is attached to the *Stack* graph in node 2 (actual hook) ;
— in this node, the condition *not empty* is met (definition of the *Stack* graph);
— thus $cond_{Hv}$ is satisfied.

c) mixin potential

The third constraint is optional : it enables the initialization of the mixin potential *Pm* from the base potential. Conceptually, we model the base potential at the virtual hook by a virtual base potential *Pv*. This potential corresponds to the value of the base potential as it is expected from the mixin. For example, the *Bounded* mixin expects *Pv* to be an integer value, the current *length* of the object (the number of *put:* messages minus the number of *get* messages). The computation of *Pv* from the actual base potential *Pb* is the responsibility of the base graph. We suppose the existence of a consulting transition, *initPv* (ex. : *length*), that the instance should support in the hook and which really concerns the base potential. Once the value of the virtual base potential is known, it is taken as the initial value of the mixin potential *Pm* : it is the responsibility of the mixin to copy it (*Pm = Pv*).

The information carried on by the base potential should be sufficiently rich to compute the virtual base potential *Pv*. Usually, the mixin potential is identical to the actual base potential. Yet, the base potential may well be more complex than the mixin potential. Suppose, for example, that the *Stack* potential keeps a list of all the messages *push:* and *pop* the instance has received : the *length* should be computed from this list.

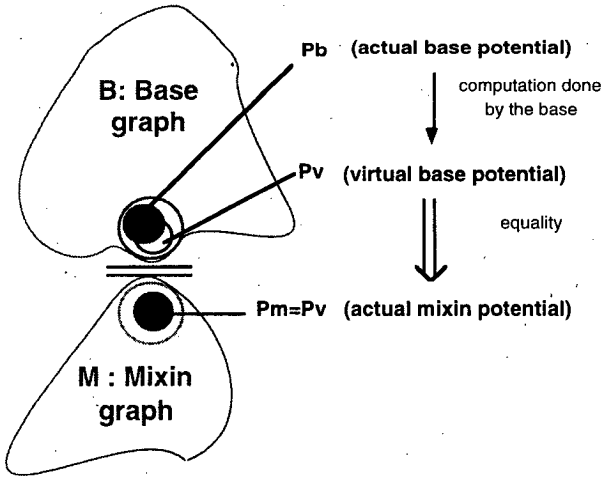


Figure 33.

Note that the condition $cond_{Hv}$ may be expressed in terms of the virtual base potential *Pv* . For example, in the case of the *Bounded* mixin, *condHv* is *(Pv > 0)*.

d) mixin structure

The mixin structure can be shown as being a multi-level selection, constrained by clause *[h]*, with one or several INITial nodes (see next figure). We term such a structure a **constrained subtree**. The clause specifies when the subtree is valid : the clause contents is a node or, more generally, a list of nodes (hooks). These hooks may well be ephemere nodes or the destinations of conjunctions.

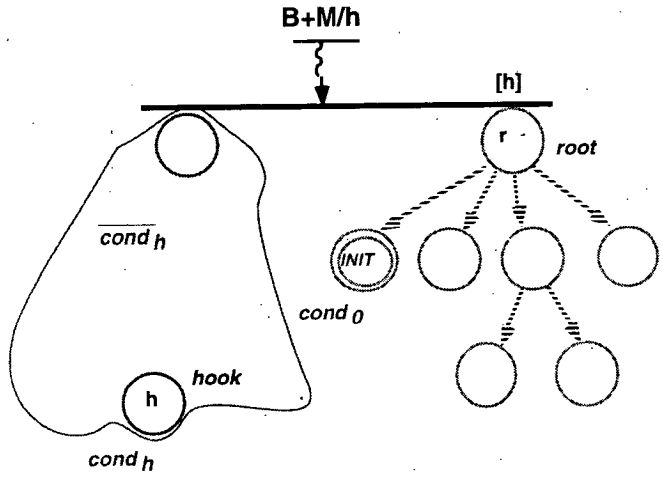


Figure 34.

As an example, let's consider the *Bounded* mixin. This one depends on the maximum number of elements the instance may store (be it *size*). Two basic nodes are defined, one by the condition *not full* and the other one by *full* (here, we implicitly suppose *size* to be greater than one). These conditions may be expressed in terms of the mixin potential : they are respectively *(1≤Pm<size)* and *(Pm=size)*. Note here that the condition at the root of the mixin ($cond_r$) may be more restrictive than $cond_{Hv}$. This is because the mixin may refine and thus constrain the behavior of instances. The root condition is effectively $cond_r$ = *(1≤Pm≤size)* while $cond_{Hv}$ is *(Pv>0)*. A "hole" thus exists. (Remember that *Pm=Pv*.)

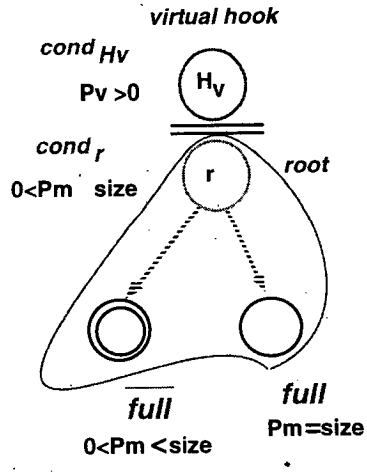


Figure 35.

e) parameterized mixins

Let's consider the *Bounded* mixin again. As just expressed, the above structure is valid only when the *size* is strictly greater than one. We require a fully general solution.

— a first one is to design a unique graph that works even when the *size* is equal to one. However, this solution will be inefficient because of the presence of selections handling the particular case "*size = 1*" (cf. the very first *push:* in *empty* and the very first *pop* in *full*) : these selections imply run-time testings[9] ;

— a better solution is to specify two mixins with a parameterized clause *[size=1]* and *[size>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 thus be incurred.

---

9 Figure 56 corresponds to this solution.

Next figure shows the general form of a parameterized mixin.

