PiCO: A Calculus of Concurrent Constraint Objects for Musical Applications. [1]

Gloria Alvarez [2] and **Juan Francisco Diaz** [3] and **Luis O. Quesada** [4] and **Frank D. Valencia** [5] and **Gerard Assayag** [6] and **Camilo Rueda** [7]

Abstract. Musical applications are very demanding on the expressive power of the underlined programming model. We propose PiCO, a calculus integrating concurrent objects and constraints as a base for music composition tools. In contrast with calculi such as [JM95, RMW92] or TyCO [Vas94], both constraints and objects are primitive notions in PiCO. The object model is extended with constraints by orthogonally adding the notion of *constraint system* found in the ρ-calculus [Smo94a]. The extended calculus provides a natural way to express more sophisticated communication behaviors via the standard message-passing synchronization mechanism. Moreover, it allows us to represent complex partially defined objects such as musical structures in a compact way. The paper includes encodings in PiCO of the notions of class and subclass. We illustrate the transparent interaction of constraints and objects by a musical example involving harmonic and temporal relations.

Keywords: Concurrent Programming, Constraint Programming, Concurrent Constraint objects, TyCO, PiCO, Formal Calculi, Mobile Processes

1 Introduction

- Why do we want Concurrent Objects with Constraints ?

Our objective is to develop computational models adapted 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) or *fundamentals*, 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 can follow complex amplitude evolutions or such as *melody*, where patterns of distances relate 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* [BR98] constraints and CLOS objects can be used to define complex musical structures. In the same spirit, but more closely integrated to the underlying Smalltalk language, *Backtalk* [PR95] provides a framework for handling constraint satisfaction within an object environment. Both systems have been successfully used in real world 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 communicability 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 [VDR97] the π-calculus was extended with the notion of constraint. In this paper we consider the addition of objects and describe encodings in it of basic notions of classes and subclassing.

Several concurrent objects calculi have been proposed recently ([Wal95], [Vas94], [AC94]). 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 by l is sent to

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

[2] **Pontificia Universidad Javeriana Cali, Colombia**. E-mail:galvarez@atlas.ujavcali.edu.co.

[3] **Universidad del Valle, Colombia**. E-mail: jdiaz@borabora.univalle.edu.co.

[4] **Pontificia Universidad Javeriana Cali, Colombia**. E-mail: lquesada@atlas.ujavcali.edu.co.

[5] **Pontificia Universidad Javeriana Cali, Colombia**. E-mail: fvalenci@atlas.ujavcali.edu.co.

[6] **IRCAM, Paris** E-mail: assayag@ircam.fr

[7] **Pontificia Universidad Javeriana Cali, Colombia** and **IRCAM, Paris** E-mail: crueda@ircam.fr, crueda@atlas.ujavcali.edu.co

Submitted to ECAI 98
April 27, 1998

an object located at a or, more generally, until a message is sent to an object located at x, with $x = a$.

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 ([Sar93]). 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 ([JM95]), 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 underlined model of interaction. The powerful constraints calculus of Oz allows other programming models (functional, objects) to coexists through syntactic encodings of those models within Oz.

Our approach is different. We want to maintain as much as possible the independence of the object and constraint models at the calculus level. Firstly, we think that this better reflects the two actual main approaches composers use for constructing musical structures. Secondly, we would like our tool to be easily adaptable to different notions of object and different constraint systems.

Consider an example. Let us suppose that we want to define some conditions for the location of an object. For instance $?(x \in \{a, b, c\}).x \triangleright [l : (\widetilde{y})P]$ can be seen as a process P which is suspended until a message is sent to a location x where x is either a,b or c. This can also be interpreted as an object capable of receiving through x messages from three different locations. Of course, for this simple example, the π-calculus process $a?[\widetilde{y}].P + b?[\widetilde{y}].P + c?[\widetilde{y}].P$ [Mil91] could very well be used for this. 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. For example:

- $!(x \in \{a_1, a_2, \ldots\}).x \triangleleft l : (\widetilde{J})$ means "Tell message $x \triangleleft l : (\widetilde{J})$ that can only be sent to objects identified by $a_1, a_2, \ldots$".
- $*x \triangleright [l : (\widetilde{y})?(x \in \{a, b\}).P]$ means "Execute process (method) P only if a message is sent to a location labeled by either a or b".

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 [Mil91] and second, by using constraints to "change" the object state by refining the partial knowledge one has of the attributes of the object.

These views are complementary. In a musical setting both are typically used. Composers may very well conceive musical processes 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 consistency with the partial information implied by the rules.

We give further bellow an example of an object environment showing the transparent and useful integration of constraints and concurrent processes in a real music composition problem.

