The Visual Model of Cordial[1]

Luis O. Quesada, Camilo Rueda, Gabriel Tamura
{lquesada, crueda, gtamura}@atlas.ujavcali.edu.co
Universidad Javeriana de Cali

ABSTRACT

We describe the visual model of *Cordial*, a visual language integrating Object-Oriented and Constraint programming. The motivation behind *Cordial* is to provide a clear notion of objects defined implicitly by means of constraints. *Cordial* is a visual language having three distinguished features: (1) A hierarchical visual model, (2) an underlined visual formalism giving precise syntax and static semantics of visual programs and (3) a dynamic semantic model based on a formal calculus integrating objects and constraints. The visual model consists of a hierarchy of layers of visual representations of object oriented concepts in which icons can be "expanded" upto the underlined visual formalism, an extension of Harel's *Higraphs* (Harel[89]). We present here the visual model, provide its formal translation into *Higraphs*, and describe the visual syntax and semantics of *Cordial*.

Keywords: Visual language, *Cordial*, constraint programming, object oriented programming, visual formalism, iconic programming.

Introduction.

Research in visual languages has known increasing activity in the last years. A variety of successful visual languages both general purpose and application specific have been defined recently. The range of visual models employed could be loosely characterized as based on actions over icons [HTI90], graph representation of control [Sco95], spreadsheet based [BA94], based on spatial relations [NK91], based on topological relations [KS90] and programming by demonstration (Hübscher[96]). *Cordial* is a visual language having three distinguished features: (1) A hierarchical visual model, (2) an underlined visual formalism [Har88] giving precise syntax and static semantics of visual programs and (3) a dynamic semantic model based on a formal calculus integrating object-oriented and constraints programming paradigms. The motivation for the design of *Cordial* is to provide an effective way to represent visually the computation of partially defined structures. These are very useful in applications such as musical composition where the notion of classes of objects obeying precisely defined rules is central. Indeed, although *Cordial* is a general purpose language, its principal aim is to provide a coherent base to develop computer aided musical composition systems. This domain is particularly demanding in at least three aspects of computation, easy program interaction with visual representations of data (musical scores), implicit definition of complex structures by sets of simple rules, and powerful operations for data structures transformation. *Cordial* is visual to address the first issue, constraint-based to address the second and object-oriented to provide for the third. The development of *Cordial* is a joint effort of researchers in three Colombian universities and in IRCAM, a musical research center in Paris, forming a research group called AVISPA. In this report we describe only the visual model of *Cordial*.. The underlined semantic model for the integration of constraints and objects is given elsewhere [VDR97]. The paper is organized as follows. In section 2 we define the basic elements in the visual model and place it in the context of existent visual languages. Section 3 pinpoints visual features of *Cordial* by discussing a programming example. Section 4 extends the *Higraph* [Har88] formalism and describes how any visual program in *Cordial* can be translated into it. Finally, in section 5 we give some conclusions and propose future works.

[1] This work is supported in part by grant 1251-14-041-95 from Colciencias-BID.

The Visual Model

The visual model of *Cordial* inherits ideas of both *Prograph* and *Pictorial Janus*. However, it departs from them in several important aspects. *Prograph* is based on the Object Oriented paradigm. Visually, *Classes* are represented as templates containing boxes for defining *atributes* . *Methods* of a class are defined externally by a contour containing a graph representing the flow of control of method invocations in the body of the method's definition. *Instances* are thus not explicitly represented. They exist implicitly as the result of invocations to a predefined *make* method. Methods defined this way are more akin to the notion of *generic* functions. In *Cordial*, instances are represented explicitly by means of a user defined *form*. Methods are defined by a layout of forms defining a pattern of *messages* sent to instances. In *Prograph* the visual elements are fixed. In *Cordial*, the user can associate new forms to language elements. *Pictorial Janus*, on the other hand, is based on the concurrent constraint programming paradigm [Sar93]. Closed contours define rules. These rules contain (and are activated by) *agents* represented also by closed contours. Both rules and agents may contain *ports*, which are small contours defining input and output "sockets" for interaction. Agents select subcontours inside rules by a process of visual pattern matching linking agents to inputs of the rule. The result of this visual pattern matching is a reduction of the agent (and rule) to the selected subcontour. Much like in *Pictorial Janus*, closed contours in *Cordial* are used to define relations. However, the graphical elements in both languages are quite different since the architecture of the visual model in *Cordial* reflects the object-oriented paradigm. *Syntactically*, a program in *Cordial* is a layout of forms on the screen. The visual vocabulary includes closed contours, icons, labels, line segments and arrows. These are used to represent different elements of a program and its execution. The underlined semantic model of *Cordial* is based on an integration of objects and constraints. User defined icons or forms are used to represent objects whereas line segments or arrows touching icons define relations. *Cordial* allows different layers of visual representations for a program. Each form in the program has an interpretation (or *expanded view*) in a lower layer. The ultimate layer achieved by expansions is the representation of the program elements in *Higraphs*, the underlined visual formalism.

