
Constraints, 6, 21–52, 2001

Integrating Constraints and Concurrent Objects in Musical Applications: A Calculus and its Visual Language

CAMILO RUEDA* crueda@atlas.ujavcali.edu.co
Department of Computer Science, Pontificia Universidad Javeriana, cl 18 118-250, via Pance, Cali, Colombia

GLORIA ALVAREZ galvarez@atlas.ujavcali.edu.co
Department of Computer Science, Pontificia Universidad Javeriana, cl 18 118-250, via Pance, Cali, Colombia

LUIS O. QUESADA lquesada@atlas.ujavcali.edu.co
Department of Computer Science, Pontificia Universidad Javeriana, cl 18 118-250, via Pance, Cali, Colombia

GABRIEL TAMURA gtamura@atlas.ujavcali.edu.co
Department of Computer Science, Pontificia Universidad Javeriana, cl 18 118-250, via Pance, Cali, Colombia

FRANK VALENCIA fvalenci@brics.dk
Department of Computer Science, Pontificia Universidad Javeriana, cl 18 118-250, via Pance, Cali, Colombia

JUAN FRANCISCO DÍAZ jdiaz@borabora.univalle.edu.co
Department of Computer Science, Universidad del Valle, ciudad universitaria Meléndez, Cali, Colombia

GERARD ASSAYAG assayag@ircam.fr
IRCAM, 1 pl Igor Stravinski, 75004 Paris, France

Abstract. We propose *PiCO*, a calculus integrating concurrent objects and constraints, as a base for music composition tools. In contrast with calculi such as [5], [9] or TyCO [16], both constraints and objects are primitive notions in *PiCO*. In *PiCO* a base object model is extended with constraints by orthogonally adding the notion of *constraint system* found in the ρ-calculus [12]. Concurrent processes make use of a constraint store to synchronize communications either via the *ask* and *tell* operations of the constraint model or the standard message-passing mechanism of the object model. A message delegation mechanism built into the calculus allows encoding of general forms of inheritance. This paper includes encodings in *PiCO* of the concepts of class and sub-class. These allow us to represent complex partially defined objects such as musical structures in a compact way. We illustrate the transparent interaction of constraints and objects by a musical example involving harmonic and temporal relations. The relationship between *Cordial*, a visual language for music composition applications, and its underlying model *PiCO* is described.

Keywords: concurrent programming, constraint programming, concurrent constraint objects, TyCO, *PiCO*, formal calculi, mobile processes, visual language, computer aided music composition

1. Introduction

– Why do we want Concurrent Objects with Constraints ?

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

Our objective is to develop computational models suitable for constructing music composition tools. Musical objects can take a wide variety of forms depending on the particular dimensions they belong to. In a harmonic (vertical) dimension, objects such as chords contain notes that can be constrained to lay within defined zones (or *registers*), to belong to defined *textures* (e.g. patterns of harmonic intervals), etc. In a temporal (horizontal) dimension, sequences of chords or notes can be defined to be positioned in such a way that they form selected rhythmic patterns. There exists also "diagonal" dimensions such as *dynamics*, where notes of chords can follow complex amplitude evolutions or such as *melody*, where patterns of distances relate chord notes in distinct temporal positions. Relationships among sets of objects in several dimensions define musical structures. These in turn can be regarded as higher level objects which are also amenable to different kinds of musical transformations. Being able to express the whole complexity of constructing a network of structures satisfying the musical intentions of a composer is a big challenge for any computer programming model.

In recently proposed computer aided musical composition systems such as *Situation* [3] constraints and *Common Lisp* objects can be used to define complex musical structures. In the same spirit, but more closely integrated to the underlying Smalltalk language, *Backtalk* [7] provides a framework for handling constraint satisfaction within an object environment. Both systems have been successfully used in practical musical settings. In both applications, however, the constraint engine is a black box barely accessible to the user. Moreover, communicating data structures back and forth between the constraint and object models is often awkward. In fact, objects containing partial information and "standard" instantiated objects are not really amenable to the same kind of computational treatment. In musical applications this lack of communication potential can be specially troublesome since the approach of the composer involves for the most part constant refinement and modification of compositional models based on the acoustical result of partial implementations.

We think that the development of computational models and of tools for computer aided music composition should go hand in hand to benefit from insights at the user level while maintaining a coherent formal base. Defining a uniform model integrating constraints and objects can be of great help to construct higher level musical applications that provide the musician with flexible ways of interaction. In [15] the π-calculus was extended with the notion of constraint. In this paper we consider the addition of objects and messages synchronized by constraints.

Several concurrent objects calculi have been proposed recently [17], [16], [1]. In these models the interactions of concurrent processes (or objects) are synchronized essentially through one of two mechanisms: the "use of a channel" and "message-passing". In *TyCO*, for instance, an object $a \triangleright [l : (\widetilde{y})P]$ can be seen as a process P which is suspended until some message selecting a method labeled l is sent to an object located at a or, more generally, located at some x such that $x = a$ is provable.

On the other hand, constraints can also be used to define a rich set of possible concurrent processes interactions, as has been shown in the *cc* model [11]. The basic operations *ask* and *tell* allow processes to define complex synchronization schemes through the use of common process variables. In the *cc* language *Oz* [5], first-class procedures and first-class cells are used to simulate objects within a concurrent constraint setting. Objects are thus

not primitive. In fact, the constraint paradigm is the only underlying model of interaction. The powerful constraints calculus of *Oz* allows other programming models (functional, objects) to coexist through suitable syntactic encodings. More recently, the *MCC* calculus [10] considers extensions of the *cc* model to account for reliable communication of agents having their own local constraints store and shows how to encode process mobility through a mechanism of "abstract" constraint messaging.

We take a different approach. We want to maintain as much as possible the independence of the object and constraint models at the calculus level. Firstly, we think each of them accurately models an approach composers usually follow in constructing musical structures. Secondly, we would like our tool to be easily adaptable to different notions of object and different constraint systems.

In addition to including both constraints and concurrent objects as primitive notions at the calculus level, we propose two features that, to our knowledge, have not been considered before. One is the notion of objects guarded by or "located" in a constraint. The other is the concept of message *delegation*. We borrow the insight in [11] of considering constraints as defining (polyadic) *types* to explain these features.

In the π-calculus a communication can take place between a reader and a writer process just when they agree on a communication channel (denoted by a *name*). We may regard each name as a distinct type and say that processes communicate just when sender and receiver are of exactly the same (singleton) *communication* type. Similarly, in the ρ-calculus [12] and other *cc* calculi, sender and receiver agents are "located" in variables. They communicate just when it can be inferred from constraints information that the two variables are equal. That is, just when they have exactly the same (not necessarily singleton-set) communication type.

In *PiCO* each receiver object defines explicitly its communication type by "locating" itself in a constraint. This constraint (let us call it $\phi(\textbf{sender})$) is parameterized in a sender location. As a type this constraint defines the set of possible requesters of services from the receiver. Given a constraint store S, the type of a receiver object located at a constraint $\phi(\textbf{sender})$ is the set of all sender locations v such that every valuation consistent with S is also consistent with $\phi(v)$. Asserting constraints *increases* the set of such v's.

Message senders "locate" themselves in a variable (or a name). Constraints on this variable define possible values for it. The communication type of the sender is the set of these values. Now, communication can take place just when it can be inferred from the information supplied by asserted constraints that the sender communication type is a *subtype* of the receiver communication type. Thus a sender can increase the set of potential destinations of a message by "moving" to a subtype (i.e. asserting extra constraints on its location), whereas a receiver can also increase the set of accepted requesters of its services by moving to a *supertype* (i.e. asserting constraints on the variables of its constraint location). Singleton type receivers communicate as π-calculus reader processes. Senders constraining their type to that of their intended receivers communicate as ρ-calculus abstractions.

A *PiCO* receiver object may determine a *delegation* communication type for those messages coming from senders of an acceptable type but requesting services the object cannot provide. This delegation type is also defined by a constraint. Delegated types must be in the complement of the original sender type since otherwise the delegated service could very

well have been provided directly by some other receiver object. Delegation means moving the type of the original message sender to the delegated type. This mechanism can be used to define different object inheritance schemes.

Consider an example. Let us suppose that we want to define some conditions for the location of an object. For instance $sender \in \{a, b, c\} \triangleright [l : (\widetilde{y})P]$ can be seen as a process P which is suspended until a message to a, b or c is sent. This can also be interpreted as an object *guarded* by a constraint to ensure reception of messages only from the three given locations. Of course, for this simple example, the π-calculus [6] process $a?[\widetilde{y}].P + b?[\widetilde{y}].P + c?[\widetilde{y}].P$, could very well be used for this. Here a non determinate choice (the $+$ symbol) of the same process P reading through a, b or c simulates the "same" occurrence of P receiving from the three locations. However, when we wish to define arbitrary conditions to execute P, to send messages to objects, or to locate objects, we need better ways to express communication. These arbitrary conditions can be naturally expressed by means of constraints.

In fact, object interactions can naturally be modeled in at least two ways. First, by means of concurrent objects whose synchronized execution simulates changes on real objects [6] and second, by using constraints to "change" the object state by refining the partial knowledge one has about the attributes of the object.

These views are complementary. In a musical setting both are typically used. Composers may very well conceive musical processes as evolving according to explicitly defined trajectories or to particular compositional *rules*, or both. In the former, object attributes can naturally be seen as being bound to values whereas in the latter attributes express only their degree of consistency with the partial information implied by the rules.

In sections 2.2 and 2.3 we present the syntax and semantics of *PiCO*. The syntax of *PiCO* adds constraint processes to the standard processes in calculi such as $TyCO$. Additionally, objects include information to identify where they have to delegate those messages they are not able to answer. Constraint processes perform the standard *Ask* and *Tell* operations of *CCP* languages. The semantics is defined operationally following the transition system for the cc-model used in [11]. Section 3 shows how classes and mutable objects carrying partial information on their attributes can be conveniently represented in *PiCO*. The notions of high-order attributes and sub-classing as a form of inheritance by *behavior reuse* and *method overriding* are also shown in that section. Section 4 shows briefly how the semantics for a concurrent constraint visual language can be expressed using *PiCO*. In section 5 we show a somewhat elaborate musical example. We discuss the relation between each of π, ρ and *MCC* calculi and *PiCO* in section 6. Section 7 gives some conclusions.

2. *PiCO* Definition

We start by giving a general definition of a constraint system [11].

2.1. Constraint System

PiCO is parameterized in a *Constraint System*. For our purposes it will suffice to base the notion of constraint system on first-order Predicative Logic, as it was done in [13], [12].[1]

A Constraint System consists of [13], [12]:

– A signature. That is, a set of functions, constants and predicate symbols with equality, including a distinguished infinite set, $\mathcal{N}$, of constants called *names* denoted $a, b, \ldots, u$. Other constants, called *values*, are written $v_1, v_2, \ldots$. They are regarded as primitive objects in the calculus that can be used as message locations.

– A consistent theory Δ (a set of sentences over the signature having a model) satisfying the condition:

  - if a, b are distinct names, then it can be inferred that $a = b$ does not hold. Similarly for distinct values.

Often Δ will be given as the set of all sentences valid in a certain structure (e.g. the structure of finite trees, integers, or rational numbers). Given a constraint system, let symbols $\phi, \psi, \ldots$ denote first-order formulae in the signature, henceforth called *constraints*. A fundamental notion of constraint systems is *entailment*. We say that formula ϕ *entails* formula ψ in a theory Δ, written $\phi \vdash_\Delta \psi$, just when the implication $\phi \rightarrow \psi$ is true in all models of Δ. Entailment is important because it adds the capability of handling disjunctive information. We say that ϕ is *equivalent* to ψ in Δ, written $\phi \models|_\Delta \psi$, iff $\phi \vdash_\Delta \psi$ and $\psi \vdash_\Delta \phi$. We say that ϕ is *satisfiable* in Δ iff $\phi \nvdash_\Delta \bot$. We use $\bot$ for the constraint that is always false and $\top$ for the constraint that is always true. A particular constraint system must, of course, have a decidable entailment relation. When no confusion arises we drop the subscript from $\vdash_\Delta$.