In sections 2 and 3 we present the syntax and semantics of PiCO. The syntax of PiCO adds constraint processes to the standard TyCO process. 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 [Sar93]. Recursive definitions are shown in section 4. Recursive definitions are not considered primitives in PiCO, since they can easily be encoded as it was done in [Mil91]. Section 5 shows how classes and mutable objects carrying partial information on their attributes can be conveniently represented in PiCO. In section 6 we show a somewhat elaborate musical example. Section 7 shows briefly how the semantics for a concurrent constraint visual language can be expressed using PiCO. Finally, section 8 gives some conclusions.

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.

A process $I \triangleright [l_1 : (\widetilde{x_1})P_1 \& \ldots \& l_m : (\widetilde{x_m})P_m]$ can be thought of as an object located at I (named I or identified by I) whose methods $(\widetilde{x_1})P_1 \ldots (\widetilde{x_m})P_m$ are labeled by a set of pairwise distinct labels $l_1 \ldots l_m$. In a method $l : (\widetilde{x})P$, $\widetilde{x}$ represents the formal parameters and P the body of the method. Other than *names*, *variables* and *primitive values* can be used as object identifiers.

A process $I \triangleleft l : [\widetilde{J}].P$ can be thought of as a message to *target* object located at I 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.

The summation form $N_1 + N_2$ represents a process able to take part in one -but only one - of two alternatives for execution. The choice of one alternative precludes the other. The null process 0 is the process doing nothing.

The process $(\nu a)P$ restricts the use of the name a to P. Another way to describe this is that $(\nu a)P$ declares a new unique name a, distinct from all external names, to be used in P. Similarly, $(\nu x)P$ (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. $*P$ (*replication*) means $P \mid P \mid \ldots$(as many copies as needed). A common instance of replication is $*I \triangleright M$, an object which can be reproduced when a requester communicates via I. Replication is often used for encoding recursive process definitions (see [Mil91]).

Finally, Constraint processes are new kinds of processes whose behavior depends on a global *store*. A store contains information given by constraints. The store is used in PiCO to control all potential communications. The Tell process $!\phi.P$ means "Add ϕ to the store and then activate P.". Thus, Tell processes will be used to influence the behavior of other processes. The Ask process $?\phi.P$ 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 $?\phi.P$ until the store contains enough information to run it.".

In what follows, we write $(\nu I_1, I_2, \ldots, I_n)$ instead of $(\nu I_1)(\nu I_2)\ldots(\nu I_n)$ and we omit $.O$ when no confusion arises.

Normal Processes: N	::=	$I \triangleleft m.P$	Message to I
	|	$I \triangleright M$	Object I
	|	$N_1 + N_2$	Indeterminate execution
	|	O	Inaction or null process
Constraint processes: R	::=	$!\phi.P$	Tell process
	|	$?\phi.P$	Ask process
Processes: P, Q	::=	$(\upsilon x)P$	New variables x in P
	|	$(\upsilon a)P$	New name a in P
	|	N	Normal process
	|	$P \mid Q$	Composition
	|	$*P$	Replicated process
	|	R	Constraint process
Object identifiers: I, J	::=	a	Name
	|	v	Value
	|	x	Variable
Collection of Methods M	::=	$[l_1 : (\widetilde{x_1})P_1 \& \ldots \& l_m : (\widetilde{x_m})P_m]$	
Messages m	::=	$l : [\widetilde{I}]$	

Table 1. PiCO syntax

3 Operational Semantics

3.1 Constraint System

PiCO is parameterized in a *Constraint System*. For our purposes it will suffice to found the notion of constraint system on first-order Predicate Logic, as it was done in [Smo94b, Smo94a] [8].

A Constraint System consists of [Smo94b, Smo94a]:

- A signature Σ (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 and used as object identifiers.
- A consistent theory Δ (a set of sentences over Σ having a model) satisfying two conditions:
  1. $\Delta \models \neg(a = b)$ for every two distinct names a, b.
  2. $\Delta \models \phi \leftrightarrow \psi$ for every two sentences ϕ, ψ over Σ such that ψ can be obtained ¿from ϕ by permutation of names.

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, symbols $\phi, \psi, \ldots$ denote first-order formulae in Σ, henceforth called *constraints*. We say that ϕ *entails* ψ in Δ, written $\phi \models_\Delta \psi$, iff $\phi \rightarrow \psi$ is true in all models of Δ. We say that ϕ is *equivalent* to ψ in Δ, written $\phi \models\mid_\Delta \psi$, iff $\phi \models_\Delta \psi$ and $\psi \models_\Delta \phi$. We say that ϕ is *satisfiable* in Δ iff $\phi \not\models_\Delta \bot$. 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.