The notion of a program in *Cordial* includes all the forms together with the functions associating them with their *expanded views* . In what follows we assume a given set F of *forms* and a set H of *views*, $F = F_C \cup F_i \cup F_m \cup F_r$, and $H = H_C \cup H_i \cup H_m \cup H_r$, where F_C, F_i, F_m, F_r (respectively, H_C, H_i, H_m, H_r) are sets of *forms* (*views*) for *Classes*, *instances*, *methods*, and *instance representatives*, respectively. Forms are thus visual representations of the well known concepts of OO programming, where *Instance representatives* could take the role of local variables in methods. Although this straightforward association can be useful at the beginning, we will have to depart from it in subtle but important ways when we precise the notion of *method* as a relation. Let $I_C : F_C \rightarrow H_C$, $I_i : F_i \rightarrow H_i$, $I_m : F_m \rightarrow H_m$, $I_r : H_m \times F_r \rightarrow H_r$, be functions mapping forms with expanded views in the underlined visual formalism (note that instance representatives are expanded within the context of the expanded view of a method in which they appear). Programs in *Cordial* are essentially a collection of forms defining classes. More precisely,

Definition 2-1

A *Program* $P = \langle F_C, I_C, m \rangle$ comprises a set F_C of forms for classes, a function I_C mapping them to expanded views, and a message *m*. The message "triggers" a method invocation. A *Class* $C = \langle F_i(C), F_m(C), I_m(C) \rangle$ is a collection of instance forms, method forms and a mapping of method forms into their expanded views.

Every instance form $o \in F_i(C)$ is unique. That is, for all classes C_k, C_j $(k \neq j)$, we have $F_i(C_k) \cap F_i(C_j) = \varnothing$. A given method form may appear in different classes. *Class*, *method* and *instance* forms can be seen as labels referring to an expanded definition in the visual formalism. Figure 2-1 gives an example of the definition of two classes, "SUNS" and "WEATHERS".

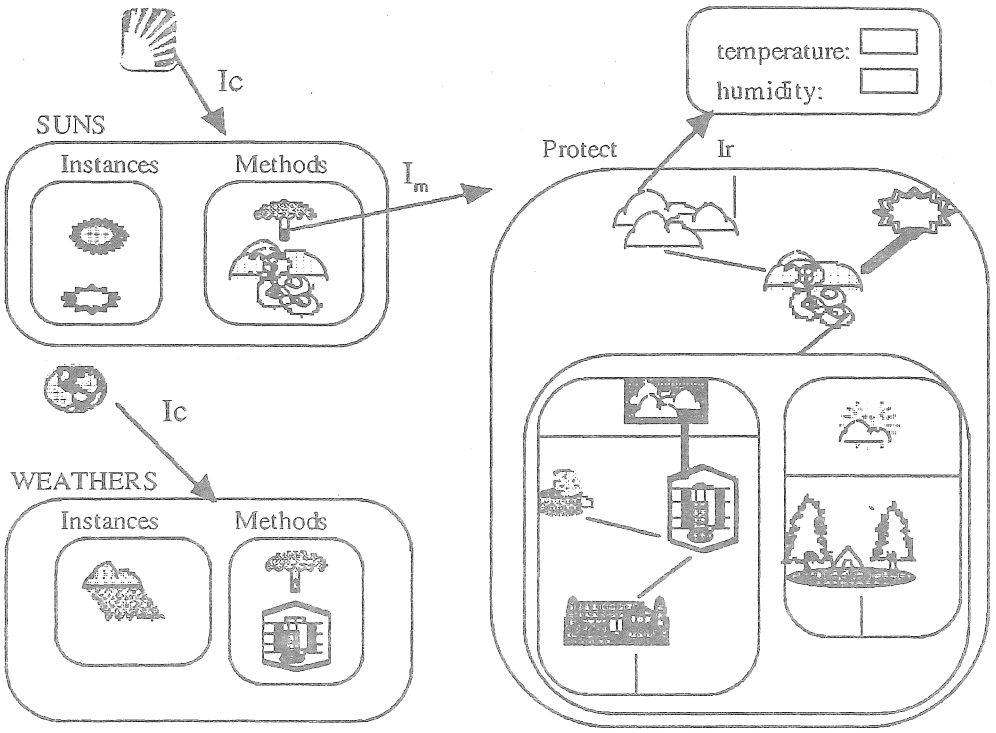

Figure 2-1 (left) Visual definition of two classes. (right) A method with two guarded subsets

Instance forms represent unique object identifiers (they exist, of course, only at program execution). Instance *representatives*, appearing in methods, are forms defined by the user to represent an element of the set of instances of a class. The particular form of an instance representative should bear resemblance with the form of the class of the instance. This is shown in Figure 2-1 by using the same form of the class with different background colors. A method form represents a set of messages. This set is made *visible* by the mapping I_m. The set of messages can be partitioned into disjoint subsets, identified by closed contours inside a single contour. Message subsets are interpreted as concurrent computational paths in the underlined semantic model. Each subset is usually *guarded*. Guards are used to select the appropriate computational path. Figure 2-1 shows an example of guarded message subsets. Guards appear in the upper part of the subset. A "wind blowing" message is sent to a "sun" instance representative (the receptor of a message is indicated by a thick line). The message has two arguments, so "wind blowing" defines a ternary relation. Forms in the guards ("dark cloud" and "partially sunny") are alternative arguments tested for membership to the relation. We approach next the notion of computation in the visual model of *Cordial*.