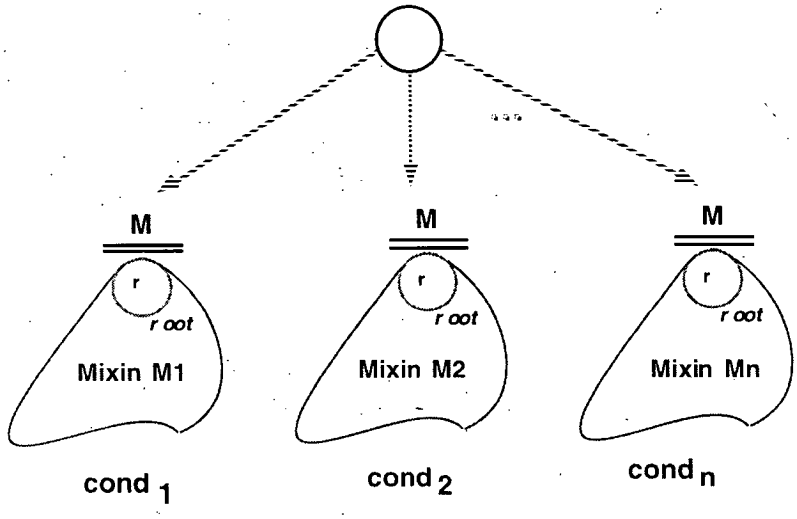


Figure 36.

f) constraints on regular transitions

The transitions that may exist inside the mixin critically depending on the OUT, CIR or IMP characteristic of the transitions declared at the virtual hook node. Any transition declared at the virtual hook should be defined in each basic node of the mixin (possibly via a group of basic nodes, i.e. via an ephemere node).

f.1) pure OUTcoming transitions

Any OUT transition in the mixin graph should be such that its destination meets *not* $cond_{Hv}$: we term this a **nowhere** transition (it is visually represented by an arrow that do not flow to any node). A possible convention is to replace a nowhere transition by an absence of transition.

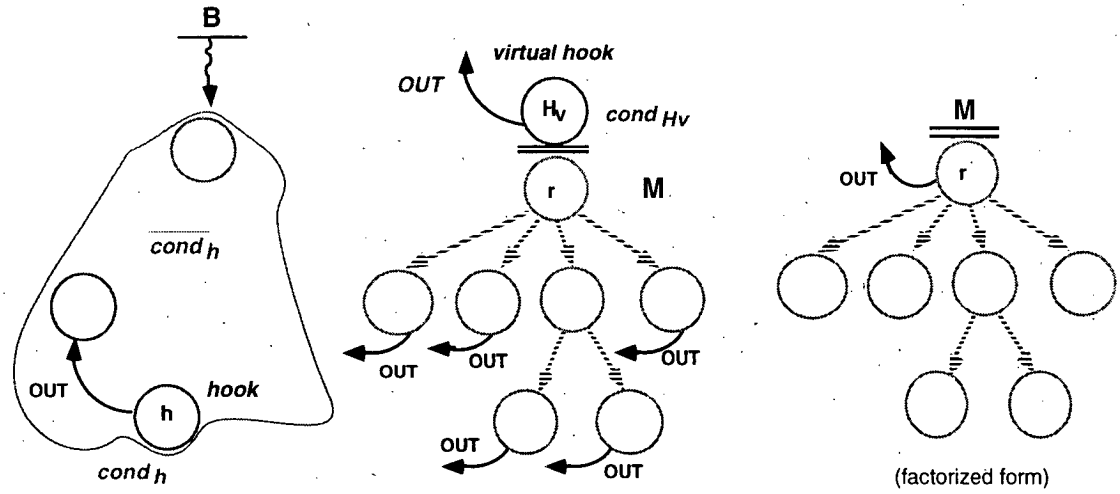
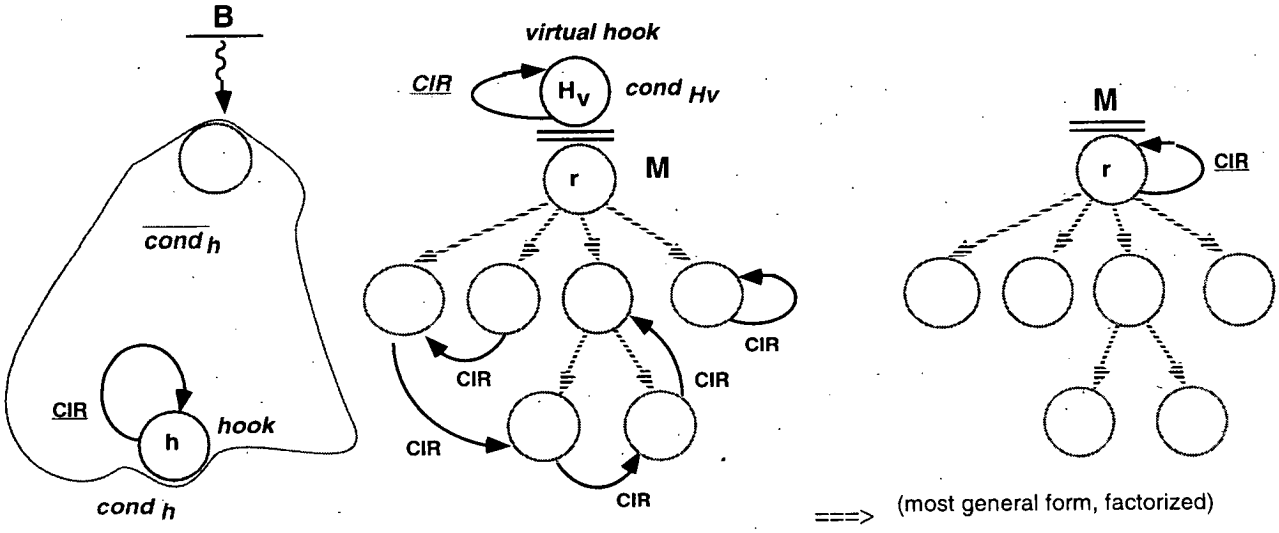


Figure 37.