As 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 an bound variables in ϕ, respectively. Finally, $fn(\phi) \subset \mathcal{N}$ is the set of names appearing in ϕ.

As we said before, constraint processes act relative to a store. A store is defined in terms of the underlined constraint system:

Definition 3.1 (Store) *A store $S = \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_r$ (with $r \geq 0$) is a constraint in Σ. When $r = 0, S$ is said to be the empty store (i.e., $S = \top$). When $S \models_\Delta \bot$, S is said to be the unsatisfiable store.*

The operational semantics of PiCO will be defined 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. A configuration is a tuple $\langle P; S \rangle$ consisting of a process P and a store S.

3.2 Structural Congruence and equivalence on configurations

We identify first the binding operators in PiCO: The binding operator for names is $(\upsilon a)P$ which declares a new name a in P. There are only two binding operators for variables: $(\upsilon x)P$ which binds x in P and $(x_1 \ldots x_n).P$ which declares formal parameters $x_1, \ldots, x_n$ in P. So we can define *free names*

[8] In [Sar93], a more general notion of a constraint system is defined. We follow [Smo94b, Smo94a] in taking Predicate Logic as the starting point, so we can rely on well-established intuitions, notions and notations.

$fn(P)$, *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 $fn(P) \cup bn(P)$.

We define structural congruence for PiCO much in the same way as is done for the π-calculus in [Mil91].

Definition 3.2 (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* (α *– conversion).*
- $(\mathcal{P}/\equiv, |, O)$ *is a symmetric monoid, where* $\mathcal{P}$ *is the set of processes.*
- $(\mathcal{NP}/\equiv, +, O)$ *is a symmetric monoid, where* $\mathcal{NP}$ *is the set of normal processes.*
- $I \triangleright M \equiv I \triangleright M'$ *if* M' *is a permutation of* M.
- $*P \equiv P \mid *P$.
- $(va)O \equiv O$, $(vx)O \equiv O$, $(va)(vb)P \equiv (vb)(va)P$, $(vx)(vy)P \equiv (vy)(vx)P$, $(va)(vx)P \equiv (vx)(va)P$.
- *If* $a \notin fn(P)$ *then* $(va)(P \mid Q) \equiv P \mid (va)Q$.
- *If* $x \notin fv(P)$ *then* $(vx)(P \mid Q) \equiv P \mid (vx)Q$.
- *If* $\phi \models\!\!\models_\Delta \psi$ *and* $P \equiv Q$ *then* $!\phi.P \equiv !\psi.Q$ *and* $?\phi.P \equiv ?\psi.Q$

Definition 3.3 (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.

3.3 Reduction relation

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

COMM describes the result of the interaction between message $I \triangleleft l_i : [\widetilde{J}].Q$ and object $I' \triangleright [l_1 : (\widetilde{x_1})P_1, \ldots, l_m : (\widetilde{x_m})P_m]$. The store is used to decide whether the object is indeed the target of the message. Process $P_i\{\widetilde{J}/\widetilde{x_i}\}$ is obtained by replacing, in parallel, every free occurrence of variables from $\widetilde{x_i}$ by identifiers (i.e., values, names or variables) from $\widetilde{J}$, respectively. Notice that Q is activated whereas the remaining normal processes, N_1 and N_2 are discarded.

The ASK and TELL rules describe the interaction between constraint processes and the store. TELL gives the way of adding information to the store. $!\phi.P$ adds the 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 [Sar93]. For example, agent $!(x = a).P$ informs messages of the form $x \triangleleft m$ that their target object is now located at 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 $?(x \in \{a, b\}).x \triangleleft m$ is able to send m to x, just when x is either object a or object 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. 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. From here on $S \gg \{I_1, \ldots, I_n\}$ will denote store $S \wedge (I_1 = I_1) \wedge \ldots \wedge (I_n = I_n)$. Intuitively, $S \gg \{x\}$ (i.e. $S \wedge x = x$) denotes the addition of variable x to store S. Thus, any variable $x \notin fv(S)$ added to the store by $S \gg \{x\}$ will not be used in subsequent declarations. If it happens that $x \in fv(S)$, we can rename x with a new variable $z \notin fv(S) \cup fv(P)$ by using the first item of Definition 3.2 (i.e. $(vx)P \equiv (vz)P\{z/x\}$ *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.

In what follows, $\Longrightarrow$ will denote the reflexive and transitive closure of $\longrightarrow$. Finally, we will say that $\langle P'; S'\rangle$ is a *derivative* of $\langle P; S\rangle$ iff $\langle P; S\rangle \Longrightarrow \langle P'; S'\rangle$.