Definition 2-2

Let functions $\mathrm{Ins}: F_C \rightarrow 2^{F_i}$, $\mathrm{Classof}: F_r \rightarrow F_C$ be such that $\mathrm{Ins}(f_C)$ is the set of all possible instances in class C and $\mathrm{Classof}(f_r)$ is the class of instance representative f_r. Let $f_{r_1}^{(m)}, f_{r_2}^{(m)}, \ldots, f_{r_n}^{(m)}$ be the instance representative forms appearing in a method *m*. The *computational space* of a method *m* in a program P is defined as $\mathrm{MCS}_P(m) = \underset{1<i<n}{\times} \mathrm{Ins}\left(\mathrm{Classof}(f_{r_i}^{(m)})\right)$, The computational space of a method is thus the product of all sets of instances represented by forms appearing in it. Each method *m* defines a *method solution space*. $\mathrm{MSS}_P(m) \subseteq \mathrm{MCS}_P(m)$, which is a refinement of its computational space. ■

The execution of a program in *Cordial* proceeds by message passing to instance representatives in the expanded view of a method. The visual program of the method triggered by the message is then run, thus possibly generating new

messages. There is no predetermined order in which messages of a method are taken into account. A message denotes a relation. Instance representatives forms involved in the message are "refined" so that the number of instances they can represent is reduced to those satisfying the relation. Each step in this process is called a *configuration*. Going from one configuration to another is determined by a binary relation R on configurations defined in the underlined semantic model. More precisely,

Definition 2-3

A *configuration* is a pair (M,s), for M a set of messages and s a *state*. A *message* $t \in M$ is a tuple $(f_m, f_1, f_2, ..., f_n)$, where $f_m \in F_m$, and $f_i \in F_r, 1 \le i \le n$, are forms of instance representatives. Form f_1 is assumed to be the receptor of the message. All other forms are its arguments. A *state* is a pair $s = Env \times Store$, where $Env = (\Phi, \varepsilon)$, and $Store = (S_\varepsilon, S_\Phi)$. $\Phi : F_r \times F_m \to H$ expands methods (associated with the class of an instance representative), $\varepsilon : F_r \to F_i$ maps a representative into a particular instance, $S_\varepsilon : F_i \to 2^{Values}$ maps instance forms to sets of values defined in the underlined semantic model (called "Partial instances", or PINS), and $S_\Phi : H_m \to Rels(2^{Values})$ maps expanded methods to sets of relations over PINS in the underlined semantic model. We say that configuration (M,s) *reduces* to configuration (M',s') iff $((M,s),(M',s')) \in R$.

We leave to section 4 the definition of functions I_C, I_m, I_i, I_r mapping forms to expanded views in the underlined visual formalism of *Higraphs*. We use these translations to specify formally the visual syntax and semantics of a program in *Cordial*. In the next section we give a general view of visual programming in *Cordial* through the discussion of short examples. In so doing we choose for simplicity an intermediate visual layer restricting somewhat the variety of forms to represent entities of a program.

3. Visual programming in *Cordial*

We give in this section a general outlook of the visual components of a program in *Cordial*. Here we only discuss the use of basic visual elements in *Cordial* with the aim of providing a general flavor of what visual programming means in the context of this language. A more complete description can be found in [QRT97]. As was mentioned above, the basic elements of a program in *Cordial* are those of OO programming, such as *classes*, *objects*, and *methods* (understood as relations). A program is a set of classes. Its execution begins with the invocation of a method in one of these classes. As usual, the definition of a class includes the specification of a set of methods. Figure 3-1.a shows the class *complex* number and its expanded view (a "double click" action is always associated with mapping a form to its expanded view). Attributes and methods (formally indistinguishable) are represented by boxes containing a signature.

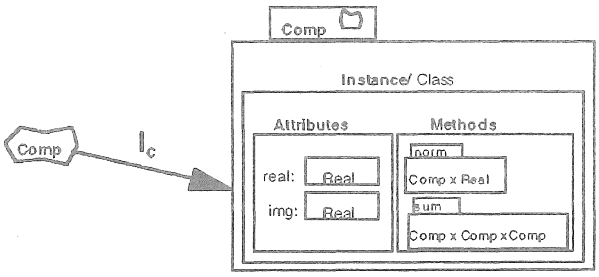

Figure 3-1.a Expanded view of Class

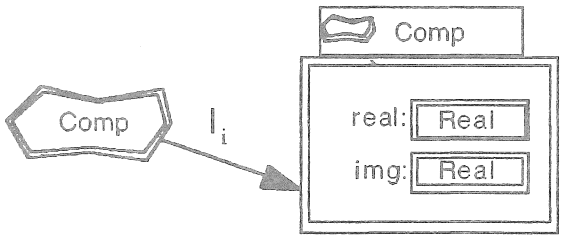

Figure 3-1.b Expanded view of object

Signatures of methods are Cartesian products since they are interpreted as relations. Objects are represented much the same way as classes (see Figure 3-1.b). Visually they are distinguished from their class by a double line contour. Their expanded view shows only the current values (if any) of attributes. Bear in mind, though, that instance representatives relate to *sets* of objects refined monotonically by constraints, so the notion of *a value* for an attribute component is not, formally, well defined. The expanded view of a method is represented by a contour labeled with the name (or form) of the method and its class name (or form). A method body (see Figure 3-2) contains instance

representative forms and method forms. Instance representative forms linked by a line to the method's contour represent parameters. A line labeled *self* identifies, as usual, the receptor of the message triggering the method defined by the contour. The ternary method *sum* in Figure 3-2 constraints an instance of a complex (say the one linked to the contour by a line labeled "2") to be the sum of two other complex numbers. The invocation of this method does not necessarily assume that all values for the attributes of the instances involved should be uniquely defined. The particular relation associated with the method is interpreted in the underlined semantic model which may impose restrictions on the set of values for instances in a method invocation. Double lines identify that instance representative form receiving