As is usual, we will use infinitely many $x, y, \ldots \in \mathcal{V}$ to denote logical variables designating some fixed but unknown element in the domain under consideration. The sets $fv(\phi) \subset \mathcal{V}$ and $bv(\phi) \subset \mathcal{V}$ denote the sets of free and bound variables in ϕ, respectively. Finally, $fn(\phi) \subset \mathcal{N}$ is the set of names appearing in ϕ.

As we said before, most processes act relative to a *store*. A store is defined in terms of the underlying constraint system. It is simply a formula representing the accumulated information supplied by asserted constraints. For our purposes it will suffice to consider the store as a conjunction of constraints. We write $S = \top$ for the "empty" store. A store S is said *unsatisfiable* when it entails *false* (i.e. when $S \vdash_\Delta \bot$).

2.2. Syntax

The syntax of *PiCO* is given in Table 1. There are three basic processes: *Messages*, *Objects* and *Constraints*.

We describe next the calculus informally. In what follows, $\widetilde{t}$ denotes a sequence $t_1, \ldots t_k$, of length $|\widetilde{t}| = k$ whose elements belong to some given syntactic category.

The null process 0 is the process doing nothing. A process $(\phi_{\mathbf{sender}}, \delta_{\mathbf{forward}}) \triangleright M$ represents an object. The purpose of an object is to respond to messages sent by other processes. The object answers a message by replacing itself with another process, or *method*. The message contains information to single out the replacement process from those available in the object method collection M. Messages are sent to objects from other processes.

Table 1. PiCO syntax.

Normal Processes: N	::=	O	Inaction or null process
	|	$I \triangleleft m$ **then** P	Message sent by I
	|	$(\phi_{\mathbf{sender}}, \delta_{\mathbf{forward}}) \triangleright M$	Object with delegation condition $\delta_{\mathbf{forward}}$ guarded by constraint ϕ
Constraint processes: R	::=	**tell** ϕ **then** P	*tell* process
	|	**ask** ϕ **then** P	*ask* process
Processes: P, Q	::=	**local** x **in** P	New variables x in P
	|	**local** a **in** P	New name a in P
	|	N	Normal process
	|	$P \mid Q$	Composition
	|	**clone** P	Replicated process
	|	R	Constraint process
Object identifiers: I, J, K	::=	a, l	Names
	|	v	Value
	|	x	Variable
Collection of Methods M	::=	$[l_1\colon (\widetilde{x_1})P_1 \& \ldots \& l_m\colon (\widetilde{x_m})P_m]$	
Messages m	::=	$l\colon [\widetilde{I}]$	

Message-sending processes are *located*. An object determines by means of constraints which message-sending processes it is willing to service. Constraint $\phi_{\mathbf{sender}}$, the *guard*, must be satisfied by any location (represented by the special variable **sender**) attempting to send messages to the object. For example, object $(\mathbf{sender} \in \{a, b\}, \mathbf{forward} \in \{b, c\}) \triangleright M$ accepts messages from $a \triangleleft m[\,]$ **then** Q since $a \in \{a, b\}$ is true. Constraint $\delta_{\mathbf{forward}}$, the *delegation condition*, represents the guard of the process the messages should be delegated to when appropriate methods for them do not exist in M. Delegation involves creating a new location, say J, and forwarding the same message m from this location. In the above example, delegation would mean that the original process $a \triangleleft m[\,]$ **then** Q is replaced by the same process located at J, i.e. $J \triangleleft m[\,]$ **then** Q. Before delegation, location J is enforced to satisfy the delegation constraint δ so that the delegated object has a chance to accept message m. That is, constraint $J \in \{b, c\}$ is asserted before replacing the original sender $a \triangleleft m[\,]$ **then** Q by $J \triangleleft m[\,]$ **then** Q. Form M in objects represents a collection of methods. Methods $(\widetilde{x_1})P_1 \ldots (\widetilde{x_m})P_m$ are labeled by a set of pairwise distinct labels (names) $Labels(M) = \{l_1, \ldots, l_m\}$.

Processes of the form $(\phi_{\mathbf{sender}}, \phi_{\mathbf{forward}}) \triangleright M$ represent objects not allowing delegation.[2] For simplicity, they will be abbreviated as $\phi_{\mathbf{sender}} \triangleright M$. We also write $(I, J) \triangleright M$, an object "located" at I delegating to an object "located" at J, as a shorthand for $(\mathbf{sender} = I, \mathbf{forward} = J) \triangleright M$. Similarly, $I \triangleright M$ is a shorthand for $(\mathbf{sender} = I, \mathbf{forward} = I) \triangleright M$ and can be thought as an object located at I. This equational guard encodes the usual notion of objects referenced by identifiers.

In a method $l : (\widetilde{x})P$, $\widetilde{x}$ represents the formal parameters and P the body of the method. *Names, variables* and *primitive values* can be used as object identifiers.

A process $X \triangleleft l : [\widetilde{J}]$ **then** P can thus be thought of as a message to a *target* object accepting X with contents or information $\widetilde{J}$. We also allow messages to have a continuation P. Label l is used to select the corresponding method in the target object. Intuitively, the result of the interaction between a message and the target object is the body of the selected method with the formal arguments replaced by the respective actual arguments in the contents of the message, if the requested method exists (i.e. if there exists a method labeled by l and the number of actual and formal arguments match). Otherwise the message is delegated, as explained above.

The process **local** a **in** P restricts the use of the name a to P. Another way to describe this is that **local** a **in** P declares a new unique name a, distinct from all external names, to be used in P. Similarly, **local** x **in** P (new process) declares a new (logical) variable x, distinct from all external variables in P.

Process composition $P \mid Q$ denotes the concurrent execution of processes P and Q. Process **clone** P (*replication*) means $P \mid P \cdots$ (as many copies as needed). A common instance of replication is **clone** $(I, J) \triangleright M$, an object which can be reproduced when a requester communicates via I. Replication is often used for encoding recursive process definitions (see [6]).

Finally, the behavior of constraint processes depends on a global *store*. A store contains information supplied by constraints. The store is used in *PiCO* to control all potential communications. The *tell* process **tell** ϕ **then** P means "Add ϕ to the store and then activate P." Thus, *tell* processes are used to influence the behavior of other processes. The *ask* process **ask** ϕ **then** P means "Activate P if constraint ϕ is a logical consequence of the information in the store or destroy P if $\neg\phi$ is a logical consequence of the information in the store. Otherwise, suspend **ask** ϕ **then** P until the store contains enough information to run it."

In what follows, we write **local** $vI_1, vI_2, \ldots, vI_n$ (vI_i name or variable) instead of **local** vI_1 **in local** vI_2 **in** $\ldots$ **local** vI_n and we omit **then** O when no confusion arises.

2.3. Operational Semantics

Binding operators in *PiCO* are as in [6]: The binding operator for names, **local** a **in** P, declares a new local name a in process P. There are only two binding operators for variables: **local** x **in** P that binds x in P and $(x_1 \ldots x_n)P$ that declares formal parameters $x_1, \ldots, x_n$ in P. A special binding exists for the variable **sender** in $(\phi_{\mathbf{sender}}, \delta_{\mathbf{forward}}) \triangleright M$. Here **sender** is a variable bound by the "guard" constraint of the object and used in the guard ϕ and (possibly) in the methods of M. Variable **sender** can only be used in objects. Similarly, variable **forward** is bound in constraint $\delta_{\mathbf{forward}}$ and can only be used there. So we can define *free names* $fn(P)$, *bound names* $bn(P)$, *free variables* $fv(P)$, *bound variables* $bn(P)$ of a process P in the usual way. The set of variables appearing in P, $v(P)$, is $fv(P) \cup bv(P)$ and similarly the set of names appearing in P, $n(P)$, is $fn(P) \cup bn(P)$.

2.3.1. *Structural Congruence and Equivalence Relation*

We define structural congruence for *PiCO* much in the same way as is done for the π-calculus in [6].

Definition 2.1 (Structural Congruence). Let structural congruence, $\equiv$, be the smallest congruence relation over processes satisfying the following axioms:

- Processes are identical if they only differ by a change of bound variables or bound names ($\alpha - conversion$).
- $(\mathcal{P}/\equiv, |, O)$ is a symmetric monoid, where $\mathcal{P}$ is the set of processes.
- $(\phi_{\mathbf{sender}}, \delta_{\mathbf{forward}}) \triangleright M \equiv (\phi_{\mathbf{sender}}, \delta_{\mathbf{forward}}) \triangleright M'$ if M' is a permutation of M.
- **clone** $P \equiv P \mid$ **clone** P.
- **local** a **in** $O \equiv O$, **local** x **in** $O \equiv O$,
  local a **in local** b **in** $P \equiv$ **local** b **in local** a **in** P,
  local x **in local** y **in** $P \equiv$ **local** y **in local** x **in** P,
  local a **in local** x **in** $P \equiv$ **local** x **in local** a **in** P.
- If $a \notin fn(P)$ then **local** a **in** $(P \mid Q) \equiv P \mid$ **local** a **in** Q.
- If $x \notin fv(P)$ then **local** x **in** $(P \mid Q) \equiv P \mid$ **local** x **in** Q.
- If $\phi \models\!\models_{\Delta} \psi$ and $P \equiv Q$ then
  tell ϕ **then** $P \equiv$ **tell** ψ **then** Q and **ask** ϕ **then** $P \equiv$ **ask** ψ **then** Q

Definition 2.2 (P-equivalence relation). We will say that $\langle P_1; S_1\rangle$ is P-equivalent to $\langle P_2; S_2\rangle$, written $\langle P_1; S_1\rangle \equiv_P \langle P_2; S_2\rangle$, if $P_1 \equiv P_2$, $S_1 \models\!\models_{\Delta} S_2$, $fn(S_1) = fn(S_2)$ and $fv(S_1) = fv(S_2)$. $\equiv_P$ is said to be the P-equivalence relation on configurations.

The behavior of a process P is defined by transitions from an initial configuration $\langle P; \top\rangle$. A transition, $\langle P; S\rangle \longrightarrow \langle P'; S'\rangle$, means that $\langle P; S\rangle$ can be transformed into $\langle P'; S'\rangle$ by a single computational step. For simplicity, we assume that all variables and names are declared in the initial configuration i.e., $fv(P) = fn(P) = \emptyset$. We define transitions on configurations next.

2.3.2. *Reduction Relation*

The reduction relation ,$\longrightarrow$, over configurations is the least relation satisfying the rules appearing in Table 2:

Notice that in rules DECV and DECN the non membership condition can always be met by α-conversion.