Runtime failure. In the cc-model [Sar93], the invariant property of the store is that it is satisfiable. This can be made to hold in PiCO by defining transitions from $\langle !\phi.P; S\rangle$ 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 [Tur95], without affecting any of our results.

Potentiality of reduction. Whenever we augment the store, we may increase the potentiality of 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.

Proposition 3.4 *If* $S_2 \models_\Delta S_1$ *and* $\langle P_1; S_1\rangle \longrightarrow \langle P_2; S_1'\rangle$ *then* $\langle P_1; S_2\rangle \longrightarrow \langle P_2; S_2'\rangle$ *and* $S_2' \models_\Delta S_1'$.

Proof: Straightforward from rules TELL, COMM and ASK:

1. *Transitions using* ASK *or* COMM: Since $S_2 \models_\Delta S_1$, for any constraint ϕ such that $S_1 \models_\Delta \phi$ we have $S_2 \models_\Delta \phi$. Thus, any P_2 obtained by using ASK or COMM in $\langle P_1; S_2\rangle$ can be obtained by using the same rule for $\langle P_1; S_2\rangle$. Neither ASK nor COMM modify the store, therefore $S_2' \models_\Delta S_1'$.
2. *Transitions using* TELL: TELL modifies the store. In this case $P_1 \equiv !\phi.Q$. For the transition $\langle !\phi.Q; S_1\rangle \longrightarrow \langle Q; S_1 \wedge \phi\rangle$ we have $\langle !\phi.Q; S_2\rangle \longrightarrow \langle Q; S_2 \wedge \phi\rangle$. Thus, $S_2' \models_\Delta S_1'$.
3. *Transitions using* EQUIV: If $\langle P_1; S_1\rangle \equiv_P \langle P_3; S_3\rangle$ and $\langle P_3; S_3\rangle \longrightarrow \langle P_3'; S_3'\rangle$, where $\langle P_3'; S_3'\rangle \equiv_P \langle P_2; S_1'\rangle$, then from 3.3 we have $S_2 \models_\Delta S_3$ and $P_1 \equiv P_3$. The desired result is obtained by applying inductively items 1,2,3,4,5.
4. *Transitions using* DEC-V *or* DEC-N: In this case $P_1 \equiv_P (vx)Q$. Suppose first that $x \notin fv(S_2)$. If $x \notin fv(S_1)$ we have $S_2 \gg \{x\} \models_\Delta S_1 \gg \{x\}$. If $x \in fv(S_1)$, then using 3.2 (i.e. $(vx)Q \equiv (vz)Q\{z/x\}$ *if* $z \notin fv(Q)$, x was replaced in P_1 by a new variable, z, such that $z \notin fv(S_1)$. For any z we have $S_2 \gg \{x\} \models_\Delta S_1 \gg \{z\}$. Similarly, if $x \in fv(S_2)$

$$\text{COMM: } \frac{S|=_\Delta I=I' \qquad 1\leq i\leq m \qquad |\widetilde{J}|=|\widetilde{x_i}|}{\left\langle N_1+I\triangleleft l_i:[\widetilde{J}].Q \mid N_2+I'\triangleright[l_1:(\widetilde{x_1})P_1,\ldots,l_m:(\widetilde{x_m})P_m];S\right\rangle \longrightarrow \left\langle Q \mid P_i\left\{\widetilde{J}/\widetilde{x_i}\right\};S\right\rangle}$$

$$\text{TELL: } \langle !\phi.P;S\rangle \longrightarrow \langle P;S\wedge\phi\rangle$$

$$\text{ASK: } \frac{S\models_\Delta \phi}{\langle ?\phi.P;S\rangle \longrightarrow \langle P;S\rangle} \quad , \quad \frac{S\models_\Delta \neg\phi}{\langle ?\phi.P;S\rangle \longrightarrow \langle O;S\rangle}$$

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

$$\text{DEC-V: } \frac{x\notin fv(S),\ \langle P;S\gg\{x\}\rangle \longrightarrow \left\langle P';S'\right\rangle}{\langle (\upsilon x)P;S\rangle \longrightarrow \langle P';S'\rangle} \quad , \quad \text{DEC-N: } \frac{a\notin fn(S),\ \langle P;S\gg\{a\}\rangle \longrightarrow \left\langle P';S'\right\rangle}{\langle (\upsilon a)P;S\rangle \longrightarrow \langle P';S'\rangle}$$

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

Table 2. Transition system

it is easy to see that for any y, $S_2 \gg \{y\} \models_\Delta S_1$ and for any z, $S_2 \gg \{x\} \models_\Delta S_1 \gg \{z\}$. Thus, the desired result is obtained by applying inductively items 1,2,3,4,5.

5. *Transitions using* PAR: PAR does not consider the store (directly) as a premise, therefore the desired result is obtained by applying inductively items 1,2,3,4,5.