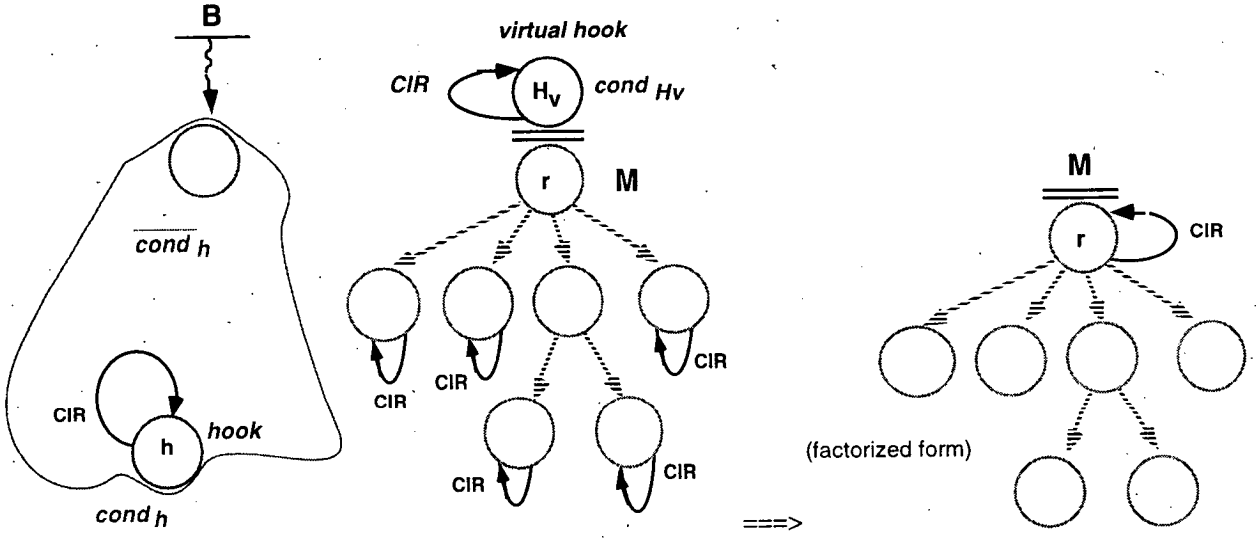
f.2) CIRcular transitions

Any similar transition is acceptable. Next figure illustrates the rule. The figure afterwards shows the most general case (all possible transitions) in factorized form at the root node (a "g-circular" transition).



Figures 38 & 39.

A special but frequent case occurs when the circular transition is side effect free (*testing* transition or transition specified as *consulting*). (This property is supposed to be recorded at the virtual node level too.) Then, whatever the considered substate of the hook, the transition which is attached to it is circular. Next figure pictures this. The figure afterwards shows the corresponding factorized form (an "i-circular" transition).



Figures 40 & 41.

f.3) IMPure outcoming transitions

We introduce a new notation : 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 analysis we made leads to the following conventions and rules :

(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 graph) ;

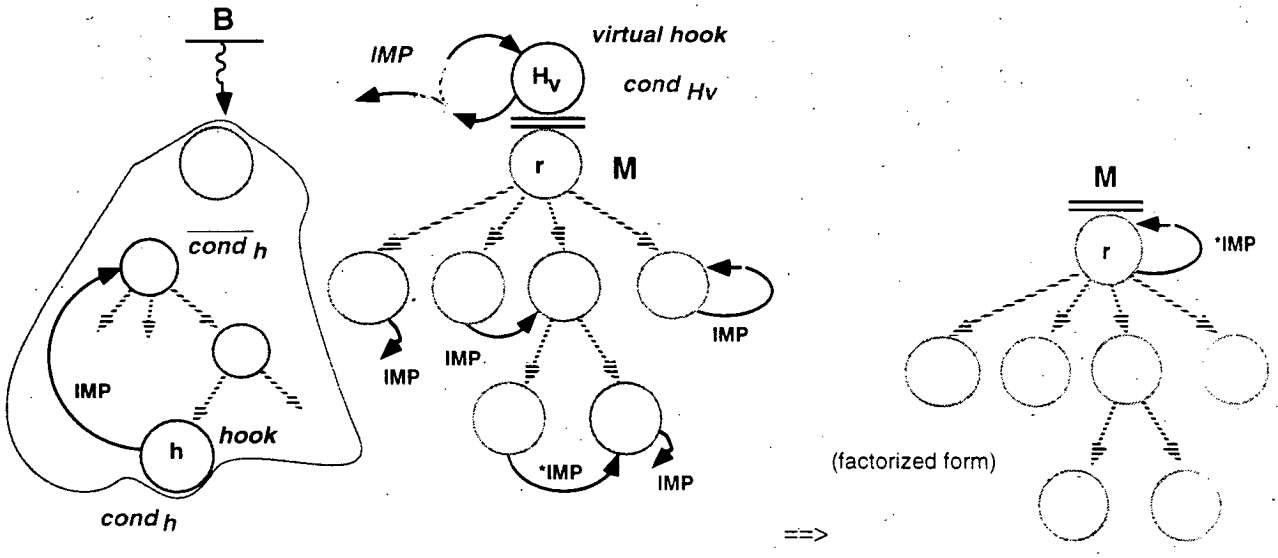
(2) if an unstarred transition IMP flows from x to y –be it circular or not, then the next state is supposed to meet $cond_{Hv}$ and it is y (or belongs to the set of substates denoted by y) ; $cond_{Hv}$ is not evaluated (nor $cond_h$) ;

(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 normally checked at attachment time).

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 graph for any impure transition at the base hook. This check can be done easily.

This set of rules may be used as such. The next figure illustrates all three cases.