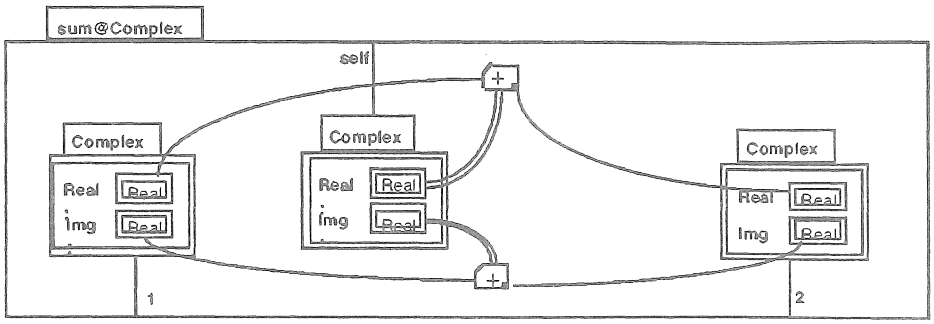

Figure 3-2. Addition of two complex numbers

the message. As mentioned before, methods may contain conditional computational paths (Figure 3-3).

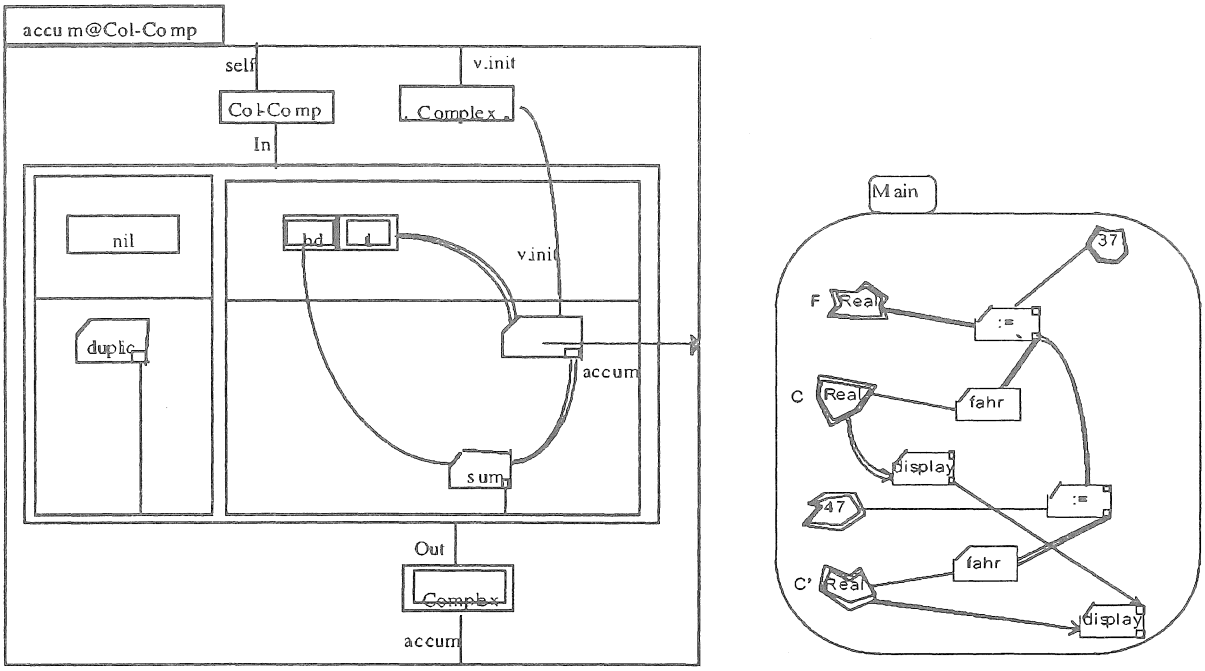

Figure 3-3. (a) Conditionals. (b) Implicit ask in method invocation

Conditionals are represented by two or more contours inside another contour, as shown in Figure 3-3(a). The semantics of conditionals is related to the notion of *Ask agents* in the concurrent constraint programming paradigm [Sar93]. Figure 3-3(a) depicts a method that implements a reduction operation on a collection of objects. The method uses an operation (addition, in this case) and a given initial value to accumulate the elements of a collection of complex numbers. The *guards* in the conditional ask for a particular form (either *nil* or composite *head-tail*) of the argument *self* (the collection) to be deduced from the information implicit in all relations that have been imposed. The *guards* represent the predicates Object(self) = nil and $\exists_{hd,tl}$ | Object(self) = cons(hd,tl). *Methods*

below the straight-line in a subcontour of a conditional are considered only if the deduction of the *guard* succeeds. The body of the right subcontour contains a recursive invocation of the method, indicated by a thick arrow pointing to the external contour. Instance representatives does not always have to be explicitly defined. In Figure 3-3(a) the third argument to the recursive invocation (labeled *accum*) is not drawn but is the implicit receptor of the message *sum*. Lines connecting arguments of *mail Boxes* (i.e. method invocations) to contours containing them represent mutually exclusive alternatives for the line marked "Out" in the figure. Labels in lines serve to order the arguments in a method invocation.