□

In the following example we describe the behavior of a simple process.

Example 3.5 *Process P_1 sends a message to object r. The contents of the message is the greater of two numbers x and y, which is received by method Q_1. Let Δ be the set of all valid sentences in the rational numbers and Σ the corresponding symbols.*

$P_1 \equiv (\upsilon x)P_2$; $P_2 \equiv (\upsilon y)P_3$; $P_3 \equiv (\upsilon r)P_4$; $P_4 \equiv (r \triangleright [l_1 : (z)Q_1 \& l_2 : (z)Q_2] \mid ?(x > y).r \triangleleft l_1 : [x] \mid ?\neg(x > y).r \triangleleft l_1 : [y] \mid !(x = y+1))$.

Since the variable declarations are different, by DEC-V, the derivatives of $\langle P_1;\top\rangle$ are the derivatives of $\langle P_2;\top\gg\{x\}\rangle$, whose derivatives are the derivatives of $\langle P_2;\top\gg\{x,y\}\rangle$. By DEC-N ($r$ denotes a name) the derivatives of $\langle P_2;\top\gg\{x,y\}\rangle$ are the derivatives of $\langle P_4;\top\gg\{x,y,r\}\rangle$, if any. The Ask agents in P_4 are suspended by $\top\gg\{x,y,r\}$, and there is no other agent sending along channel r, so we can only reduce $\langle P_4;\top\gg\{x,y,r\}\rangle$ by applying TELL combined with PAR and EQUIV. Thus,

$\langle P_4;\top\gg\{x,y,r\}\rangle$

$\longrightarrow \langle (r \triangleright [l_1 : (z)Q_1 \& l_2 : (z)Q_2] \mid ?(x > y).r \triangleleft l_1 : [x] \mid$
$?\neg(x > y).r \triangleleft l_1 : [y] \mid 0); (\top \gg \{x,y,r\}) \wedge x = y+1\rangle$

Now using ASK combined with PAR and EQUIV,
$\longrightarrow \langle (r \triangleright [l_1 : (z)Q_1 \& l_2 : (z)Q_2] \mid r \triangleleft l_1 : [x] \mid$
$?\neg(x > y).r \triangleleft l_1 : [y] \mid 0); (\top \gg \{x,y,r\}) \wedge x = y+1\rangle$

We can eliminate the null process by using $\equiv_P$,
$\equiv_P \langle (r \triangleright [l_1 : (z)Q_1 \& l_2 : (z)Q_2] \mid r \triangleleft l_1 : [x] \mid$
$?\neg(x > y).r \triangleleft l_1 : [y]); (\top \gg \{x,y,r\}) \wedge x = y+1\rangle$

Using ASK combined with PAR,
$\longrightarrow \langle (r \triangleright [l_1 : (z)Q_1 \& l_2 : (z)Q_2] \mid r \triangleleft l_1 : [x] \mid 0);$
$(\top \gg \{x,y,r\}) \wedge x = y+1\rangle$

Using $\equiv_P$ we can rewrite the processes so they can have the correct format for COMM, and then eliminate the null process,
$\equiv_P \langle ((r \triangleright [l_1 : (z)Q_1 \& l_2 : (z)Q_2] + 0) \mid (r \triangleleft l_1 : [x] + 0));$
$(\top \gg \{x,y,r\}) \wedge x = y+1\rangle$

Finally, applying COMM and $\equiv_P$,
$\longrightarrow \langle (Q_1\{x/z\} \mid 0); (\top \gg \{x,y,r\}) \wedge x = y+1\rangle$
$\equiv_P \langle Q_1\{x/z\}; (\top \gg \{x,y,r\}) \wedge x = y+1\rangle$

Thus, $\langle P_1;\top\rangle \longrightarrow \langle Q_1\{x/z\}; (\top \gg \{x,y,r\}) \wedge x = y+1\rangle$.

Behavioral equivalence. In [VDR97] a reduction equivalence relation for an extension of the π-calculus with constraints was defined. This relation equates configurations whose agents can communicate on the same channels at each transition. For each process identifier I and label l, this is expressed by means of an observation predicate $\downarrow^S_{I,l}$ 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 [Smo94b]. Names, conveniently used, provide a unique reference to concurrent objects which can be used for data encapsulation as in [Tur95]. Names and Variables in PiCO are different entities because of the presence of constraints.

The following example illustrates this difference. Let $P_1 \equiv (\upsilon x)(\upsilon y)(?\neg(x = y).Q)$ and $P_2 \equiv (\upsilon a)(\upsilon b)(?\neg(a = b).Q)$ and let Δ be the set of sentences valid in the natural numbers. It is easy to see that $\langle P_2;\top\rangle \longrightarrow \langle Q;\top\gg\{a,b\}\rangle$. However, since $?\neg(x = y).Q$ is suspended by $\top\gg\{x,y\}$, there is no reduction for $\langle P_1;\top\rangle$.