Figures 42 & 43.

g) practical rules

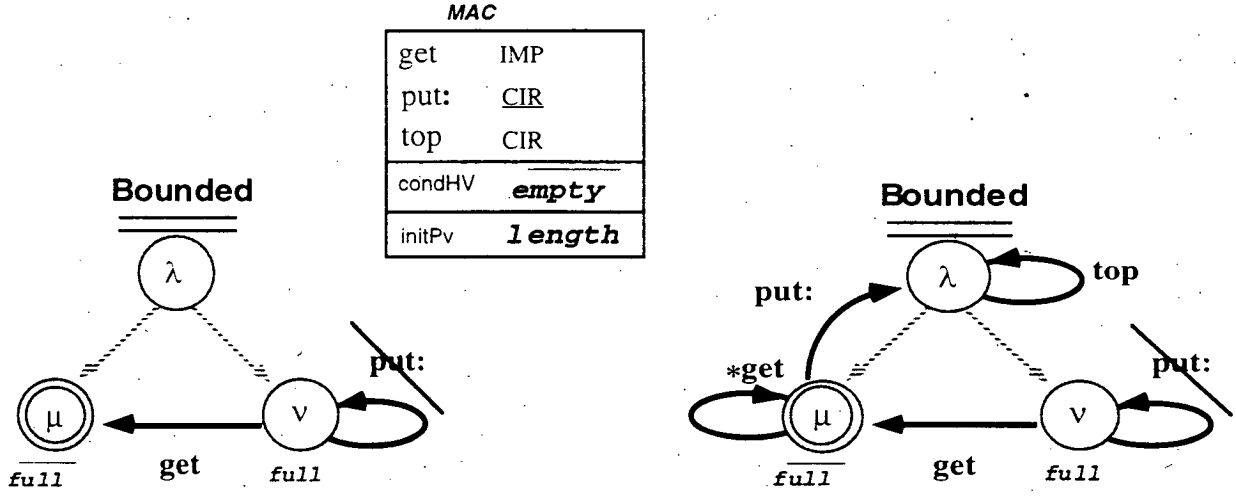
Practical rules were devised, based on the idea of specifying only transitions which differ from usual defaults (cf. [7]).

3.2.3 Example

Let's see how the *Bounded* mixin is defined and how *Bounded-Stack* is obtained.

a) The *Bounded* mixin

Next figure shows the *Bounded* mixin together with its Mixin Attachment Constraints table (MAC) ; this representation is drawn using our practical rules. The figure afterwards shows the same mixin once the (not overridden) default transitions have been added.



Figures 44 & 45.

Two basic states (substates of *not empty* in fact) exist : *full* and *not full*. When a *put:* message occurs in *not full*, a test occurs that possibly moves the instance state to *full*. In *full*, the *put:* transition is marked as **unwilling**. (This means the usual effect of a *push:* message is cancelled (redefined) ; however, the transition still formally exists to respect the usual constraints on transitions. For more details, see [7].) In *full*, a *get* makes the instance *not full* again.

This mixin supposes the *size* (maximum number of elements in the *Stack* instance) is strictly greater than 1. To remove this restriction, a parameterized mixin is to be used as explained in the above subsection (§e).

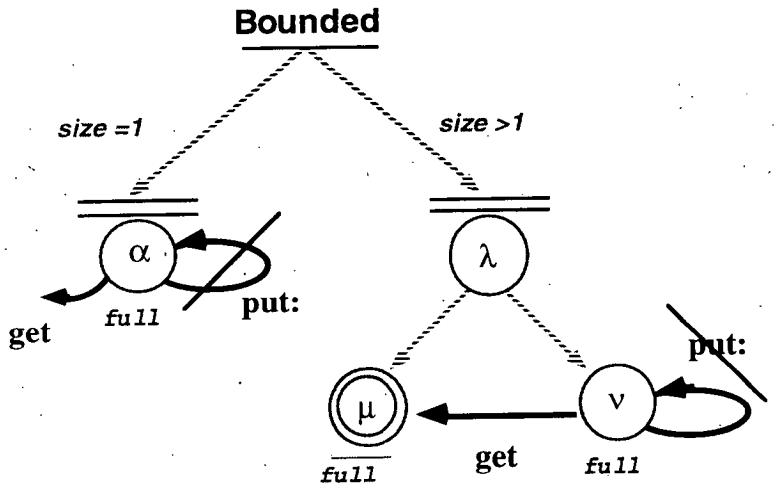


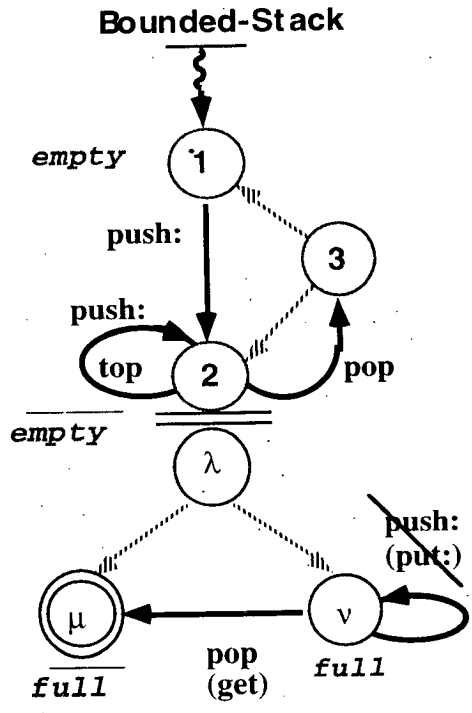
Figure 46.

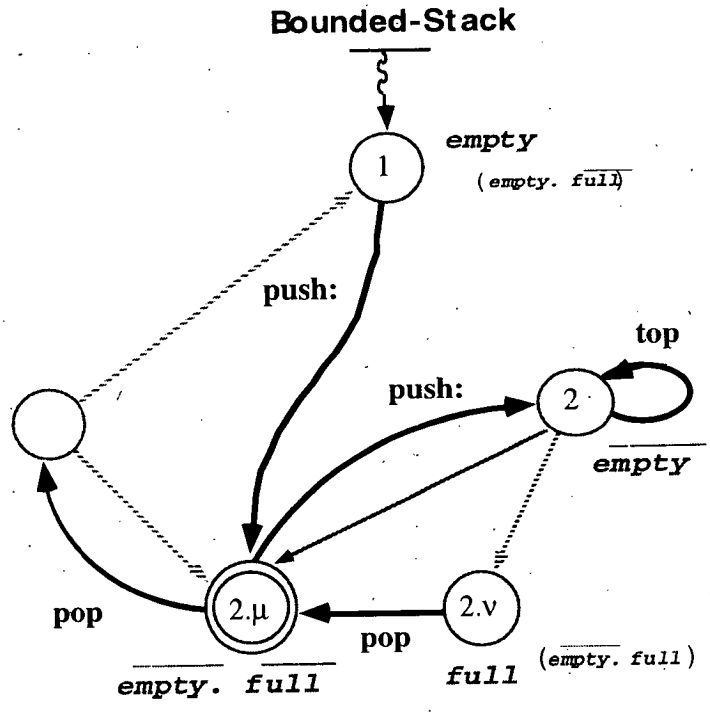
Note the added mixin (*size=1*) refines the specification of the IMP *get* transition as being a nowhere transition (instead of **get*) as if the specification was an OUT transition. This is correct but since both cases of **get* are not covered, the underlying system should warn the user.

b) The graph of *Bounded-Stack*

As shown previously, the graph of a *Stack* exhibits three nodes. When bounded, a *Stack* instance has a different behavior. Yet, this one is unchanged when in nodes 1 and 3 ; it is altered only in node 2 (*not empty*).