COMM describes the result of the interaction between message $I' \triangleleft l\colon [\widetilde{J}]$ **then** Q and object $(\phi_{\mathbf{sender}}, \delta_{\mathbf{forward}}) \triangleright [l : (\widetilde{x})P\&\ldots]$. The store is used to decide whether the object is indeed the target of the message. Process $P\{\widetilde{K}/\widetilde{x'}, I'/\mathbf{sender}\}$ is obtained by first α-

Table 2. Transition system.

$$\text{COMM: } \frac{S\vdash_\Delta \phi[I'/\mathbf{sender}] \qquad |\widetilde{K}|=|\widetilde{x}|}{\left\langle I' \lhd l\colon [\widetilde{K}]\ \mathbf{then}\ Q \mid (\phi_{\mathbf{sender}},\delta_{\mathbf{forward}})\rhd [l\colon (\widetilde{x})P\&...];S\right\rangle \longrightarrow \left\langle Q \mid P\{\widetilde{K}/\widetilde{x'}, I'/\mathbf{sender}\};S\right\rangle}$$

$$\text{DEL: } \frac{\mathrm{S}\vdash_\Delta \phi[\mathrm{I}'/\mathbf{sender}] \quad \mathrm{S}\sqcup\delta[\mathrm{I}'/\mathbf{forward}]\vdash_\Delta \bot \quad \mathrm{l}\notin \mathrm{Labels(M)}}{\left\langle \begin{pmatrix} I' \lhd l\colon [\widetilde{K}]\ \mathbf{then}\ Q \mid \\ (\phi_{\mathbf{sender}},\delta_{\mathbf{forward}})\rhd M \end{pmatrix};\mathrm{S}\right\rangle \longrightarrow \left\langle \begin{pmatrix} \mathbf{local}\ J\ \mathbf{in\ tell}\ \delta[J/\mathbf{forward}] \\ \mathbf{then}(J\lhd l\colon [\widetilde{K}]\ \mathbf{then}\ Q)\mid \\ (\phi_{\mathbf{sender}},\delta_{\mathbf{forward}})\rhd M \end{pmatrix};\mathrm{S}\right\rangle}$$

$$\text{TELL: } \langle \mathbf{tell}\ \phi\ \mathbf{then}\ P;S\rangle \longrightarrow \langle P; S\wedge\phi\rangle$$

$$\text{ASK: } \frac{S\vdash_\Delta \phi}{\left\langle \mathbf{ask}\ \phi\ \mathbf{then}\ P;S\right\rangle \longrightarrow \langle P;S\rangle} \quad , \quad \frac{S\vdash_\Delta \neg\phi}{\left\langle \mathbf{ask}\ \phi\ \mathbf{then}\ P;S\right\rangle \longrightarrow \langle O;S\rangle}$$

$$\text{PAR: } \frac{\langle P;S\rangle \longrightarrow \left\langle P';S'\right\rangle}{\langle Q\mid P;S\rangle \longrightarrow \langle Q\mid P';S'\rangle}$$

$$\text{DEC-V: } \frac{x\notin fv(S),\ \langle P;S\gg\{x\}\rangle \longrightarrow \left\langle P';S'\right\rangle}{\left\langle \mathbf{local}\ x\ \mathbf{in}\ P;S\right\rangle \longrightarrow \langle P';S'\rangle}$$

$$\text{DEC-N: } \frac{a\notin fn(S),\ \langle P;S\gg\{a\}\rangle \longrightarrow \left\langle P';S'\right\rangle}{\left\langle \mathbf{local}\ a\ \mathbf{in}\ P;S\right\rangle \longrightarrow \langle P';S'\rangle}$$

$$\text{EQUIV: } \frac{\langle P_1;S_1\rangle \equiv_P \left\langle P_1';S_1'\right\rangle \qquad \langle P_2;S_2\rangle \equiv_P \left\langle P_2';S_2'\right\rangle \qquad \langle P_1;S_1\rangle \longrightarrow \langle P_2;S_2\rangle}{\left\langle P_1';S_1'\right\rangle \longrightarrow \left\langle P_2';S_2'\right\rangle}$$

converting $(\widetilde{x})P$ to $(\widetilde{x'})P'$ so that I' is not in the local variables $\widetilde{x'}$ and then replacing, in parallel, every free occurrence of variables from $\widetilde{x'}$ by identifiers (i.e., values, names or variables) from $\widetilde{K}$, respectively. If variable **sender** occurs free in P' it is also replaced by I'. Notice that message continuation Q is activated when the message is received.

DEL describes message delegation. Let $I \lhd l : [\widetilde{K}]$ **then** Q be a message sent to object $(\phi_{\mathbf{sender}}, \delta_{\mathbf{forward}}) \rhd M$ and let us suppose that label l does not exist in M. In this case the message is forwarded by a new location J satisfying the delegation condition. To avoid non-terminating executions we only do this when it is possible to decide that the "new" forwarder of the message is indeed "different" from the previous one in the sense that it cannot communicate with the original object.

The ASK and TELL rules describe the interaction between constraint processes and the store. TELL gives the way of adding information to the store. Process **tell** ϕ **then** P adds constraint ϕ to store S and then activates its continuation P. Such augmentation of the store is the major mechanism in CCP languages for a process to affect the behavior of other processes in the system [11]. For example, agent **tell** $(x = a)$ **then** P informs messages of the form $x \lhd m$ that their target objects are now those accepting messages from a.

ASK gives the way of obtaining information from the store. The rule says that P can be activated if the current store S entails ϕ, or discarded when S entails $\neg\phi$. For instance, process **ask** $(x \in \{a, b\})$ **then** $x \lhd m$ is able to send m to objects accepting messages both from a or b.

An *ask* process that cannot be reduced in the current store S is said to be *suspended* by S. A process suspended by S might be reduced in some augmentation of S. In particular, as the *store* becomes stronger more messages can potentially be accepted by objects. Thus *ask* processes add the "blocking ask" mechanism in *cc* models to the synchronization scheme of object calculi.

PAR says that reduction can occur underneath composition. DEC-V is the way of introducing new variables. Expression $S \gg \{I_1, \ldots, I_n\}$ is as shorthand for store $S \wedge (I_1 = I_1) \wedge \cdots \wedge (I_n = I_n)$. Intuitively, $S \gg \{x\}$ (i.e. $S \wedge x = x$) denotes the addition of variable x to store S when $x \notin fv(S)$. Thus, any variable $x \notin fv(S)$ added to the store by $S \gg \{x\}$ will not be used in subsequent declarations. If $x \in fv(S)$, we can rename x with a new variable $z \notin fv(S) \cup fv(P)$ by α-conversion (i.e. **local** x **in** $P \equiv$ **local** z **in** $P\{z/x\}$ *if* $z \notin fv(P)$). DEC-N is defined in a similar way. Rule EQUIV simply says that P-equivalent configurations have the same reductions.

2.4. Computation as Processes Reduction

A computation in *PiCO* can be seen as a sequence of *states* or *configurations*. Each state consists of two items, a process and a store. The computation proceeds from one state to the next by deciding which process or combination of processes are *enabled* at that point. The decision may require looking at the information accumulated in the store. A process or combination of enabled processes is then selected and run. Running processes result in transforming them into others and (possibly) adding information to the store. The effects of running the processes is thus a new state. The computation ends (if ever) when no enabled processes remain. For example, let the store be $S = \top$, and consider the program (% denotes the *modulo* function),

clone ((**sender** % $2 = 0$)$\wedge$(**sender** $\neq 0$))
  $\triangleright$[e: (z, y)**local** x, w **in** (**tell**(**sender** $\div 2 = x \wedge w = z \times z$)) | $x \triangleleft e[w, y]$] |
clone (**sender** % $2 = 1$)
  $\triangleright$[e: (z, y)**local** x, w **in** (**tell**(**sender** $- 1 = x \wedge w = y \times z$) | $x \triangleleft e[z, w]$)] |
clone (**sender** $= 0$) $\triangleright$ [e: (z, y) **tell** *result* $= y$] |
local x, y, z **in** (**tell** $(x = 2 \wedge y = 1 \wedge z = 3)$ | $x \triangleleft e[z, y]$).

A possible computation for the parallel composition of the four processes in the program given above is the following:

tell $(x = 2 \wedge y = 1 \wedge z = 3)$; $S = \top$

$\downarrow$

$x \triangleleft e[z, y]$ | (**sender** % $2 = 0$)$\wedge$(**sender** $\neq 0$)) $\triangleright$ [...]
; $S = (x = 2 \wedge y = 1 \wedge z = 3)$

$\downarrow$

tell$(x \div 2 = x_1 \wedge w = z \times z)$; $S = (x = 2 \wedge y = 1 \wedge z = 3)$

$$\downarrow$$

$x_1 \triangleleft e[w, y] \mid (\textbf{sender}\ \%\ 2 = 1) \triangleright [\ldots]$
; $S = (x = 2 \wedge y = 1 \wedge z = 3 \wedge x_1 = 1 \wedge w = 9)$

$$\downarrow$$

$\textbf{tell}(x_1 - 1 = x_2 \wedge w_1 = y \times w)$
; $S = (x = 2 \wedge y = 1 \wedge z = 3 \wedge x_1 = 1 \wedge w = 9)$

$$\downarrow$$

$x_2 \triangleleft e[w, w_1] \mid (\textbf{sender} = 0) \triangleright [\ldots]$
; $S = (x = 2 \wedge y = 1 \wedge z = 3 \wedge x_1 = 1 \wedge w = 9 \wedge x_2 = 0 \wedge w_1 = 9)$

$$\downarrow$$

$\textbf{tell}\ \mathit{result} = w_1$
; $S = (x = 2 \wedge y = 1 \wedge z = 3 \wedge x_1 = 1 \wedge w = 9 \wedge x_2 = 0 \wedge w_1 = 9)$

$$\downarrow$$

$S = (x = 2 \wedge y = 1 \wedge z = 3 \wedge x_1 = 1 \wedge w = 9 \wedge x_2 = 0 \wedge w_1 = 9 \wedge \mathit{result} = 9)$

Computations can be defined operationally in terms of an equivalence relation, $\equiv_P$, on configurations describing computation states and a one-step reduction relation, $\longrightarrow$, describing transitions on these configurations. In the following, a configuration is represented as a tuple $\langle P; S\rangle$ consisting of a process P and a store S and $\Longrightarrow$ will denote the reflexive and transitive closure of $\longrightarrow$. Finally, we will say that $\left\langle P'; S'\right\rangle$ is a *derivative* of $\langle P; S\rangle$ iff $\langle P; S\rangle \Longrightarrow \left\langle P'; S'\right\rangle$.

Runtime failure. In the cc-model [11] the invariant property of the store is that it is satisfiable. This can be made to hold in *PiCO* by defining transitions from $\langle \textbf{tell}\ \phi\ \textbf{then}\ P; S\rangle$ just when $S \wedge \phi$ is satisfiable, and otherwise reducing to a distinguished configuration called *fail*. *Fail* denotes a runtime failure which is propagated thereafter in the usual way. For simplicity we do not consider runtime failures, but we can add these rules orthogonally, as in [14], without affecting any of our results.

Potentiality of reduction. Whenever we augment the store, we may increase the potentiality of process reduction, that is, the number of possible transitions from a configuration. The following proposition states that any agent P' obtained from a configuration $\langle P; S_1\rangle$ can be obtained from a configuration $\langle P; S_2\rangle$, S_2 being an augmentation of S_1. The proposition can be easily proved from the operational rules given above.

Proposition 2.3 *If* $S_2 \vdash S_1$ *and* $\langle P_1; S_1\rangle \longrightarrow \left\langle P_2; S_1'\right\rangle$ *then* $\langle P_1; S_2\rangle \longrightarrow \left\langle P_2; S_2'\right\rangle$ *and* $S_2' \vdash S_1'$.

We consider next an example illustrating how delegation works in *PiCO*. Let us suppose we have two objects generating musical chords depending on a base note:

clone (**sender** $\in \{do, re, \ldots, si\}$, **forward** $\in \{do\sharp, re\sharp, \ldots, la\sharp\}$)
$\triangleright$[*basic*: Chord_1(**sender**)] |

clone (**sender** $\in \{do\sharp, re\sharp, \ldots, la\sharp\}$)
$\triangleright$[*basic*: $Chord_1$(**sender**) & $minor$: (x) $Chord_2(x,$ **sender**$)$]

Parallel composition of the above objects with the process

tell *note* $\in \{re,\ fa\}$ **then** *note* $\triangleleft$ *minor*[*note*]

behaves as follows:

1. Since message location *note* satisfy the guard of the first object (with **sender** replaced by *note*), a communication is established.