4 Recursive Definitions

It is often convenient to define processes recursively. Recursively-defined processes have the form $D(x_1,\ldots,x_n) \stackrel{def}{=} P$, where P may contain occurrences of D (perhaps with different arguments), $fv(P) \subseteq \{x_1,\ldots,x_n\}$ and $fn(P) = \emptyset$. When no confusion arises we write D (without arguments) instead of $D(x_1,\ldots,x_n) \stackrel{def}{=} P$.

"Definition-making" and its applications are not primitives, since they can easily be encoded by means of replication [Mil91]. In some other calculi (e.g. Most versions of TyCO), however, recursion is primitive [Vas94]. Keeping in mind that recursion can be simulated with the standard syntax of PiCO, we will extend the syntax with recursive process definitions and their invocations, (i.e. $P := \ldots \mid D(x_1,\ldots,x_n) \stackrel{def}{=} P \mid D(y_1,\ldots,y_n).Q$) and assume there is a transition rule:

APPLY: $\left\langle D(x_1,\ldots,x_n) \stackrel{def}{=} P \mid D(y_1,\ldots,y_n).Q; S \right\rangle \longrightarrow \left\langle D(x_1,\ldots,x_n) \stackrel{def}{=} P \mid P\{y_1,\ldots,y_n/x_1,\ldots,x_n\} \mid Q; S \right\rangle$

APPLY denotes the usual result of applying $y_1,\ldots,y_n$ to definition D. Definition D remains in the derivatives and the result of the invocation is obtained by replacing in P the formal arguments by the actual parameters and by activing its continuation Q. Symbol D is assumed to be included in Σ.

5 Classes of Objects

In this section we show how classes and object attributes can be encoded in PiCO by using recursive definitions and constraints. Attributes are represented in PiCO by variables containing partial information given by tell operations. Much like in ([Smo94b]) classes can be viewed as object generators giving initial conditions or *class constraints* over their attributes. Class constraints are conditions which must be satisfied regardless the attributes' changes and thus define the fundamental properties of instances belonging to the class. From this point of view Classes can be codified as:

$< ClassName > (x_0, x_1, \ldots, x_n) \stackrel{def}{=} !\phi[x_1,\ldots,x_n].x_0 \triangleright M$

where $< ClassName >$ denotes the name (or identifier) for the class given in Σ, x_0 the object identifier (or $self$), and $x_1,\ldots,x_n$ the object attributes. The operation $!\phi[x_1,\ldots,x_n]$ imposes on each object of the class a constraint ϕ over its attributes. Methods in M can be arbitrarily defined. In order to ensure persistence and mutability each method should contain a recursive call to $< ClassName >$. Special methods to read and update attributes should also be present. Thus M should have the form:

$$
\begin{array}{l}
[update_{<ClassName>} : (y_1,\ldots,y_n) \\
\quad < ClassName > (x_0, y_1, \ldots, y_n)\& \\
read_{<ClassName>} : (z)z \triangleleft receive : [x_1 \ldots x_n]. \\
\quad < ClassName > (x_0, x_1, \ldots, x_n)\& \\
dispose_{<ClassName>} : ()0\& \\
\ldots]
\end{array}
$$

An object can be updated by using the method $update_{<ClassName>}$ which receives new attributes. these will be constrained by the initial condition of the Class. Using method $read_{<ClassName>}$ the object can report its current state to a requester object. The latter should be able to respond to the appropriate *receive* message. A method $dispose_{<ClassName>}$ reduces to the null process thus terminating the object's existence.

Example 5.1 *In this example the creation of a simple object in PiCO is illustrated. A "Estrada" chord contains three intervals (expressed in number of semitones between consecutive notes) whose sum modulo 12 is equal to zero. To represent the intervals of a chord we assume a constraint system over suitable finite domains of integers. Class E, of "Estrada" chords is defined as follows.*

$$
\begin{array}{l}
E(x, x_1, x_2, x_3) \stackrel{def}{=} \\
!((x_1 + x_2 + x_3)\ mod\ 12 = 0).x \triangleright [\ update_E : (y_h, y_{c1}, y_{c2}) \\
\qquad\qquad E(x, y_h, y_{c1}, y_{c2})\& \\
\qquad\qquad read_E : (z)z \triangleleft receive : [x_1, x_2, x_3]. \\
\qquad\qquad E(x, x_1, x_2, x_3)\& \\
\qquad\qquad dispose_E : ()0]
\end{array}
$$

Where x_1, x_2, x_3 denote the intervals between consecutive notes in the chord. Any update of an object belonging to class E must satisfy the class constraint $((x_1+x_2+x_3)\ mod\ 12 = 0)$

An instance c of class E is created and then sent a message setting one of its intervals to 4 semitones. Another interval is constrained to be bigger than one semitone.