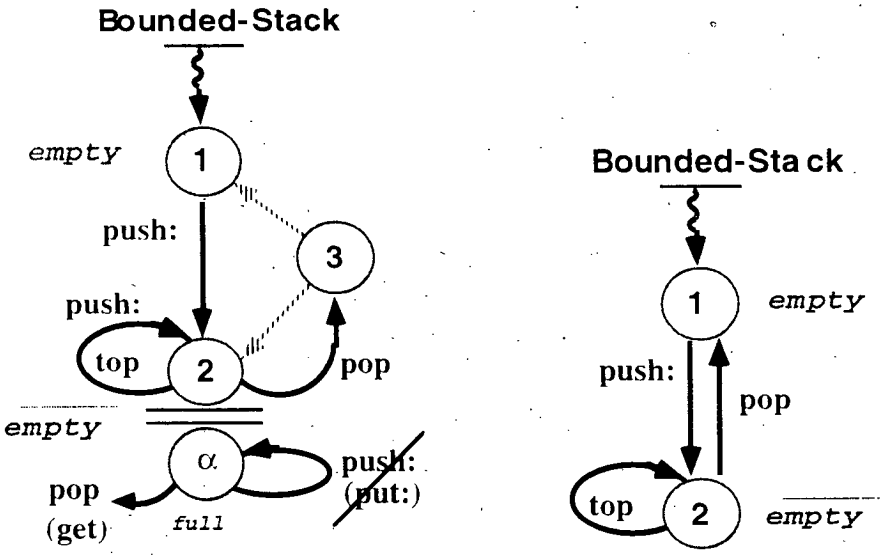
Next figure shows *Bounded-Stack* as a derivation of *Stack* using the *Bounded* mixin in case the *size* is strictly greater than one. Node 2 of *Stack* is the hook. Since the mixin has adopted fairly general names, some renaming is necessary : the mixin uses *put:* and *get* in place of *push:* and *pop*. To better demonstrate the actual behavior, the figure afterwards shows the equivalent graph obtained after expansion : first, we replace the mixin by its equivalent expression (according to figure 45) ; then we use the definition of the starred transition. We also change the destination of the very first *push:* transition to the INIT node *not full* and remove the unwilling transition *push:* by nothing (once alone, it is useless). The figure is drawn once the conditions have been simplified and the useless parts have been removed : the reader can check the obtained graph is the same than the one shown in figure 28. It is efficient (no useless tests).





Figures 47 & 48.

Next two figures correspond to the degenerated case (*size* = *1*). To better demonstrate the actual behavior, the second figure shows the equivalent graph obtained after expansion. This one does not contain an ephemere node any longer : the *pop* transition of the mixin being a nowhere transition, a *pop* message in state 2 moves the instance state immediately back to state 1.



Figures 49 & 50.

c) Interpretation in a multidimensional space

Next figure explicits the behavior of a *Bounded-Stack* instance (with *size* > 1) using the *stack* and *bounded* dimensions. (The *Bounded* mixin is exactly the same than in the above figure 45.)

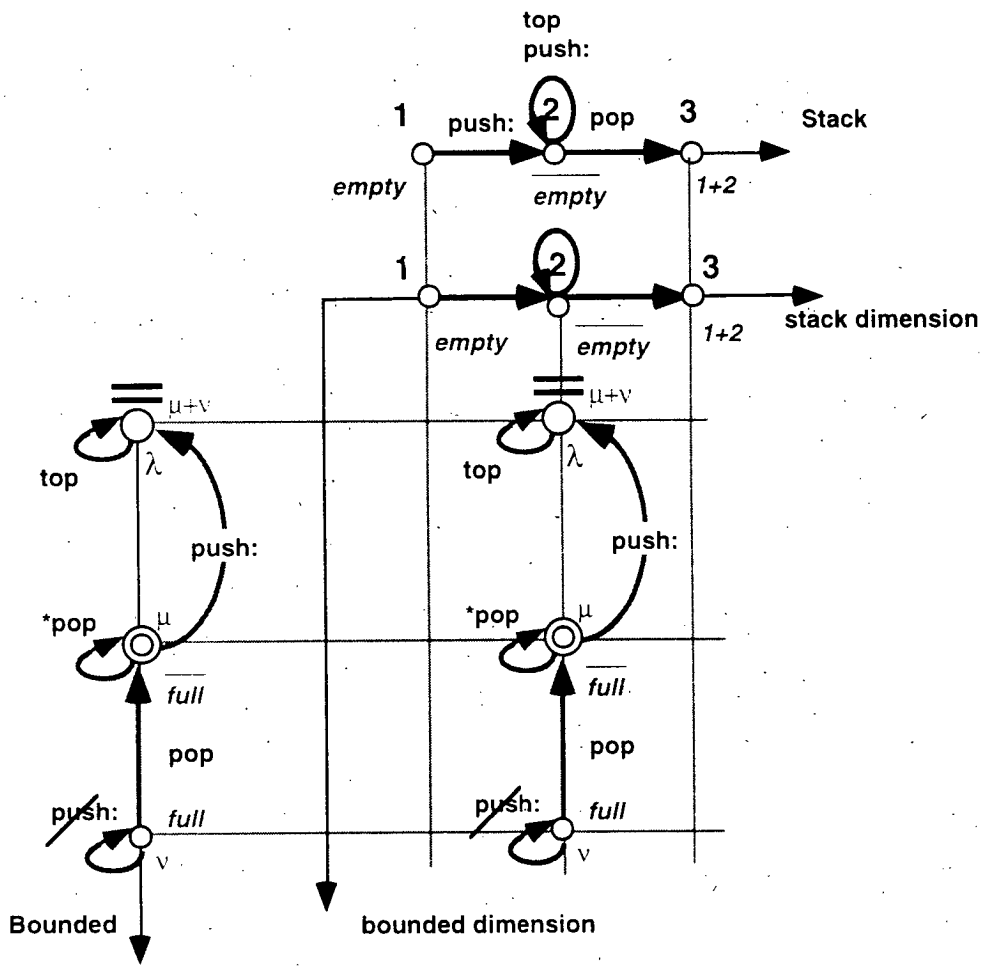


Figure 51.