2. Message *minor* does not exist in the methods of the first object, therefore delegation is attempted. Since message location *note* does not satisfy the delegation condition in the first object, delegation is accepted.

3. A new process **local** J **in tell** $J \in \{do\sharp, re\sharp, \ldots, la\sharp\}$ **then** $J \triangleleft$ *minor*[*note*] is constructed.

4. The new process is able to communicate with the second object. The result is executing process $Chord_2(note,\ J)$.

Note that the original and delegated objects could very well have methods in common (label *basic* in both objects).

Behavioral equivalence. In [15] a reduction-equivalence relation for an extension of the π-calculus with constraints was defined. This relation equates configurations whose agents can communicate through the same channels at each transition. For each process identifier I and label l, this is expressed by means of an observation predicate detecting the possibility of performing a communication with the external environment along identifier I and label l in a store S. Behavioral equivalence can be similarly defined for *PiCO*. The details of this are out of the scope of this paper.

Names and Variables. In the π-calculus there is no difference between names and variables [13]. Names and Variables in *PiCO* are different entities because of the presence of constraints.

The following example illustrates this difference. Let

$$P_1 \equiv \textbf{local}\ x\ \textbf{in local}\ y\ \textbf{in}\ (\textbf{ask}\ \neg(x = y)\ \textbf{then}\ Q)$$

and

$$P_2 \equiv \textbf{local}\ a\ \textbf{in local}\ b\ \textbf{in}\ (\textbf{ask}\ \neg(a = b)\ \textbf{then}\ Q)$$

Since a and b are different names, configuration $\langle P_2; \top\rangle$ is able to reduce process P_2 to Q, whereas process P_1 is suspended in configuration $\langle P_1; \top\rangle$. The need for names is justified because, conveniently used, they provide a unique reference to concurrent objects which can be used for data encapsulation as in [14]. We will use this useful property to encode classes.

3. Classes

In this section we discuss the definition in *PiCO* of basic object-oriented constructs. Most definitions in the rest of the paper use the restricted syntax of objects located in variables or names, but not in constraints. The reason is that we want to stress the relationship between defining processes in the calculus and programming in *Cordial*, the visual language running on top of it. As of this writing we have not yet included visual representations of constraint guarded objects in *Cordial* so our musical examples in *PiCO* do not use this feature.

The basic object-oriented constructs we discuss are *classes*, *instances* and *inheritance*. *Classes* denote sets of objects. A class is just a frame used to construct objects belonging to its associated set. It consists of *attributes*, *class constraints* and *methods*. Attributes define the local state of instances (i.e. objects) of the class. Methods implement operations on the state. Class constraints are predicates that must be satisfied by the state of any object of the class. Classes can be organized hierarchically. This hierarchy is defined by an *inheritance* relation. For the purpose of the simple examples presented here, a class B is said to *inherit* from class A if class A can provide some of the services (i.e. methods) for class B. Thus, class B may decide to respond to some requests by passing the request to class A. Delegation in *PiCO* objects is used to implement this behavior.

Classes are modeled as cloned (i.e. persistent) located objects without delegation. A particular class object has at least two methods: *new* and *super*. The unary method *super* gives access to the location of the object representing the superclass (if any) of the class currently being defined. Method *new* creates a new instance of the class. This instance is a located object with the collection of methods defined for the class. Each attribute defined for an instance of the class is stored in a *cell*, in a similar way as is done in *Oz* [13]. Cells are objects having reading and writing methods, as shown below.

clone *cellmaker*$\triangleright$
[*create*: (x, y) $x \triangleright$ [*get*: (z) **tell**$z = y$
then cellmaker $\triangleleft$ *create*(x, y)&
set: (z)*cellmaker* $\triangleleft$ *create*(x, z)]]

Message *cellmaker* $\triangleleft$ *create*$[c,$ *value*$]$ creates a cell object located at c. The contents of the cell is initially set to *value*. Notice that setting a value creates a new cell. Since attributes in instances are represented by cells, this means that previous constraints imposed on an attribute are "lost" when a new value is assigned to it. In particular, class constraints (as explained below) must be asserted again for new values of instance attributes.

The instance of a class has a unary method for each attribute. This method can be used to access this attribute cell. Classes in *PiCO* are supposed to include one or several class constraints that must be satisfied by the attributes of each instance of the class. Class constraints are asserted at instance creation. The implementation of classes is shown below. In what follows, we write $\widetilde{x}$ for a sequence $x_1, x_2, \ldots, x_k$. We also take the convention that when a subscripted variable or name x_i appears in a process, the process really represents the parallel composition of all processes of the same form, one for each variable or name in the sequence. For example, $cell_i \triangleleft set[v_i]$ in process **local** $\widetilde{cell}$, $\widetilde{v}$ **in** $cell_i \triangleleft set[v_i]$ represents the parallel composition $cell_1 \triangleleft set[v_1] \mid cell_2 \triangleleft set[v_2] \mid \ldots$ containing one subprocess for

each corresponding element of $\widetilde{cell}$, $\widetilde{v}$

$$\begin{array}{l}\textbf{clone}\ class \triangleright\ [new{:}(self)\ \textbf{local}\ d, \widetilde{cell}, \widetilde{val}, \widetilde{c}\ \textbf{in}\\ \qquad superclass \triangleleft new[d]|\\ \qquad \textbf{clone}\ (self, d) \triangleright\ [\ attr_i{:}\ (x)\textbf{tell}\ x = cell_i\ \&\\ \qquad\qquad update{:}\ (\widetilde{v})cell_i \triangleleft set[v_i]\ |\\ \qquad\qquad\qquad c_i \triangleleft cnstr[\widetilde{v}]]\\ \qquad |\ cellmaker \triangleleft create[cell_i, val_i]\\ \qquad |\ \textbf{clone}\ c_i \triangleright [cnstr{:}\ (\widetilde{v})\, R_i(\widetilde{v})]\ |\ c_i \triangleleft cnstr[\widetilde{val}]\\ \quad super{:}\ (r)\textbf{tell}\ r = superclass]\end{array}$$

Method *new* of a class creates instances for that class and all classes in the superclass chain. This is accomplished by message $superclass \triangleleft new[d]$. Object *superclass* is representing the superclass (if any) and d is a new instance of that class returned by the message. Cells associated with attributes are created by messages $cellmaker \triangleleft create[cell_i, val_i]$, where variable val_i represents the attribute initial value. Message $c_i \triangleright [cnstr{:}\ (\widetilde{v})\, R_i(\widetilde{v})]$ defines an object capable of asserting class constraint R_i on $attribute_i$. Notice that constraint R_i may involve values for other attributes. Message $c_i \triangleleft cnstr[\widetilde{val}]$ asserts the attribute constraint. Class constraints are "reified" as objects in this way in order to simplify reasserting them when attribute values are changed.

In some of the examples we discuss further below we assume the existence of a class *integer* denoting integer values.

In the musical example we describe next we will refer to a particular type of musical chord containing four notes defined modulo transposition (i.e. the base note of the chord is not specified). These *Estrada* chords, named after the mexican composer who first formally defined them, contain three intervals (expressed in number of semitones between consecutive notes) whose sum modulo 12 must be equal to zero. To represent the intervals of a chord we assume a constraint system over some suitable finite domain of integers. Class E, of *Estrada* chords, is defined as follows.

$$\begin{array}{l}\textbf{clone}\ E \triangleright\ [\ new{:}\ (self)\\ \qquad \textbf{local}\ c, \widetilde{cell}, \widetilde{val}\ \textbf{in}\\ \qquad \textbf{clone}\ self \triangleright\ [\ x_i{:}\ (y)\ \textbf{tell}\ y = cell_i \&\\ \qquad\qquad update{:}\ (\widetilde{v})\ldots\ \&\\ \qquad |\ cellmaker \triangleleft create[cell_i, val_i]\\ \qquad |\ \textbf{clone}\ c \triangleright [cnstr{:}\ (\widetilde{v})\ \textbf{local}\ w, z\ \textbf{in}\\ \qquad\qquad\qquad v_1 \triangleleft add[v_2, w]\ |\ v_3 \triangleleft add(w, z)\\ \qquad\qquad\qquad z \triangleleft modulo[12, 0]\ ]\\ \qquad c \triangleleft cnstr[\widetilde{val}]\ \&\\ \quad super{:}\ (r)\ \textbf{tell}\ r = E]\end{array}$$

Each of the three attributes of a *Estrada* chord contains one interval between consecutive notes of the chord, represented, as usual, by cells containing integers. The class constraint, encoded in the replicated object located at c, ensures that attributes obey the *Estrada* property.

3.1. Objects with Complex Attributes

Objects may have attributes containing other arbitrary objects. A standard example are the classes *cons* and *list* defined bellow. The class *cons* has attributes *car* and *cdr* that allow accessing the components of the pair represented by the instance.

$$
\begin{array}{l}
\textbf{clone}\ \mathit{cons} \triangleright [\mathit{new}{:}(\mathit{self}) \\
\qquad \textbf{local}\ a_{car}, a_{cdr}, v_{car}, v_{cdr}\ \textbf{in} \\
\qquad \textbf{clone}\ \mathit{self} \triangleright [\mathit{car}{:}\ (x)\textbf{tell}\ x = a_{car}\& \\
\qquad\qquad\qquad \mathit{cdr}{:}\ (x)\textbf{tell}\ x = a_{cdr}\& \\
\qquad\qquad\qquad \mathit{update}{:}\ (v_{car}, v_{cdr}) \ldots\ \& \\
\qquad | \ \mathit{cellmaker} \triangleleft \mathit{create}[a_{car}, v_{car}] \mid \ \ldots \\
\quad \mathit{super}{:}\ (r)\ \textbf{tell}\ r = \mathit{cons}]
\end{array}
$$

List are modeled as instances with a unique attribute $value$, whose associated cell ($cell_{self}$ in the program bellow) can be assigned either the particular identifier *nil* (the empty list) or a *cons* object. The *cdr* part of this object should always be restricted to a *list* object. As usual, $list$ instances also have methods *head* and *tail*.

$$
\begin{array}{l}
\mathit{list} \triangleright [\mathit{new}{:}(\mathit{self}) \\
\qquad \textbf{local}\ \mathit{cell}_{\mathit{self}}, v\ \textbf{in} \\
\qquad \textbf{clone}\ \mathit{self} \triangleright [\mathit{value}{:}\ (x) \\
\qquad\qquad\qquad\qquad \textbf{tell}\ (x = \mathit{cell}_{\mathit{self}})\& \\
\qquad\qquad\qquad \mathit{head}{:}\ (x)\ \textbf{tell}\ x = \mathit{self}.\mathit{value}.\mathit{car}\ \& \\
\qquad\qquad\qquad \mathit{tail}{:}\quad (x)\ \textbf{local}\ \mathit{newlist}\ \textbf{in} \\
\qquad\qquad\qquad\qquad \mathit{list} \triangleleft \mathit{new}[\mathit{newlist}] \\
\qquad\qquad\qquad\qquad | \ \mathit{newlist}.\mathit{value} \leftarrow \mathit{self}.\mathit{value}.\mathit{cdr}\& \ldots] \\
\qquad \mathit{cellmaker} \triangleleft \mathit{create}[\mathit{cell}_{\mathit{self}}, v] \\
\quad \mathit{super}{:}\ (r)\ \textbf{tell}\ r = \mathit{list}]
\end{array}
$$

Message, $\mathit{list} \triangleleft \mathit{new}[l] \mid l \triangleleft \mathit{update}[\mathit{acons}]$, where $\mathit{acons}.\mathit{car} \leftarrow a \mid \mathit{acons}.\mathit{cdr} \leftarrow \textbf{nil}$ creates the list $[a]$.