$P \equiv E \mid (vc, x_1, x_2, x_3)(\ E(t, x_1, x_2, x_3) \mid c \triangleleft update_E : [x_1, x_2, 4] \mid !(x_2 > 1)\ )$

Reduction gives:

$$
\begin{array}{l}
\langle P; \top \rangle \Longrightarrow \langle E \mid c \triangleright [update_E : (y_1, y_2, y_3)E(x, y_1, y_2, y_3)\& \\
\qquad\qquad read_E : (z)z \triangleleft receive : [x_1, x_2, x_3]. \\
\qquad\qquad\quad E(x, x_1, x_2, x_3)\& \\
\qquad\qquad dispose_E : ()0] \\
\qquad \mid c \triangleleft update_E : [x_1, x_2, 4] \mid !(x_2 > 1) \\
\qquad ; \\
\qquad (\top \gg \{c, x_1, x_2, x_3\}) \wedge (x_1 + x_2 + x_3)\ mod\ 12 = 0\rangle \\
\\
\Longrightarrow \langle E \mid c \triangleright [update_E : (y_1, y_2, y_3)E(x, y_1, y_2, y_3)\& \\
\qquad\qquad read_E : (z)z \triangleleft receive : [x_1, x_2, 4]. \\
\qquad\qquad\quad E(x, x_1, x_2, 4)\& \\
\qquad\qquad dispose_E : ()0] \\
\qquad ; \\
\qquad (\top \gg \{c, x_1, x_2, x_3\}) \wedge (x_1 + x_2 + x_3)\ mod\ 12 = 0 \wedge \\
\qquad (x_1 + x_2 + 4)\ mod\ 12 = 0 \wedge x_2 > 1)\rangle
\end{array}
$$

Chord c has now one interval fixed to value 4. Changing this value again can only be done by "ignoring" the current constraints (except the class constraint). This can be done in this simple example by using new local variables, as follows:

$Q \equiv E \mid (vc, x_1, x_2, x_3)(\ E(c, x_1, x_3, x_3) | c \triangleleft update_E : [x_1, x_3, 4].(vy_1, y_2)(c \triangleleft update_E : [y_1, y_3, 3]) | !(x_2 > 1)\ )$

5.1 Sub-classing

A very simple notion of sub-classing can be defined as providing extensions of a predefined class or (subclass). The extension involves new methods and new attributes. No method overriding is considered. Subclasses may be defined as:

$$
\begin{array}{l}
< SubClassName > (x_0, \widetilde{x}, \widetilde{y}) \stackrel{def}{=} !\phi[\widetilde{x}, \widetilde{y}].(\ x_0 \triangleright M \mid \\
\qquad\qquad < SuperClassName > (x_o, \widetilde{x})\ )
\end{array}
$$

Where $< SubClassName >$ represents some name in Σ identifying the subclass, $< SuperClassName >$ is the name in Σ for the superclass, M represents the collection of new methods and $\widetilde{y}$ represents the new attributes. The expression $!\phi[\widetilde{x},\widetilde{y}]$ constraints new and old attributes. As before, M should have special methods to read, update and destroy the object. A subclass instance is represented as a composition of several PiCO objects identified by the same name. One of them represents the object containing the new methods and attributes (M and $\tilde{y}$). Each one of the others contains the methods of a class in the inheritance hierarchy. That is, an instance x of a subclass of a class is represented by two objects both identified by x, an instance of a subclass of this subclass by three objects and so on. The collection of new methods M should be defined as follows:

$$
\begin{array}{l}
[update_{<SubClassName>} : (\widetilde{x'},\widetilde{y'}).\\
\quad x_0 \triangleleft dispose_{<SuperClassName>} : [].\\
\quad < SubClassName > (x_0,\widetilde{x'},\widetilde{y'})\&\\
read_{<SubClassName>} : (z)z \triangleleft receive : [\widetilde{x},\widetilde{y}].\\
\quad dispose_{<SuperClassName>} : [].\\
\quad < SubClassName > (x_0,\widetilde{x},\widetilde{y})\&\\
dispose_{<SubClassName>} : ()dispose_{<SuperClassName>} : []\\
\&\ldots]
\end{array}
$$

Method invocations in the subclass disposes of the object in its superclass to preserve the right number of objects in the representation of a subclass.