As was mentioned before, *Cordial* is a concurrent language. There is no implicit ordering in method invocations since messages define constraints on the arguments. In certain situations the need might arise to force an explicit ordering. Figure 3-3(b) shows a method *main* invoking two Fahrenheit-to-Celsius conversion (*fahr*) methods. Arrows instead of lines in messages implement an *implicit-ask* (Smolka[]) operation over the instance representative form linked to the tail of the arrow. Before a method is invoked (*display* in Figure 3-4), it should be deducible that all constraints imposed on the form have reduced the set of instances it represents to a singleton set (i.e. the form represents a unique value). In this way, the upper *display* method in Figure 3-4 effectively "waits" until the conversion from 37 degrees Fahrenheit to Celsius has completed. Assignment is considered a relation, much like in [Smo94]. The use of arrows to represent *implicit-ask* operations in *Cordial* is really an abbreviation. The same effect could be achieved using conditionals. In

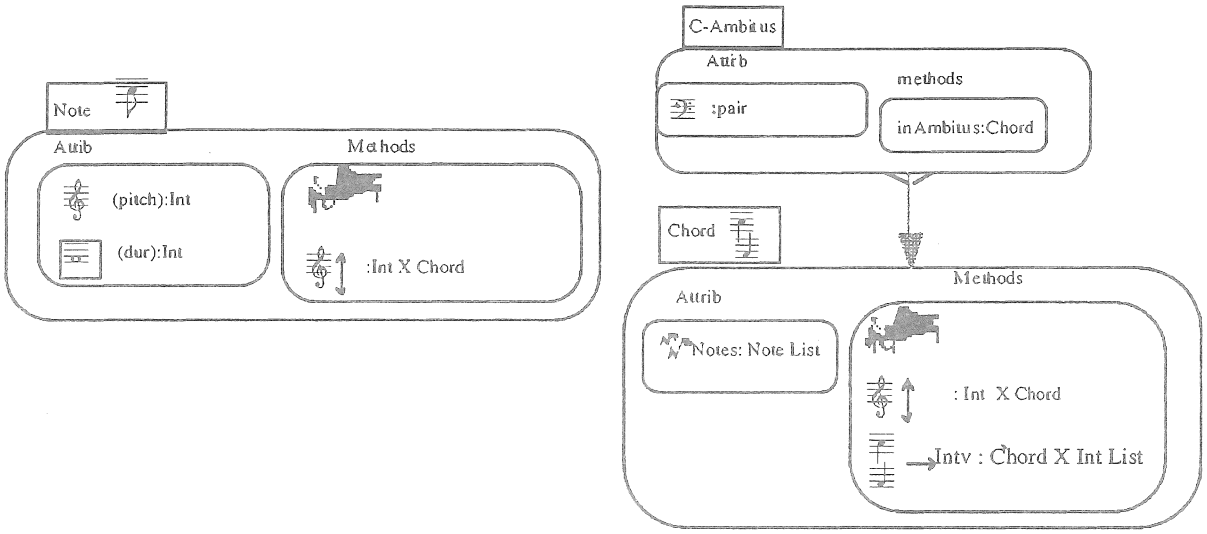

Figure 3-4. Representation of inheritance.

Figures 3-4 and 3-5 we show a somewhat more elaborate example. Three classes are defined, *Note*, *Chord* and *C-ambitus*. The latter inherits from *Chord* as is shown by the down pointing arrow. Methods are identified by icons. The piano player, for example, represents the method capable of playing an object. A chord contains a list of notes. *C-ambitus* is a chord for which every note must be in a given pitch register, that is, its pitch must be less than and greater than two given integers (the elements of the pair (X,Y) in figure 3-5). Method *Inambitus* establishes the constraint that insures this is so. It invokes a method of the same name for the class *List* which uses a recursive call within a conditional to set constraints ">=" and "<=" for the pitch of each note in the list. In the next section we describe the translation of *Cordial* programs into the underlined visual formalism.

4. Interpreting forms in Higraphs

Giving formal syntax and semantics to visual programs is an active research topic. Different extensions of the traditional grammatical formalisms for linear languages have been proposed, leading to a variety of new types of grammars, notably *graph* [RS95], *relational* [CGNTT91] and *constraint multiset* [Mar94]. General rewrite systems has also been tried as in [NK91].