4. *Cordial*: A Visual Language for *PiCO*

Our research project includes the development of visual programming tools for musical composition. *Cordial* [8] is a visual programming language integrating object oriented and constraint programming intended for musical applications. The semantics of *Cordial* has been defined in terms of *PiCO*. In this section we illustrate briefly some constructs of *Cordial* together with their denotation in *PiCO*.

Cordial is an iconic language in the spirit of [2]. The basic notion is that of a *patch*. A patch is a layout of forms (icons or other) on the screen.*Links* between forms in a patch establish control dependencies in a computation. Data types and structures are also defined visually. A visual class definition in *Cordial*, for example, has three main sections (see figure 1): attributes, methods and constraints.

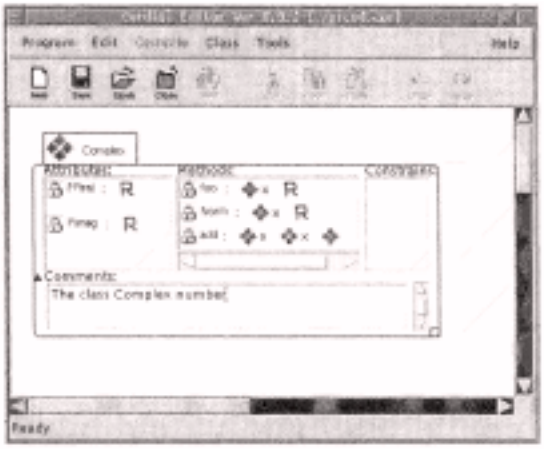

Figure 1. Class definition in *Cordial*.

The translation of the class definition in figure 1 into a *PiCO* process can be roughly defined as follows:

clone *complex*▷
[*new*: (*self*)
local $c, cell_r, cell_i, v_r, v_i$ **in**
clone *self*▷
[*real*:(x)**tell** $x = cell_r$&
imag: (x)**tell** $x = cell_i$&
add: ($complex_1, complex_2$)
$self.real \triangleleft add[complex_1.real, complex_2.real]|$
$self.imag \triangleleft add[complex_1.imag, complex_2.imag]\& \ldots]$
$cellmaker \triangleleft create[cell_r, v_i]| \ldots \&$
super: (r)**tell** $r = complex$]

A method definition (see figure 2), is a collection of messages (the envelopes in figure 2), conditionals and formal arguments (the circles in figure 2). Links connecting two elements define identity relations. For example, linking visual object V_o with the i-th circle of envelope E means that the i-th argument of the message associated with E is equal to V_o. The body of the method in figure 2 shows a visual representation of objects as rectangular boxes with an icon inside. The target of the message is the unique object connected by a double line with the envelope. A message defines a particular relation that must be satisfied by the arguments and target object of the message.

Non primitive method *foo* of class *complex* shows the use of a conditional (figure 3).

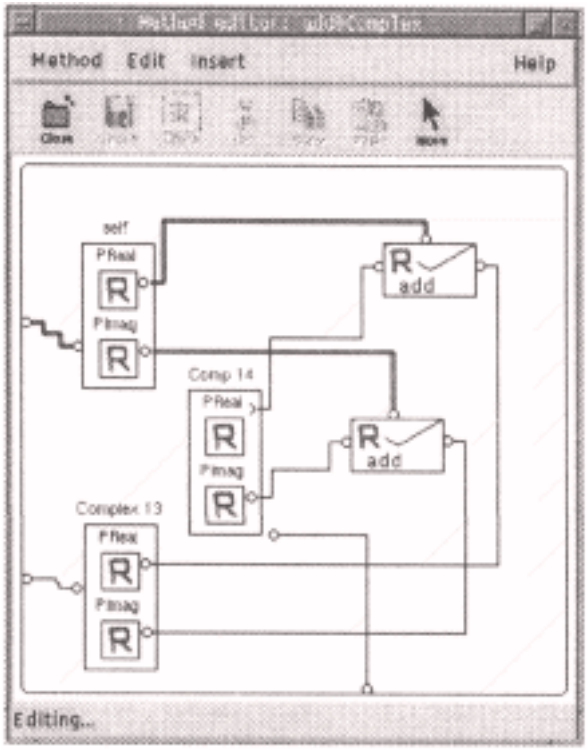

Figure 2. Method definition in *Cordial*.

The definition of method *foo* in *PiCO* is:

$$
\begin{aligned}
&foo(r)\textbf{local } tmp\textbf{in } (self \lhd norm(tmp) \mid \\
&\qquad\qquad \textbf{ask } tmp = 1 \textbf{ then tell } r = 1 \mid \\
&\qquad\qquad \textbf{ask } tmp = 0 \textbf{ then tell } r = -1 \mid \\
&\qquad\qquad \textbf{ask } tmp \neq 1 \wedge tmp \neq 0 \textbf{ then tell } r = tmp)
\end{aligned}
$$

5. Using *Cordial*: A Music Composition Example

In this section we solve a simple (but non trivial) music composition problem. With the solution of this problem we argue the advantages of programming in a visual concurrent constraint object oriented language that allows including constraints in class definitions.

The example makes extensive use of constrained classes to control the evolution of two melodic voices: each voice evolves according to independent melodic properties, but they must synchronize at given temporal points and they must also satisfy a number of harmonic properties when their notes sound at the same time.

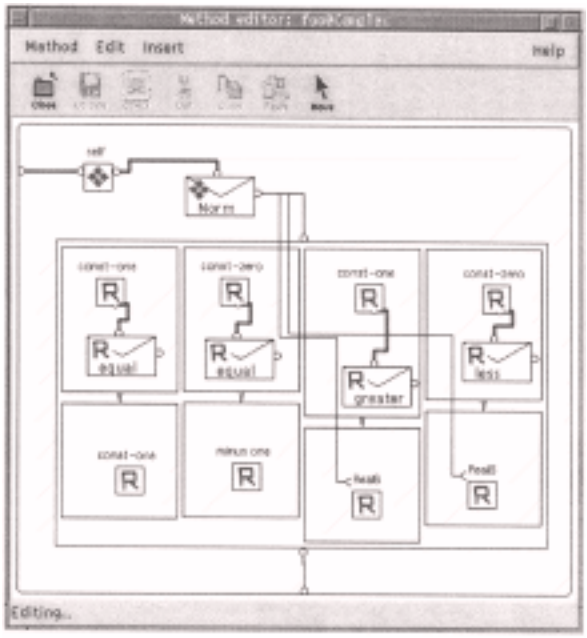

Figure 3. Message definition and conditional in *Cordial*.

The two melodic voices, $Voice_1$ and $Voice_2$, will start at the same time and will be generated until an external condition is met. Notes in the two voices have three attributes: *pitch*, *duration* and *dynamic*. Their pitch values should be in a set of allowed ambitus (i.e. a range) *Amb*, their durations must belong to a given set, say, $\{4, 2, 1\}$ (where 2 represents, say, a quarter note). Notes must also satisfy the following four conditions:

1. If n_1 and n_2 are two consecutive notes in $Voice_i$ ($i \in \{1, 2\}$) having pitches equal to x and y, respectively, then **tell** $max(x, y) - min(x, y) \in Melody_i$, where $Melody_i$ is a given set of integers.

2. Notes are divided into groups of duration equal to 4. A group could be made to correspond to any meaningful rhythmic division, for instance a beat or a measure. Each group contains notes of both $Voice_1$ and $Voice_2$.

3. Notes starting a group are constrained differently. The first note in each duration group has its *dynamics* equal to, say, 127. Other notes have its *dynamics* equal to 70.

4. Let n_1 and n_2 be notes from the same group in $Voice_1$ and $Voice_2$ respectively. If they sound at the same time and their durations are both greater than 1 then the absolute difference between their pitch values must be in a certain interval set, *HARMONSET1*. In any other case, the absolute difference between their pitches must be in *HARMONSET2*.

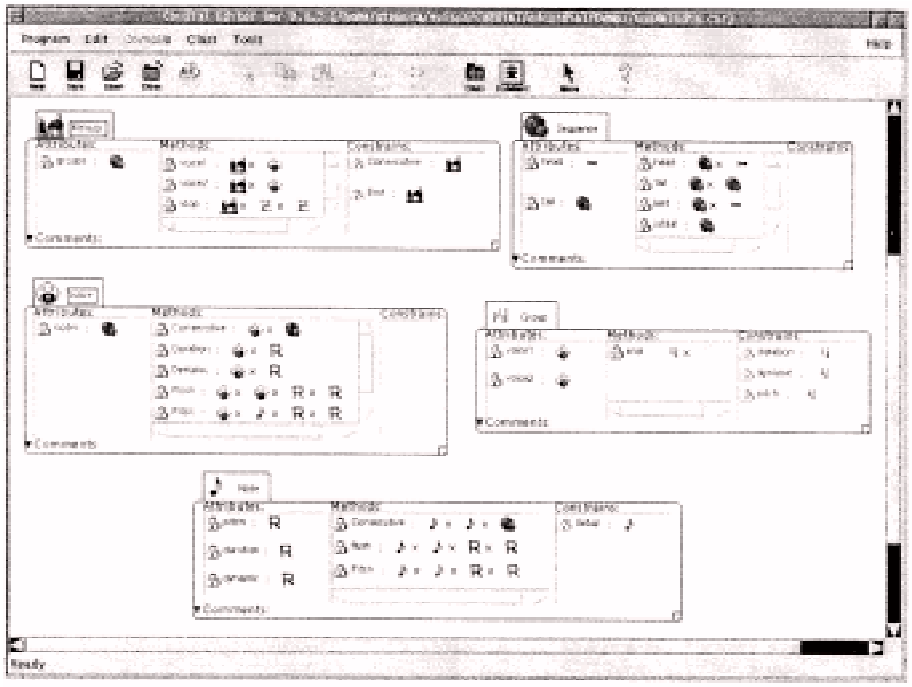

Figure 4. Class definitions.

To construct a solution for this problem taking advantage of the object oriented features in *Cordial*, several classes are first defined. In the *Cordial* editor all of these must be drawn on the program area, as can be seen in figure 4. Class *Melody* is the main class. It has the attribute *groups*, a sequence of groups representing the two voices of the melody, two class constraints, *consecutive* and *end*, and three class methods, *voice*1, *voice*2, and *stop*. Class *Group* models a time window on the two voices of the class *Melody*. Each of these voices are represented by the class *Voice*, modeled as a sequence of notes. Each note is represented by class *Note*. This class characterizes a sound by its pitch, duration, and dynamics.

We describe next some of the methods that define the implementation of the proposed solution. For the sake of brevity, not all of the required methods are illustrated in the figures.

The main method (figure 5) simply creates an instance of the class *Melody*. Class constraints encode all constraints required by the problem. Constraint *Consecutive* ensures that both voices of the melody satisfy the consecutive notes constraint of class *Voice*. That is, the melodic distance between consecutive notes must belong to a particular set. Constraint *End* restricts the voices on the melody to stop only when their last notes have the same duration. In other words, it constraints the length of the *groups* attribute to the number of groups generated.