Let's consider the *Stack* token. This one moves along the horizontal axis. When present in node 2, another token is created : this one is the *Bounded* token ; it moves along the vertical axis erected in node 2. This token is created in λ. λ is the source of a selection on μ (*not full*) and ν (*full*). The *Bounded* token automatically moves to *not full* because *not full* is declared as being the initial node of the selection. When the *Bounded* token is in *not full*, a starred *pop* message makes the test *not empty* to be checked. If the condition is not verified, the *Bounded* token disappears ; given the result of the test, the *Stack* token moves to *empty*. If the condition is verified, the *Bounded* token stays in *not full*. A *push:* message makes it move to the selection source. From there it goes automatically either to *full* or to *not full*. In *full*, besides the special behavior in case of a *push:* message, a *pop* message will make the *Bounded* token to move back to *not full*.

4. RELATED WORK

Our proposal derives in direct line from OOP and happens to be related to higraphs. These two main relationships are analyzed in the following two subsections. Only the main aspects are pointed out and discussed in length. Other points or related works are discussed in our research reports. Here are the main pointers : concerning the typestate concept [24], see [6] ; concerning objectcharts (these are *"extended statecharts in which the effect of state transitions on [object] attributes are specified"* [13]), see [8]. Concerning the relationships with work done in cognitive psychology, see [9] for the ergonomy of the visual formalism and [10] for the impact on the programming activity.

4.1 Object-oriented programming side

4.1.1 OOP in general

As announced in the introduction, we developed our modeling with the idea of pushing further the basic idea of OOP, which –in our view– is to give data-structures (objects) the responsibility they can handle. 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 basic idea of OOP is excellent, but appears as having been not pushed to its ultimate whenever possible. Considered from our viewpoint, a traditional class is like a single point (it offers a single dictionary of methods). Whereas our approach distinguishes several coordinates on each dimension and thus exhibits a number of points for a given class, traditional OOP aggregates them into one. To take a metaphor, traditional OOP considers a class like an atom whereas our approach opens it and reveals an internal structure. This deeper understanding enables a better modeling ; concepts get purer.

4.1.2 Predicate classes

The *Person* graph depicted in this paper is derived from the actual code of the *Person* example given in [12]. This work, the closest to ours we know of when considering the strict OOP filiation, promotes the concept of "predicate classes".

Like regular classes, "predicate classes" specify slots and methods that affect the behavior of objects. But, different from regular classes, they are attached a predicate (what we named a condition in our 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. 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 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 node in a class graph) and using it to directly access the adequate dictionary. This is akin to having a dictionary per node. *"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 multidimensional graph.

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

(1) the level of abstraction : a class graph abstractly models the behavior of instances while a predicate class consists in actual implementation code ;

(2) the (structural) specification of the next state. In our graphs, a 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 graph has been well devised[10] : 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 our approach, these required optimizations are somehow directly captured into the class graphs (by the specification of transition destinations) : optimizations are thus done without effort. To summarize, a class 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 class graph whereas the opposite cannot be done. Somehow, we can state that our approach is "predicate-classes + destinations". For example, the next figure —which abstracts the source code given in the cited reference— can be derived from the *Person* graph of figure 19. (Predicates classes are shown in rounded rectangles ; methods, as tiny circles.) For more details, see [6].

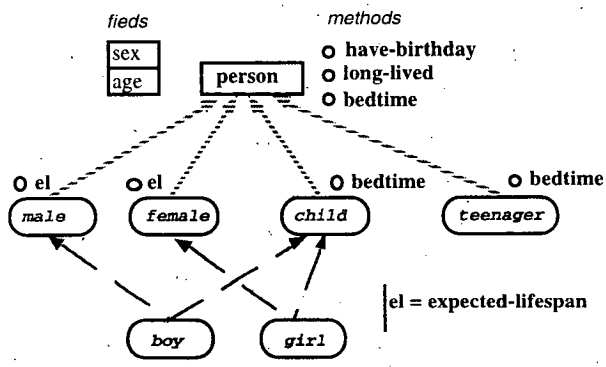


Figure 52.

4.2 Higraph side

4.2.1 Higraphs

a) Insideness vs. connectedness

We already noted that using insideness in place of connectedness leads to a form of higraphs that we can term "class higraphs". Since insideness is essentially meant to represent hierarchical relationships, multiple inheritance cases (such as node 11 or 13 in figures 19 and 22) should be represented differently in case of a pure insideness style. The next two figures show this possibility and its equivalent in class graph form. A different option would be to mix both styles (note that the statechart formalism uses both : cf. in [20] the joint and fork connectors (AND-type as our conjunction and disjunction) ; the condition, selection and junction connectors (OR-type as our selection)).