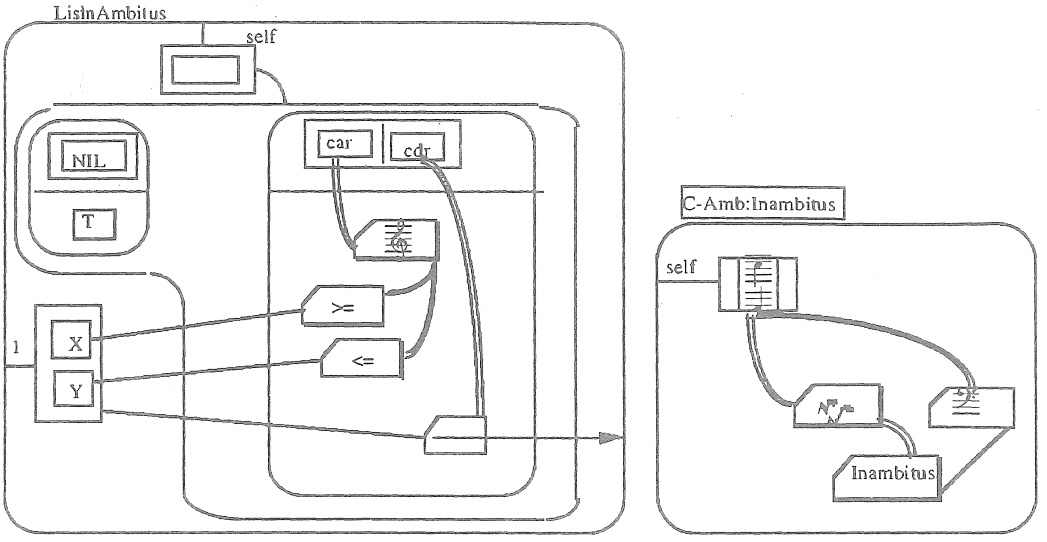

Figure 3-5 Implementation of the *ambitus* constraint

A hierarchical classification of syntax formalisms for visual languages can be found in [Mey91]. A logical approach to specification of syntax and semantics has been proposed recently, and its convenience evaluated in the definition of *Pictorial Janus* [Haa96], [GC96]. We follow a different path. In our view, the syntax of *Cordial* is formed out of graphical elements built on a layer above *Higraphs*. This layer should be high enough to allow ease of programming but also low enough to be directly translatable to it. In this way syntax and static semantics of *Cordial* programs are expressed by this translation. *Higraphs* provide a clearly defined visual formalism useful in a variety of applications. It derives its expressive power from an integration of two well known mathematical formalisms, *Venn diagrams* and *graphs*. The integration can neatly express structural composition of entities (via set theoretic operations) as well as relations. Sets are represented by closed contours (called *blobs*). A dashed line partitioning a *blob* defines a Cartesian product. Relations are depicted by arrows or lines linking blobs together. Relations can be labeled inside a diamond. Thus *Higraphs* represent graphs whose nodes can have structure. Figure 4-1 represents sets X,Y,A,B,C,D,E,F satisfying $Y,E,F \subseteq X$, $D \subseteq E$, $Y = (A \cup B) \times (C \cup D)$ (in *Higraphs* product is unordered) and relation $R \subseteq C \times F$. We give formal visual syntax and semantics to *Cordial* by defining a translation of a program into a *Higraph*. We will do this by a composition of expansions in which method forms are first expanded into its associated contour and this contour together with the forms contained in it then (recursively) expanded into a *Higraph*. In Figure 4-2 method form m is expanded into a contour showing two method forms m_1, m_2 and four instance representatives ir_1, ir_2, ir_3, ir_4. Method m can be further expanded into a *Higraph* as shown in Figure 4-2. *Blobs* b_1, b_2 are sets of instances satisfying the constraints imposed by messages involving m_1, m_2 in Figure 4-2. More precisely,

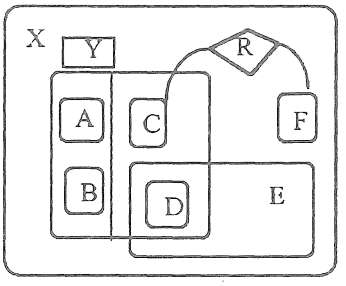

Figure 4-1 A *Higraph*

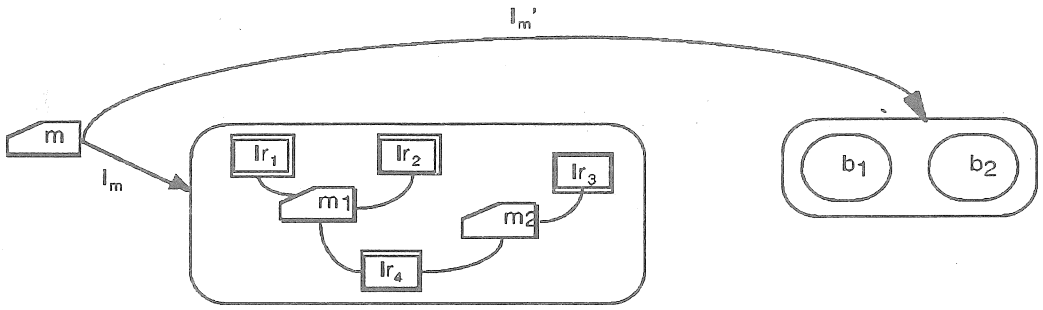

Figure 4-2 Method expansion

Definition 4-1

Let M_I be a method form appearing within the contour of the expanded view of method M. Let $f_{r_1}^{(M)},\cdots f_{r_p}^{(M)} \in I_m(M)$, $f_{r_1}^{(M_1)},\cdots f_{r_q}^{(M_1)} \in I_m(M_1)$ be the instance representatives appearing in the expanded views of M and M_I, respectively. Let $A:\{1,2,\cdots,q\} \rightarrow \{1,2,\cdots,p\}$ be an index mapping function such that $A(j) = i$ just when $f_{r_j}^{(M_1)}$ is linked to argument $f_{r_i}^{(M)}$ in the invocation of method M_I within the expanded view of M. Given a program P, the *blob set translation* of M_I in M is defined as $BST_M(M_1, A) =$

$$\left\{(x_1, x_2, \cdots, x_p) \mid (x_1, \cdots, x_p) \in MCS_P(M) \wedge \left(\exists_{y_1,\cdots,y_q} : (y_1, \cdots, y_q) \in MSS_P(M_1) \wedge (\forall_{j \in Dom(A)} : x_{A(j)} = y_j)\right)\right\}$$

Translation of a method form in a message into a *blob* thus entails two operations: first mapping the arguments of the message to instance representatives within the expanded view of the method, and then using the method's solution space to refine the computational space of the expanded view in which the method appears. The following is an example of this process.