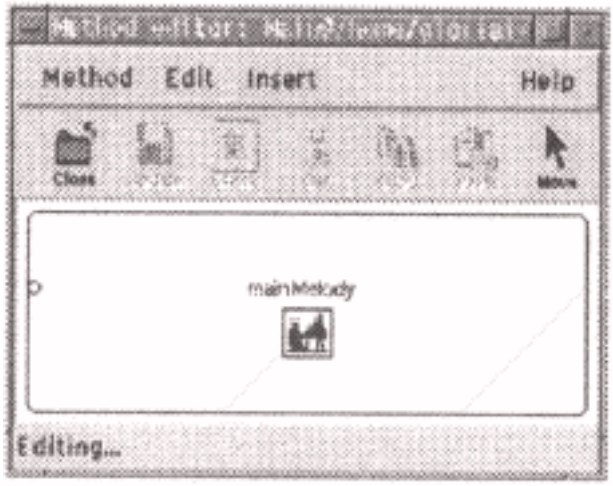

Figure 5. Main method.

Methods *Voice*1 (figure 6) and *Voice*2 recursively iterate over the sequence of groups (attribute *groups* in class *Melody*), effectively extracting from them their corresponding voices by building a list with their elements.

Method *Stop* (figure 7) limits the number of groups in a melody by instantiating the length of its *group* attribute on the number of groups generated before the stop condition is met.

When the (presumably) final notes of each voice in a group have the same duration, the end condition holds. Method *End* in class *Group* (figure 8) signals this by instantiating its string argument to "stop." This class has three class constraints. Class constraint *Duration* sets the duration of each group on each voice by instantiating the corresponding durations to some specific value. Class constraint *Pitch* constrains pitches of simultaneously sounding notes in two voices of the group to a given set of values. This is accomplished by method *Pitch* of classes *Voice* and *Note*. Finally, class constraint *Dynamic* (figure 9) sets the value of the *dynamic* attribute of the first note in each voice to 127. The *dynamic* attribute of all other notes in the group is set to 70.

Method *Duration* in class *Voice* (figure 10) restricts the sum of the duration of the notes on the sequence to a given value. Method *Dynamic* sets the dynamics of the notes on the voice to a given value, whereas method *Consecutive* forwards itself to each note of the voice.

Method *Pitch* in class *Voice* is defined for two different signatures: the first one (see figure 11) asserts pitch relations between its own voice notes and the notes of the supplied voice argument. The second asserts pitch relations between its own voice notes and the single note supplied as argument.

Class constraint *Set-up* restricts pitch and duration attributes of a note to belong to a given set of values. Finally, *Pitch* asserts the above mentioned relation on the pitch of two simultaneously sounding notes.

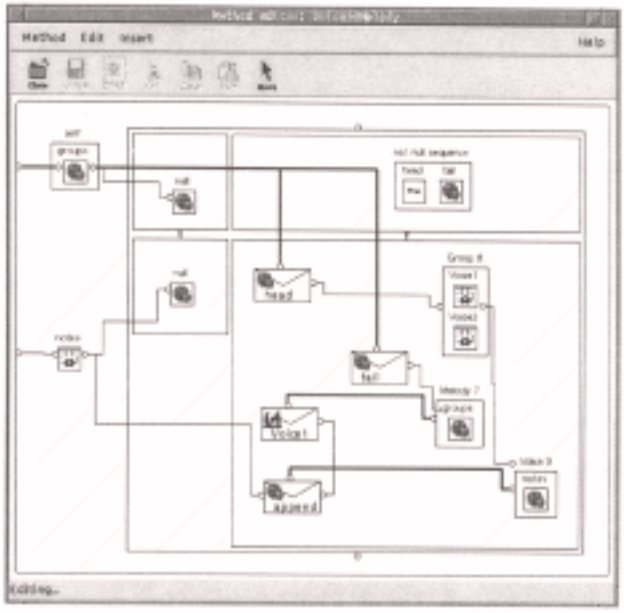

Figure 6. Method *voice*1 in *Melody*.

5.1. Translation of the Example into PiCO

The first issue is to find suitable constraint systems for musical applications. Finite domains (FD) systems [4] can handle most musical problems in the (traditional) harmonic domain, while other aspects such as rhythm or timbre would very likely need dense domains and a richer set of operations. For the purposes of the above problem, it is clear that FD is adequate. We assume the constraint system is defined over some suitable finite domain (for *pitch*, *duration* and the like).

clone *note*▷
  [*new*: (*self*)**local** $cell_p$, $cell_{du}$, $cell_{dy}$, v_p, v_{du}, v_{dy} **in**
      clone *self*▷
        [*pitch*: (x)**tell** $x = cell_p$&
         duration: (x)**tell** $x = cell_{du}$&
         dynamic: (x)**tell** $x = cell_{dy}$&
         contiguous: (*aNote*) **local** int_1, int_2 **in**
              $self.pitch \triangleleft sub[aNote.pitch, int_1] | int_1 \triangleleft abs[int_2] |$
              $int_2 \triangleleft inrank[MELODY]$&
         harmony: $(note', ot_1, ot_2)$ **local** *hint*, rt_1, rt_2 **in**
              $rt_1 \leftarrow \mathbf{4} - ot_1 - self.duration|$
              $rt_2 \leftarrow \mathbf{4} - ot_2 - note'.duration|$
              ask

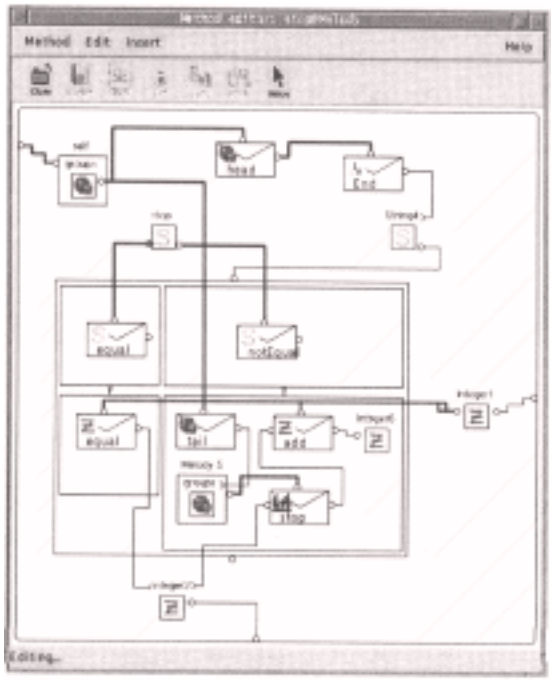

Figure 7. Method *stop* in *Melody*.

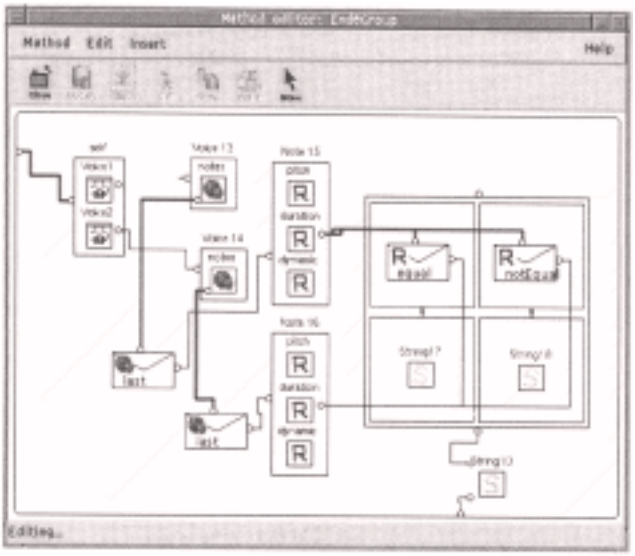

Figure 8. Method *end* in *Group*.

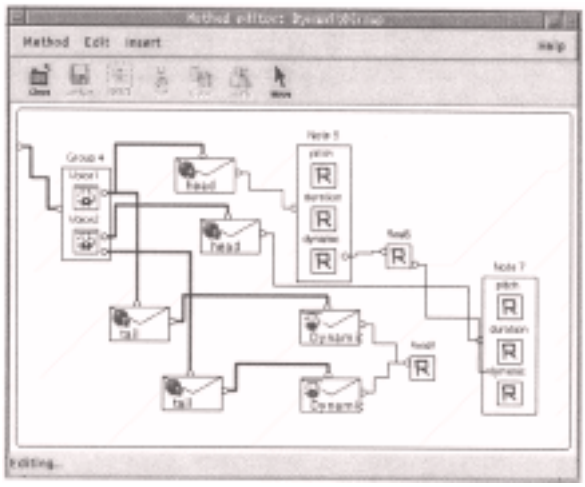

Figure 9. Class constraint *Dynamic* in *Group*.

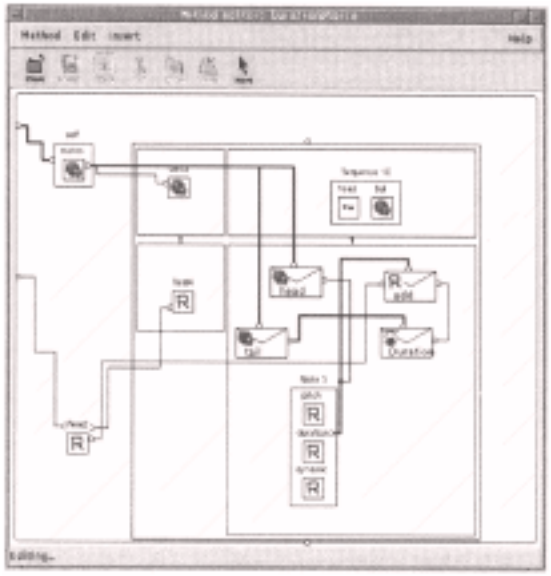

Figure 10. Method *Duration* in *Voice*.

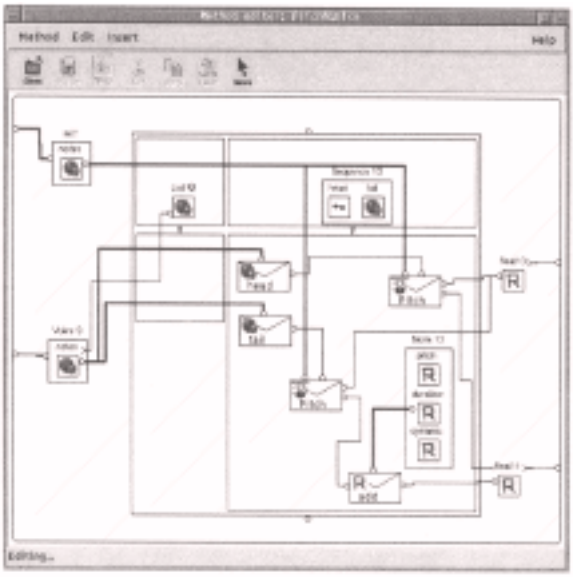

Figure 11. Method *Pitch* in *Voice*.