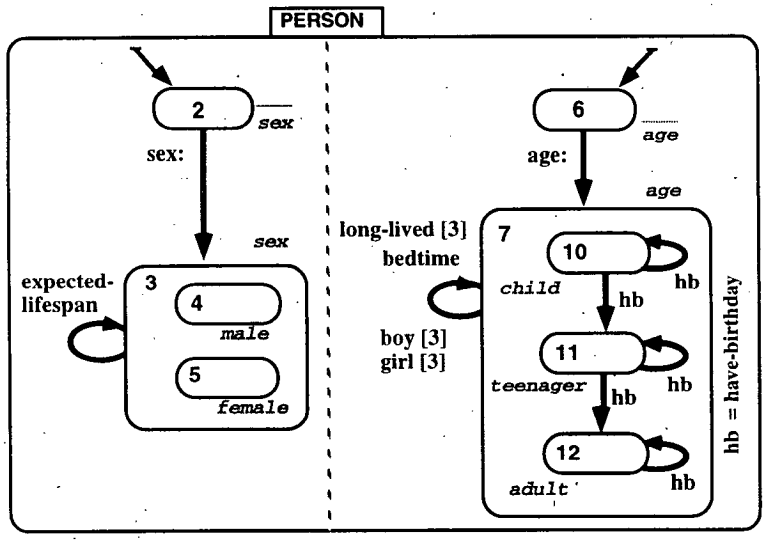


Figure 53.

[10] The *Person* graph illustrates this point : modeling *have-birthday* as a g-circular transition would not be optimal (yet, the correct result would be obtained).

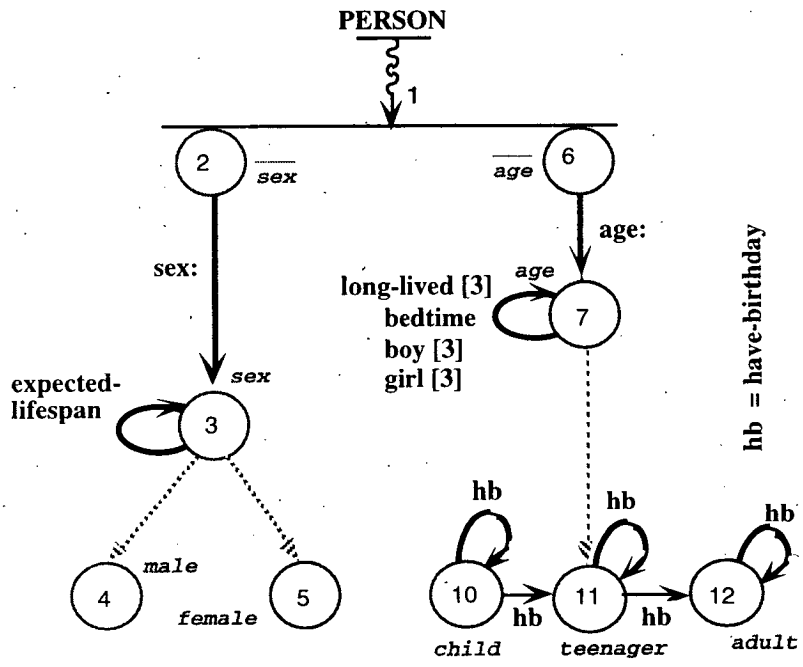


Figure 54.

As reported by references [14] and [15], 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. We are not going to aliment this debate : for the moment, we consider that a realistic system should support both styles depending on the user's choice. If a serious cognitive study based on actual observations were to draw definitive conclusions, these will be taken into account.

b) Important properties of higraphs valid for class graphs

In his papers, Harel insists much on partitioning 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 partitioning, 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).

As noted previously, the decomposition construct in class graphs is equivalent to the partitioning in higraphs, while transitions to ephemere nodes in class graphs are equivalent to hyperedges in higraphs. If we consider a given class higraph, its class graph equivalent can be built using this correspondence. Concerning this equivalent, the above claims are obviously maintained. They are also maintained for class graphs in general.

4.2.2 The McGregor-Dyer proposal

Concerning our proposal for mixins, [22] seems to be the work which, by many aspects, is the closest to ours.

a) 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"* ;

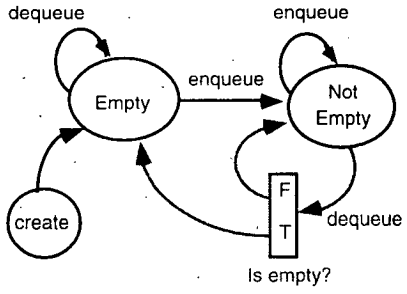
(2) *"Any new state introduced in a child class is wholly contained in a existing state of one of the parent classes"* ;

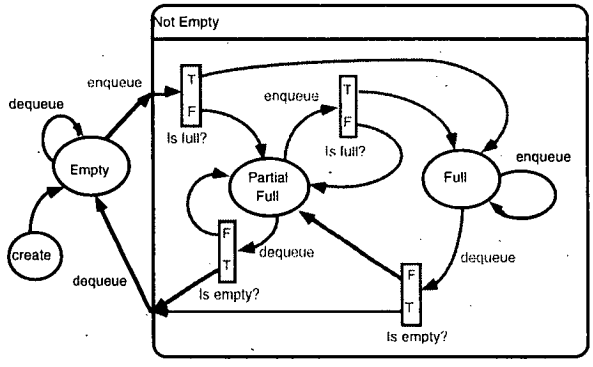
(3) *"A child class may not delete a transition from the state machine of one of its parent classes".*

Our 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.

b) The *Bounded-Queue* example

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 [13]. The goal is to express the inheritance relationship between a *Queue* (figure 55) and a *Bounded-Queue* (figure 56).





Figures 55 and 56.

A comparison can be made with the *Stack* graph (figure 2) and the derived *Bounded-Stack* that uses the parameterized *Bounded* mixin (figure 46) : because the transposition from *Stack* to *Queue* is obvious, we will speak about the *Queue* of figure 2 and the *Bounded-Queue* of figure 47 (when *size >1*) or figure 49 (when *size =1*). The transition names given here are those used in the cited reference.