Example. In Figure 4-2, let $MSS_P(M_1) = \{(f_1, f_5, f_7, f_4), (f_2, f_9, f_3, f_1)\}$, $A = \{(1,3),(2,1),(3,4)\}$. Then $BST_M(M_1, A) = \{(f_5, x, f_1, f_7)\} \cup \{(f_9, x, f_2, f_3)\}$, for all $x \in Ins(Classof(f_{r2}^{(m)}))$.

Higraphs can represent set containment but not set membership. In the next section we extend *Higraphs* by giving a visual representation to unitary sets. Since methods are translated into *blobs* (i.e. sets) we define rules for constructing its elements (i.e. the tuples in $MSS_p(M)$). In section 4-1 we define precisely an interpretation function υ mapping methods to elements in its solution space. This allows us to fully translate *Cordial* programs into the visual formalism.

4-1. Semantics of *Cordial* in *Higraphs*.

As was mentioned previously, a program in *Cordial* is a set of class forms and a message (method invocation). Each form is mapped by I_C into an expanded view (see section 2) comprising a set of instances and a set of method forms. The set of instances (a *blob*, thus) is not further treated here since it is assumed to be interpreted as a set of values in the underlined semantic model. The set of method forms is defined by first expanding each method with I_m and then translating the result into a *Higraph*. The translation into the underlined visual formalism can thus be seen as the composition of two functions $\upsilon \circ I$, one expanding the appropriate form (class or method) and the other translating the result into a *Higraph*. Class *blobs* thus contain singleton sets of method *blobs* and method *blobs* contain *blob set translations* (see definition 4-1) of messages appearing in them. The specific elements in all these *blobs* are given by the recursively defined function υ. *Blob set translations* of primitive messages are computed in the underlined semantic model. We begin by extending the original definition of a *Higraph* [Har88] to include the different types of *blobs* in *Cordial*.

A problem with the standard definition of *Higraphs* is that membership relations cannot be represented. We extend *Higraphs* by adding singleton sets represented as atomic *blobs* (i.e. those not containing sub *blobs*). We depict atomic *blobs* by closed contours with double lines labeled with the form representing their unique element (see figure 4-3).

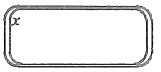

Figure 4-3. A unitary *blob*.

Definition 4-2

A *Higraph* $H = \{B_u, B_c, B_b, B_d, B_k, B_m, \upsilon\}$ comprises unitary *blobs* (B_u), *blobs* for *classes* (B_c), *method body* (B_b), *disjunctions* (conditionals, B_d), *clauses* (B_k), and *messages*. (B_m). s is the sub-*blob* function mapping each *blob* to the set of its subsets. $\sigma : \{B_c, B_b, B_d, B_k, B_m\} \rightarrow 2^{\{B_c, B_b, B_d, B_k, B_m\}}$

A model for H is a pair $M = (CS, \upsilon)$ where CS (computational space) is the set of tuples representing all possible values for each instance representative and $\upsilon : \{B_c, B_b, B_d, B_k, B_m\} \rightarrow 2^{CS}$ maps *blobs* to sets of values. υ is defined recursively as follows:

0. If $x \in B_u$, $\upsilon(x) = \upsilon(y)$, where $y \in x$

1. If $x \in B_c$, $\upsilon(x) = \bigcup_{y \in x} \{\upsilon(I_m(y))\}$.

2. If $x \in B_b$, $\upsilon(x) = \bigcap_{y \in x} \rho(y)$, where $\rho(y) = \begin{cases} \upsilon(y) \text{ if } y \in B_d \\ \upsilon(I_m(y)) \text{ otherwise} \end{cases}$

3. If $x \in B_d$, $\upsilon(x) = \bigcap_{y \in x} \upsilon(y)$

4. If $x \in B_k$, $v(x) = \begin{cases} \upsilon(x_b), & \text{if } \upsilon(x_g) \supseteq S \\ S, & \text{if } \upsilon(x_g) \subset S \end{cases}$, where x_g, x_b are the *guard* and *body* , respectively, of the clause and

$S = \left(\bigcap_{z \in b_M, z \neq x} \upsilon(z) \right) \cap MCS(M)$, where b_M is the *blob* of the method M in which the clause x is defined.

5. if $x \in B_m$, $\upsilon(x) = \begin{cases} V \subseteq CS, \text{if primitive}(x) \\ \bigcap_{y \in I_m(x)} BST_x(y, A_x) \end{cases}$

Function υ thus defines sets of tuples of instances that can *potentially* belong to the relations defined by methods. In the dynamic semantic model, a particular sequence of method invocations define refinements of these sets. The interpretation is defined recursively in the structure of a visual program. A class *blob* is thus interpreted as the set containing each interpretation of the method forms appearing in the class. A method body *blob* (i.e. the result of applying I_m to a method form) is interpreted as the intersection of the sets in the interpretations of all forms (*messages* or *conditionals*) appearing in it. Disjunctions (i.e. *conditionals*) are also interpreted as intersections of the sets interpreting all clauses in the disjunction. The interpretation of a clause form (i.e. each alternative in a disjunction) is essentially a set theoretic definition of the *Ask* operation in concurrent constraint languages: the interpretation depends on whether the guard is or is not deducible from information provided by other messages in the same method (this is the role of *S*). Finally, interpretation of message forms (*mailboxes* in the examples) is either given by the underlined semantic model or computed as the intersection of *blob set translations* of forms in the expanded view of the method invoked by the message.