$rt_1.value \leq note'.duration.value \wedge$
$self.duration.value > 1 \wedge$
$note'.duration.value > 1$
then
hint $\leftarrow$ *abs*(*self.pitch* − *aNote.pitch*)
hint $\triangleleft$ *inrank*[*HARMONISET*$_1$]|
ask
$\neg(rt_1.value \leq note'.duration.value \wedge$
$self.duration.value > 1 \wedge$
$note'.duration.value > 1)$
then
hint $\leftarrow$ *abs*(*self.pitch* − *aNote.pitch*)
hint $\triangleleft$ *inrank*[*HARMONISET*$_2$]|
ask ... **then** ... & ...]
cellmaker $\triangleleft$ [$cell_p$, v_p] | ... &
super: (*r*) **tell** r = *note*]

|

clone *voice*$\triangleright$
[*new*: (*self*)**local** *c*, *a*, *v* **in**
clone *self*$\triangleright$
[*notes*: (*x*)**tell** $x = a$&

```
        contiguous: ( ) local b1, b2 in
            self.notes ◁ isnil[b1]|
            ask b1 = ⊥ then self.notes.tail ◁ isnil[b2]|
                ask b2 = ⊥ then
                    self.notes.head ◁ contiguous[self.notes.tail.head]
                    voice1 ◁ notes[self.notes.tail]| voice1 ◁ contiguous[ ]&
        duration: (int) local b, int2 in
            self.notes ◁ isnil[b]|
            ask b = ⊤ then tell int.value = 0|
            ask b = ⊥ then
                voice1 ◁ notes[self.notes.tail]|
                voice1 ◁ duration[int2]|
                self.notes.head.duration ◁ add[int2, int]&
        dynamic: ( ) . . . &
        harmony_v: (voice2, ot1, ot2) local b, ot0, voice1 in
            self.notes ◁ isnil[b]
            ask b = ⊥ then
                ot1 ◁ add[self.notes.head.duration, ot0]|
                voice1 ◁ notes[self.notes.tail]|
                voice1 ◁ harmony_v[voice2, ot0, ot2]|
                voice1 ◁ harmony_n[self.notes.head, ot2, ot0]&
        harmony_n: (note2, ot1, ot2) . . . & . . .]
    cellmaker ◁ [a, v]& super: (r) tell r = voice]

|

clone group▷
  [ new: (self)local c1, c2, c3, a1, a2, v1, v2 in
      clone self▷
        [ voice1: (x)tell x = a1& voice2: (x)tell x = a2& . . .]|
      c1 ▷ M_group1|c1 ◁ duration[]| c2 ▷ M_group2|c2 ◁ dynamic[]|
      c3 ▷ M_group3|c3 ◁ harmony[]| cellmaker ◁ [ai, vi]&
    super: (r)tell r = group]

|

clone melody▷
  [ new: (self)local c, a1, v1, in
        clone self▷
          [ groups: (x)tell x = a1&
            voice1: (notes) local melody1 in
              self.groups ◁ isnil[b]|
              ask b = ⊤ then tell notes = nil
              ask b = ⊥ then
                  self.groups.head.voice1 ◁ cat[notes2, notes]|
                  melody1 ◁ groups[self.groups.tail]|
```

$$
\begin{aligned}
&\qquad\qquad melody_1 \lhd voice_1[notes_2]\& \\
&\qquad voice_2\colon\ (notes) \ldots \& \ldots]| \\
&\quad c \rhd M_{melody1}|c \lhd melody[]|\ cellmaker \lhd [a_1, v_1]\& \\
&super\colon\ (r)\textbf{tell}\ r = melody]
\end{aligned}
$$

where:

$$
\begin{aligned}
M_{melody1} \equiv [melody\colon\ ()\textbf{local}\ notes_1, notes_2\ \textbf{in} \\
self \lhd voice_1[notes_1]|self \lhd voice_2[notes_2]| \\
voice_1 \lhd notes[notes_1]|voice_2 \lhd notes[notes_2]| \\
voice_1 \lhd contiguous[]|voice_2 \lhd contiguous[]
\end{aligned}
$$

$$
\begin{aligned}
M_{group1} \equiv [duration\colon\ (int) \\
self.voice_1 \lhd duration[\mathbf{4}]|self.voice_2 \lhd duration[\mathbf{4}]
\end{aligned}
$$

$$
\begin{aligned}
&M_{group2} \equiv \\
&[dynamic\colon\ ()\ \textbf{local}\ voice_1, voice_2,\ \textbf{in} \\
&\quad \textbf{tell}\ self.voice_1.head.dynamic = \mathbf{127} \\
&\quad |\textbf{tell}\ self.voice_1.head.dynamic = \mathbf{127}| \\
&\quad voice_1 \lhd notes[self.voice_1.tail]|voice_2 \lhd notes[self.voice_2.tail]| \\
&\quad voice_1 \lhd dynamic[]|voice_2 \lhd dynamic[]
\end{aligned}
$$

$$
M_{group3} \equiv [harmony\colon\ ()\ self.voice_1 \lhd harmony_v[self.voice_2, \mathbf{0}, \mathbf{0}]
$$

6. Related and Future Work

In this section we highlight the expressiveness of *PiCO* by showing natural and elegant encodings of basic constructs of the π, ρ and MCC calculi (see [6], [12], [10]). We do this informally and do not provide proofs of correctness of the encodings. However, we conjecture bisimilarities between the ρ-calculus and a restricted version of *PiCO* having no delegation and only equality for constraint guards, and between the *MCC*-calculus and this same reduced *PiCO* supplied with the *MCC* condition on the store for process communication. We plan to pursue the proof of these conjectures in the near future.

6.1. PiCO and the π-Calculus

One of the key features of the π-calculus is process mobility. In [6] this is illustrated using the example of mobile telephones. A CENTRE is in permanent contact with two BASE stations,each in a different part of the country. A CAR with a mobile telephone moves about the country; it should always be in contact with a BASE. If it gets rather far from its current BASE contact, then (in a way that is not modeled in the above mentioned paper) a hand-over procedure is initiated, and as a result the CAR relinquishes contact with one BASE and assumes contact with another.

A *PiCO* solution in the same spirit as that shown in [6] is:

$$
\begin{array}{l}
car\_industry \triangleright [make\_car\colon\ (car) \\
\qquad\qquad car \triangleright [talk\colon\ (m)\textbf{local}\ base\ \textbf{in} \\
\qquad\qquad\qquad\qquad central \triangleleft get\_base[base]|base \triangleleft com[m]| \\
\qquad\qquad\qquad\qquad car\_industry \triangleleft make\_car[car]]]| \\
base_i \triangleright [com\colon\ (m)\ P(m)]| \\
\textbf{local}\ c\ \textbf{in} \\
\qquad cellmaker \triangleleft create[c, 0]| \\
\qquad \textbf{clone}\ central \triangleright [get\_base\colon\ (base)c \triangleleft get[base]\ \textbf{then}\ c \triangleleft set[(base + i)\%2]]
\end{array}
$$

Here, object *car_industry* is used for creating car processes, objects $base_i$, $i = 1, 2$ process messages sent by cars, and object *central* decides which base must receive the messages. Note that *central* simply switches the base receiving the message, as in [6].

In a more practical setting, we would have a set of communicating centers in a country. Each of them having a set of independent associated bases. Each one of these bases is able to process a message just when its distance to the *car* is less than or equal to some given value. The role of the central base is to choose some base of a different communication center when messages from a car cannot be transmitted by any of the bases belonging to the current center.

We use *PiCO* constraint "located" objects for modeling this:

$$
\begin{array}{l}
car\_industry \triangleright [make\_car\colon\ (car, pos) \\
\qquad\qquad car \triangleright [talk\colon\ (m)pos \triangleleft com[m]|car \triangleleft make\_car[car, pos]\& \\
\qquad\qquad\qquad get\colon\ (x)\textbf{tell}\ pos = x|car\_industry \triangleleft make\_car[car, pos]\& \\
\qquad\qquad\qquad set\colon\ (x)car \triangleleft make\_car[car, x]]]| \\
\phi_i(sender) \triangleright [com\colon\ (m)\ P(m)]| \\
\Psi(sender) \triangleright [com\colon\ (m)\ \textbf{local}\ base\ \textbf{in} \\
\qquad\qquad\qquad\qquad central \triangleleft get\_base(sender, base)|base \triangleleft com[m]]| \\
\textbf{clone}\ central \triangleright [get\_base\colon\ (pos, base)P(pos, base)]
\end{array}
$$

Here, $\phi_i(p) = distance\_to\_B_i(p) \leq d_i,\ i = 1, 2$ and $\Psi(p) = \neg\phi_1(p) \wedge \neg\phi_2(p)$.

In addition to the communication method *talk*, a car has methods *get* and *set* for retrieving or setting its position attribute. Bases are "located" in constraints ϕ_i. An additional object "located" in Ψ is used to call *central* just when there is no base a car can communicate with.

The above solution can be improved in two aspects:

- Elegance: Since a *car* object is very similar to a *cell* object (see section 3), we can model a *car* as an object that knows how to answer *talk* messages but delegates to a cell object the answering of *get* and *set* messages.
- Efficiency: Notice that methods *talk*,*get* and *set* rebuild a car object each time they are called. This can also be avoided using cells.

The result is:

$$
car\_industry \triangleright [make\_car\colon\ (car, pos)
$$

Table 4. Basic *MCC* processes in *PiCO*.

MCC **process**	*PiCO* **encoding**
$send(a, \lambda\vec{x}\ \phi(\vec{x}))$	$\mathbf{local}\ i\ \mathbf{in}\ i \triangleright [assert\colon\ (\vec{x})\ tell\ \phi(\vec{x})]\ \mid\ a \triangleleft m[i]$
$receive(a, \vec{y}).S(\vec{y})$	$a \triangleright [m\colon\ (i)\ i \triangleleft assert[\vec{y}]\ \mathbf{then}\ S(\vec{y})]$

$$
\begin{array}{l}
\mathbf{local}\ cell_{car}\mathbf{in}\\
\quad cellmaker \triangleleft create[cell_{car}, pos]|\\
\quad \mathbf{clone}\ (car, cell_{car})\ \triangleright [talk\colon\ (m)\mathbf{local}\ pos\ \mathbf{in}\\
\qquad\qquad car \triangleleft get[pos]|\\
\qquad\qquad pos \triangleleft com[m]]]|\\
\mathbf{clone}\ mover \triangleright [new\_pos\colon (car)\mathbf{local}\ pos\ pos'\ \mathbf{in}\\
\qquad car \triangleleft get[pos]|P(pos, pos')|car \triangleright set[pos']]
\end{array}
$$

6.2. MCC Processes in PiCO

As was mentioned in the introduction, *MCC* [10] considers extensions of the *cc* model to account for reliable communication of agents having their own local constraints store. A fundamental contribution of *MCC* is the ability to restrict communications to non inconsistent stores so as to maintain independence of deductions among local stores. This cannot be directly modeled in *PiCO* using the operational rules given in section 2.3. Of course, by adding the same condition of store consistency to the premise of the communication rule it is likely that the "local stores" behavior of *MCC* could be encoded in *PiCO*.

We describe here a general encoding in *PiCO* of the important *send* and *receive MCC* processes. We illustrate the encoding by comparing the behavior of both *MCC* and *PiCO* in a simple communication example using these kind of processes. We then argue that the *MCC* behavior is correctly simulated by the encoding.

It was shown in [10] how to implement dynamic reconfiguration of communication networks in the *MCC*-calculus. The implementation is illustrated with an example using two agents p and q sharing a channel a. Agent p also shares channel b with a third agent r. The following procedure declarations are given (declarations of s and t are left unspecified):

$$
\begin{array}{rcl}
p(a, b) & :: & \exists_x send(b, \lambda x\ x = a) \parallel \exists_{\vec{y}} receive(a, \vec{y}).s(\vec{y})\\
q(a) & :: & send(a, \lambda\vec{x}C)\\
r(b) & :: & \exists_d (receive(b, d).\exists_{\vec{z}} receive(d, \vec{z}).t(\vec{z}))
\end{array}
$$

These declarations together with the program $\exists_{a,b}(p(a, b) \parallel q(a) \parallel r(b))$, results in communication of channel a through channel b, so that a becomes shared by the three sites. This makes it possible for $send(a, \lambda\vec{x}C)$ to communicate with any of the other sites. This corresponds to the "scope extrusion" on the π-calculus.

Encodings in *PiCO* for *send* and *receive MCC* processes and equivalent processes for the *MCC* definitions are shown in tables 4 and 5.

Table 5. MCC example in *PiCO*.