We can make four remarks :

— the first one is about the formalism of the cited paper : concerning the *Queue*, it corresponds almost exactly to the graph of figure 2. In other words, it is based on connectedness. The vertical box containing the notation *T(rue) F(alse)* corresponds to a selection construct, yet in a less general manner. (Note that we didn't draw a *dequeue* transition when the instance is *empty*.) Concerning the *Bounded-Queue*, the notation used in the cited reference mixes connectedness and insideness ;

— the second remark is of importance : the cited paper does not extract the *bounded* behavior into an independent structure (mixin) to make it reusable. As a matter of fact, none of the examples shown in the cited paper abstracts away a supplement of behavior into a reusable construct. Reusability will only be obtained by copying part of the objectchart, which is by far a less good solution than ours ;

— the third remark is about modularity : the cited *Bounded-Queue* state diagram mixes all cases (when the *size* is set to one, a single *enqueue* sets the instance state to *full* ; a single *dequeue* sets the instance state back to *empty*). Our formalism covers all the cases but clearly distinguishes the case *(size=1)* from the case *(size>1)* thanks to a parameterized mixin (figure 46). The connection of the *Bounded* mixin to the *Queue* graph is also much simpler than the hardwired solution proposed in the cited paper. In this one, the *Queue* diagram is not kept as such, but profoundly modified : the circular *enqueue:* transition is removed as well as the squared testing box (this one appears twice inside the sub-diagram replacing the *not empty* state) ;

— the fourth remark is about efficiency : our modular proposal also corresponds to a simpler working in final. As explained before, the parameterized mixin is used while not impairing the efficiency (the useless submixin is discarded) : thus no tests about parameters (here, the *size*) are going to be mixed with the tests to be done for marking instance states. This is not the case with the state diagram proposed in the cited paper.

5. CONCLUSION

In an overview of methodologies (termed methods) in OOP [1], Aksit and Bergmans, the authors, wrote : *"Most methods (...) consider states as an important aspect of object-oriented software development. (...) Although these methods consider states as an essential issue in object-oriented programming, they do not address the integration of states with inheritance. (...) Only OMT (...) considers the issues of generalization and/or specialization of state specifications as significant. Specialization of state diagrams is partially possible. (...) We claim that a notation for the specification of state diagrams should be suitable for extension by subclasses. (...)".*

The work described in this paper proposes such a notation.

The behavior of the instances of a class are modelled with a graph in a multidimensional space using a few number of simple and well organized concepts. A class graph describes possible instance states and regular transitions between these states. Regular transitions model the existence and effect of messages (single dispatch) or generic function calls (multiple dispatch). A class graph is structured : it is built using three types of constructs (the decomposition, the selection and the optional conjunction). The dynamic semantics of these three constructs may be expressed using a more basic element, the reflex transition. Reflex transitions are also the support for the inheritance of transitions —as well as for implementation items— in one class graph (local inheritance).

Two operators enable abstract implementations, hence a hierarchy of class graphs. The composition operator (which directly derives from the decomposition construct) makes a class inherit from several base classes ; the derivation operator (which is a constrained form of composition) enables a base class to be extended with (a) possibly parameterized mixin class(es) in a checkable and very modular way.

Our work also proposes a simple yet run concept related to invariants. Named "potential", it gives regular transitions an abstract body and enables checking and simulation in a modular way, i.e. dimension by dimension (including for mixins).

Bibliography

[1] Aksit & Bergmans. *"Obstacles in Object-Oriented Development"*. OOPSLA'92 Proceedings.

[2] Bellamy & Carroll. *"Case-based reuse"*. ACM SIGPLAN Symposium : Object-Oriented Programming Emphasizing Practical Applications. 1990.

[3] Borron *"Types and Type-States in LeTool"*. ESPRIT Conference '87, pp. 1276-1287

[4] Borron. *"Colored-Object Programming : Describing Interfaces"*. GL'95 Proceedings. (Huitièmes Journées Internationales sur le Génie Logiciel et ses Applications.) 1995.

[6] Borron. *"Colored-Object Programming : color graphs, a visual formalism for synthesizing the behavior of objects"*. February 1996. Revised, april 1996. RR 2876.

[7] Borron. *"Colored-Object Programming : mixin and derivation, two conjoint concepts for a rigorous handling of independent supplementary behaviors"*. February 1996. Revised, april 1996. RR 2877.

[8] Borron. *"Colored-Object Programming : Inheritance by dimensions"*. October 1995. Revised, april 1996. RR 2878.

[9] Borron. *"Colored-Object Programming : About the ergonomy of the visual formalism"*. January 1995. Revised, april 1996. RR 2879.

[10] Borron. *"Colored-Object Programming : About the programming activity"*. January 1996. Revised, april 1996. RR 2880.

[11] Borron. *"Colored-Object Programming : Ergonomic and cognitive issues"*. (May 1996). ERGO-IA'96 Proceedings.

[12] Chambers. *"Predicate classes"*. ECOOP'93.

[13] Coleman, Hayes & Bear. *"Introducing Objectcharts or How to use statecharts in Object-oriented design"*. IEEE Transactions Software Engineering, Vol 18, # 1. 1992.

[14] Fitter & Green . *"When do diagrams make good computer languages ?"*. International Journal Man-Machine Studies. Vol.. 11. 1979.

[15] Green. *"Pictures of programs and other processes, or how to do things with lines"*. Behavior and Information psychology. Vol. 1, # 1. 1982

[16] Green. *"Cognitive dimensions of notations"*. Sutcliffe & Macaulay (Eds.), People & Computers (Vol 5). Cambridge University Press. 1989.

[17] Harel. *"Statecharts : a visual formalism for complex systems"*. Science of Computer Programming. Vol. 8, # 3. 1987.

[18] Harel. *"On visual formalisms"*. Communications ACM, Vol. 31, # 5. 1988.

[19] Harel & Gery. *"Executable object modeling with statecharts"*. 18th Conf. Soft. Eng. 1996.

[20] Harel & Naamad. *"The STATEMATE semantics of statecharts"*. ACM Trans. Soft. Eng. Method. 1996.

[21] Lange & Moher. *"Some strategies of reuse in an object-oriented programming environment"*. Bice & Lewis (Eds.), Proceedings of CHI'89.

[22] McGregor & Dyer. *"A note on inheritance and state machines"*. Software Engineering Notes, Vol 18, # 4. 1993.

[23] Rist *"Plans in programming : definition, demonstration and development"*. Empirical Studies of Programmers. Ablex. 1986.

[24] Strom & Yemini. *"Typestate : A Programming Language Concept for Enhancing Software Reliability"*. IEEE Transactions Software Engineering, Vol. SE-12, #1. 1986.

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