5. Conclusions and future work.

We have introduced *Cordial*, a visual language integrating OO and constraint programming. We characterized its visual model as a hierarchy of graphical layers where icons representing classes, methods and instances can be expanded to show their components. We argued that this feature allows the program's appearance to be tailored to fit different types of users. Choosing an intermediate layer we illustrated programming in *Cordial* as message passing between objects, interpreted as relations. We argued that the semantics given to arrows in *Cordial* completed the representation of *Ask* and *Tell* operations of CCP languages. Finally, we gave a set theoretic static semantics of

visual programs by a formal translation into an extension of the *Higraph* formalism. This translation defines methods as *blobs* containing tuples of instances defined by set theoretic operations over values defined in an underlined semantic model. *Cordial* is part of a project aimed at defining powerful tools for musical composition. It inherits from ideas in Patchwork and *Niobé* [Rue94]. An implementation of the visual model in JAVA is currently under way. The strategy for constructing a semantic model is to define a concurrent calculus integrating objects and constraints. An extension with constraints (but not objects) is given in [VDR97]. Adding objects to this calculus, constructing from ideas in Tyco [Vas94] is under way. We are currently working on more declarative syntax specification of *Cordial* using descriptive logic [Haa96]. We plan to test the usability of the language by constructing a musical orchestration system in the near future.

6. References

[BA94] M. Burnett and A. Ambler. Interactive Visual Data Abstraction in a Declarative Visual Programming Language. *Journal of Visual Lang. and Comp.* 5(1), March 1994, 29-60.

[CGNTT91] C. Crimi, A. Guercio, G. Nota, G. Pacini, G. Tortora, M. Tucci. Relation grammars and their application to multi-dimensional languages. *JVLC*, vol.2, n.4, pp.333-346, 1991.

[GC96] J. M. Gooday and A. G. Cohn .Visual Language Syntax and Semantics: A Spatial Logic Approach . *Proc. of the Internat. Workshop on Theory of Visual Lang.*, Italy, May 1996.

[Haa96] V. Haarslev . A Fully Formalized Theory for Describing Visual Notations. 1996

[Har88] D. Harel. On Visual Formalisms. In *CACM VL* 31, May 1988, pages 514-530.

[HM94] M. Henz and M. Müller. Programming in Oz. . In: *DFKI Documentation series*, 1994.

[HTI90] M. Hirawaka, M. Tanaka, and T. Ichiwaka. An Iconic Programming System: Hi-Visual. *IEEE Trans. Software Eng.* Oct 1990, pp. 71-80

[Hüb96] Hübscher, Roland, Composing Complex Behavior from Simple Visual Descriptions. In *1996 IEEE Symposium on Visual Languages*, Boulder,CO, Sept. 1996. pp. 88-94.

[KS90] M. K. Kahn and V. A. Saraswat. Complete Visualizationn of Concurrent Programs and their Executions. In *1990 IEEE Workshop on Visual Languages*, pages 7-15

[Mar94] K. Marriot. Constraint multiset grammars. In *IEEE Symposium on Vis. Languages*, 1994.

[Mey91] B. A. Meyer. Taxonomies of visual programming and programming visualizations. *IEEE Workshop on Visual Languages*, pp 56-61, 1991.

[NK91] M. Najork and S. Kaplan. The Cube Language. In 1991 *IEEE Workshop on Vis. Lang.*, pp. 215-220.

[QRT97] L. Quesada, C. Rueda, G. Tamura. Programación Visual en Cordial. *ReporteTécnico* AV-97-03, Grupo AVISPA, Universidad javeriana de Cali, 1997.

[RS95] M. Rekers and A. Schürr. A Graph Grammar Approach to Graphical Parsing. In *VL'95*. pp. 195--202.

[Rue94] C. Rueda. A Visual Programming Enviroment for Constraint-Based Musical Composition. *Proceedings, XIV Congresso da Sociedade Brasileira de Computacao. Caxambu, Brasil, 1994.*

[Sar93] V. A. Saraswat. Concurrent Constraint Programming. MIT Press, Cambr. MA, 1993

[SS94] C. Schulte and G. Smolka. Encapsulated search and constraint programming in Oz. In: *Second Workshop on Principles and Practice of Constraint Programming*, 1994.

[Smo94] G. Smolka. A foundation forhigher - order concurrent constraint programming. *In Lecture Notes in Computer Science*, vol. 845, pp. 50-72. München, Sept 1994. Springer-Verlag.

[Smo94] G. Smolka. The definition of Kernel Oz. In: DFKI Oz documentation series, 1994.

[Sco95] S. Steinman and K. G. Carver. Visual Programming with Prograph CPX. 1995

[Vas94] V. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *Proceedings of 8th European Conference on Object-Oriented Programming* (ECOOP'94).

[VDR97] F. Valencia, J. F. Diaz, and C. Rueda. The π^+-calculus: Uses and Behavioral Equivalence. Submitted to *DSL97*, Santa Barbara, Calif.,1997.