MCC **process**		*PiCO* **encoding**
$p(a,b)$	$\equiv$	**local** x **in** [**local** i_1 **in** $i_1 \triangleright$ [*assert*: (x) *tell* $x = a$] $\mid b \triangleleft m[i_1]]$ $\mid$ **local** $\vec{y}$ **in** $a \triangleright \left[m\text{: } (i)\ i \triangleleft \mathit{assert}[\vec{y}]\ \textbf{then}\ s(\vec{y})\right]$
$q(a)$	$\equiv$	**local** i_2 **in** $i_2 \triangleright \left[\mathit{assert}(\vec{x})\text{: } \mathit{tell}\ C\right] \mid a \triangleleft m[i_2]$
$r(b)$	$\equiv$	**local** d **in** $[b \triangleright [m\text{: } (i)\ i \triangleleft \mathit{assert}[d]$ **then** **local** $\vec{z}$ **in** $d \triangleright [m\text{: } (i)\ i \triangleleft \mathit{assert}[\vec{z}]$ **then** $t(\vec{z})]]]$

The above *MCC* program is encoded in *PiCO* as

$$\textbf{local}\ a, b\ \textbf{in}\ [\ [\![p(a,b)]\!]_{PiCO}\ \mid\ [\![q(a)]\!]_{PiCO}\ \mid\ [\![r(b)]\!]_{PiCO}]$$

where $[\![p(x)]\!]_{PiCO}$ denotes the *PiCO* encoding for the *MCC* process $p(x)$. Assuming the same order of execution in *MCC* and *PiCO*, it is not hard to see that the same behavior is observed. That is, communication between $b \triangleleft m[i_1]$ and

$$b \triangleright \left[m\text{: } (i)\ i \triangleleft \mathit{assert}[d]\ \textbf{then local}\ \vec{z}\ \textbf{in}\ d \triangleright [m\text{: } (i)\ i \triangleleft \mathit{assert}[\vec{z}]\ \textbf{then}\ t(\vec{z})]\right]$$

yields $d = a$ once the continuation of method m in b is reduced. This results in the sharing of channel a.

6.3. PiCO and the ρ-Calculus

Table 6 shows how the behaviors of constructs in the ρ-calculus can be captured by executing suitable *PiCO* expressions. In this table, reduction of expressions in the ρ-calculus and of configurations in *PiCO* are denoted by rules. The intended meaning is that the expression or configuration in the premise reduces in one or more steps to the expression or configuration in the conclusion. A relation between reduction of expressions in the two calculus can thus be defined as follows:

– Application of an abstraction in the ρ-calculus can be modeled in *PiCO* as a process that sends an appropriated message to a persistent object containing a method that simulates the abstraction.

– The representation of tell operations is straightforward. In Table 6, $\Psi \equiv \phi_1 \wedge \phi_2$.

– The behavior of Cells in the ρ-calculus are captured in *PiCO* by the *cellmaker* object process. Consulting the value of a cell corresponds to sending the message *get* to the created cell object while updating a cell is achieved by sending the message *set* to the cell object. In Table 6 $\mathit{cell}(x,u)$ represents a cell having x as its identifier and u as its value.

– Conditional expressions of the ρ-calculus are modeled by the composition of two *PiCO* *ask* processes. The fist one represents the "then" branch and the second one the "else" branch. The two possible behaviors are shown in table 6.

Table 6. PiCO and *Rho*.

PiCO	**Rho**
$\dfrac{\langle \mathbf{clone} x \lhd [l(\overline{x}):\ E] \mid x \lhd l[\overline{z}];\ \phi \rangle}{\langle \mathbf{clone} x \lhd [l(\overline{x}):\ E] \mid E[\overline{z}/\overline{y}];\ \phi \rangle}$	$\dfrac{\phi \wedge x:\ \overline{y}/E \wedge x'z}{\phi \wedge x:\ \overline{y}/E \wedge E[\overline{z}/\overline{y}]}$
$\dfrac{\langle \mathbf{tell}\phi_2 \mathbf{then} 0;\ \phi_1 \rangle}{\langle 0;\ \Psi \rangle}$	$\dfrac{\phi_1 \wedge \phi_2}{\Psi}$
$\dfrac{\langle newcell(x,y) \mid x \lhd get[v] \ \mathbf{then}\ x \lhd set[u];\ \phi \rangle}{\langle \mathbf{clone}\ cellmaker \rhd [..] \mid cell(x,u) \mid \mathbf{tell}\ v = y \ \mathbf{then}\ 0;\ \phi \rangle}$	$\dfrac{\phi \wedge x:\ y \wedge x'uv}{\phi \wedge x:\ u \wedge v = y}$
$\dfrac{\langle \mathbf{ask}\ \Psi \ \mathbf{then}\ E \mid \mathbf{ask}\ \neg\Psi \ \mathbf{then}\ F;\ \phi \rangle}{\langle E;\ \phi \rangle}$	$\dfrac{\phi \wedge \mathbf{if}\ \Psi \ \mathbf{then}\ E \ \mathbf{else}\ F \ \mathbf{fi}}{\phi \wedge E}$
$\dfrac{\langle \mathbf{ask}\ \Psi \ \mathbf{then}\ E \mid \mathbf{ask}\ \neg\Psi \ \mathbf{then}\ F;\ \phi \rangle}{\langle F;\ \phi \rangle}$	$\dfrac{\phi \wedge \mathbf{if}\ \Psi \ \mathbf{then}\ E \ \mathbf{else}\ F \ \mathbf{fi}}{\phi \wedge F}$

6.4. Future Work

In addition to the formal proof of bisimilarities with related calculi mentioned above, future works include defining suitable constraint systems for different domains in the area of computer aided music composition. We also plan to include constraint guards for instances in the visual formalism of *Cordial* and recasting in *Cordial* the system for harmonic structure generation of *Situation* [3], originally written as a constraint satisfaction tool in *Common Lisp*. We will also be collaborating in the near future with the swedish composer Orjan Sandred, currently doing musical research at IRCAM, in implementing in *Cordial* a system for rhythm structure generation.

7. Conclusions

We defined *PiCO*, a calculus integrating objects and constraints. We did this by adding variables and allowing agents to interact through constraints in a global store. *PiCO* is parameterized in a constraint system and thus independent of a particular domain of application, but has been thought to provide a basis for music composition tools.

Message-object communications is not restricted to equality of sender and receiver channels. The full power of constraints is used instead to restrict implicitly the type of sender channels accepted by an object. We argued that this allows a more natural modeling of situations in which object services are to be provided to collections of requesters. We discussed concrete examples taken advantage of this.

We showed how the operation of *delegation* built into the calculus allows simple representations of inheritance schemes.

We defined the operational semantics of *PiCO* by an equivalence relation and a reduction relation on configurations of an agent and a store.

We described examples showing the generalized mechanism of synchronization of *PiCO* and the transparent interaction of constraints and communicating processes. They also show the possibility to define mutable data and persistent objects in *PiCO*. Finally we show how classes and subclasses with attributes containing partial and mutable information can be easily codified in the calculus.

We implemented in the calculus a non trivial musical example. This example was also implemented in *Cordial*, a high level visual language for music composition. We used this example to illustrate the visual environment of *Cordial* and its close semantic relation to *PiCO*. We argued that the possibilities of constraining every instance of a class to obey user defined properties and of associating particular visualizations to specific instances (such as a score object) should provide more intelligent facilities for musical structures construction and editing. In fact, it is not hard to imagine having in *Cordial* constraints driven score editors in which the space of possible musical material is adapted to the particular needs of a musical piece. We intend to explore this path in the near future.

PiCO is part of an on going research project in programming tools for music composition. The project includes the definition of a visual language with a rigorous semantic model, a visual program editor and a compiler. An abstract machine for *PiCO* has been implemented in *Java* (http://atlas.ujavcali.edu.co/sp/grupos/avispa/software/mapico.html). The current version uses a subset of the finite domain constraints defined in [4]. A β version of the visual editor of *Cordial* and its compiler into (the abstract machine of) *PiCO*, both written in *Java*, can be found in

ftp://atlas.ujavcali.edu.co/pub/grupos/avispa/cordial/

Acknowledgments

We are greatly indebted to the reviewers for their comments and constructive criticism that has helped us to improve this paper significantly. We would also like to thank the colombian science foundation (*Colciencias*) for its financial support, and all composers at IRCAM for providing a unique environment where programming research is motivated by musical thought.

Notes

1. In [11], a more general notion of a constraint system is defined. We follow [13], [12] in taking Predicative Logic as the starting point, so we can rely on well-established intuitions, notions and notations.
2. Objects $(\phi_{\mathbf{sender}}, \delta_{\mathbf{forward}}) \triangleright M$ allow message delegation only if it can be inferred that the original sender does not satisfy the delegation condition represented by $\delta_{\mathbf{forward}}$

References

1. M. Abadi and L. Cardelli (1994). "A Theory of Primitive Objects: Untyped and First-order Systems." *Proceedings of Theoretical Aspects of Computer Software*.
2. G. Assayag and C. Agon (1996). "OpenMusic Architecture." *Proceedings of ICMC '96*. Hong Kong, China.

3. A. Bonnet and C. Rueda (1998). Situation: Un Langage Visuel Basee sur les Contraintes pour la Composition Musicale. *Recherches et Applications en Informatique Musicale*. M. Chemillier and F. Pachet, editors, HERMES, Paris, France.
4. P. Codognet and D. Diaz (1996). "Compiling Constraints in clp(FD)." *J. Logic Programming*, 27(1): 1–199.
5. N. Joachim and M. Müller (1995). "Constraints for Free in Concurrent Computation." *Asian Computing Science Conference*. K. Kanchanasut and J.-J. Lévy, editors, pages 171–186, Pathumthani, Thailand.
6. R. Milner (1991). "The Polyadic π-Calculus: A Tutorial." Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science. Also in *Logic and Algebra of Specification*, F. L. Bauer, W. Brauer and H. Schwichtenberg, editors, Springer Verlag, (1993).
7. F. Pachet and P. Roy (1995). "Mixing Constraints and Objects: A Case Study in Automatic Harmonization." *Proceedings of TOOLS Europe'95*. pages 119–126, Versailles, France.
8. L. Quesada, C. Rueda, and G. Tamura (1997). "The Visual Model of Cordial." *Proceedings of the CLEI97*. Valparaiso, Chile.
9. J. P. R. Milner and D. Walker (1992). "A Calculus of Mobile Processes, Parts I and II." *Journal of Information and Computation*, 100: 1–77.
10. J. Réty (1997). "Un Langage Distribué Concurrent Avec Contraintes." JFPLC-97.
11. V. A. Saraswat (1993). *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA.
12. G. Smolka (1994a). "A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards." Research Report RR-94-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany.
13. G. Smolka (1994b). "A Foundation for Higher-order Concurrent Constraint Programming." *1st International Conference on Constraints in Computational Logics*. J.-P. Jouannaud, editor, pages 50–72, München, Germany.
14. D. N. Turner (1995). "The Polymorphic Pi-Calculus: Theory and Implementation." Ph.D. Thesis, Laboratory for Foundations of Computer Science.
15. F. Valencia, J. F. Diaz, and C. Rueda (1997). "The π^+-Calculus." *Proceedings of the CLEI97*. Valparaiso, Chile.
16. V. T. Vasconcelos (1994). "Typed Concurrent Objects." *Proc. of 8th European Conference on Object-Oriented Programming (ECOOP'94)*, M. Tokoro and R. Pareschi, editors, volume 821 of Lecture Notes in Computer Science, pages 100–117.
17. D. Walker (1995). "Objects in the π-Calculus." *Journal of Information and Computation*, 116(2): 253–271.