Example 5.2 *We extend the previously defined class E to define a subclass EF of "Estrada" chords. EF chords are restricted to an octave and have one of the intervals equal to a fifth (7 semitones). EF chords contain an additional attribute, say x_4, representing the pitch of the base note of the chord. This pitch is constrained to a certain scale expressed as a set of integers (say integer 60 corresponds to middle C). Subclass EF is defined as:*

$$
\begin{array}{l}
EF(x_0,x_1,x_2,x_3,x_4) \stackrel{def}{=}\\
\quad !(max(x_1,x_2,x_3) = 7 \wedge x_4 \in \{60,62,64,65,67\}).\\
\quad\quad (x_0 \triangleright [play_{EF} : (y)Q.dispose_E : [].EF(x_0,x_1,x_2,x_3,x_4)\&\\
\quad\quad\quad update_{EF} : (y_1,y_2,y_3,y_4).x_0 \triangleleft dispose_E : [].\\
\quad\quad\quad\quad EF(x_0,y_1,y_2,y_3,y_4)\&\\
\quad\quad\quad read_{EF} : (z)z \triangleleft receive : [x_1,x_2,x_3,x_4].dispose_E : [].\\
\quad\quad\quad\quad EF(x_0,x_1,x_2,x_3,x_4)\&\\
\quad\quad\quad dispose_{EF} : ()O]\\
\quad\quad | \ E(x_0,x_1,x_2,x_3))
\end{array}
$$

6 Using PiCO in real world problems: A music problem

We discuss below a simple example which illustrates the kind of musical structures that PiCO should be able to handle. The idea is 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.

The two melodic voices, $Voice_1$ and $Voice_2$, will start at the same time and will be generated until a external condition is met. Notes in the two voices have three attributes: *pitch*,*duration* and *dynamics*. Their pitch values should be in a set of allowed ambitus Amb, their durations must belong to a given set, say, $\{1/2, 1/4, 1/8\}$ (where 1/4 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 $!max(x,y) - min(x,y) \in Melody_i$, where $Melody_i$ is a given set of integers.
2. Notes are divided into groups of duration equal to 1/2. 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 127. Others 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 1/8 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 $HARMONSET2$.

The first issue is to define a suitable constraint system for musical applications. finite domains (FD) systems ([CD96]) 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. To solve the above problem the constraint system is thus defined over integer and rational arithmetic over some suitable finite domain (for *pitch*, *duration* and the like). Integers will be used as objects identifiers. For the sake of simplicity expressions E such as $i+1$, $r1-x2$ and the like, will denote a new variable, say z, where $z = E$. For instance, message $(i+1)\triangleleft : pitch[z_1]$ is a shorthand for the process $(vz)(\ !(z = i+1) \mid z \triangleleft : pitch[z_1]\ )$

Note is a persistent *PiCO* object having *self* as identifier with attributes *pitch*,*dur* and *dyn*. For simplicity, the notes of $Voice_1$ use positive integers as identifiers: 1 the first one, 2 the second one and so on. Lower notes $Voice_2$ use negative integers: -1,-2 and so on. Whenever a note is created, its attributes are constrained by class constraint $(pitch \in Ambitus \wedge dur \in \{1/2, 1/4, 1/8\})$. It has methods for reporting its attributes, others methods may be defined for updating its attributes.

$$
\begin{array}{l}
Note(self,pitch,dur,dyn) \stackrel{def}{=}\\
!(pitch \in Ambitus \wedge dur \in \{1/2,1/4,1/8\}).\\
self \triangleright [pitch : (p)!(p = pitch).Note(self,pitch,dur,dyn)\&\\
\quad\quad duration : (d)!(d = dur).Note(self,pitch,dur,dyn)\&\\
\quad\quad dynamics : (d)!(d = dyn).Note(self,pitch,dur,dyn)\&\\
\quad\quad \ldots]
\end{array}
$$

Group is defined similarly. Group objects use positive integers as identifiers: 0 the first one, 1 the second one, and so on. A group object has four attributes: fn_1: its first note in $Voice_1$, ln_1: its last note in $Voice_1$, fn_2: its first note in $Voice_2$, ln_2: its last note in $Voice_2$. So, $fn_1, fn_1+1, \ldots, ln_1-1, ln_1$ and $fn_2, fn_2-1, \ldots, ln_2+1, ln_2$ are the notes of the group. Class constraint $(ln_1 \geq fn_1 > 0 \wedge ln_2 \leq fn_2 < 0)$ specifies allowed values for its attributes.

$Group(self, fn_1, ln_1, fn_2, ln_2) \stackrel{def}{=}$
$!(ln_1 \geq fn_1 > 0 \wedge ln_2 \leq fn_2 < 0).self\triangleright$
$[firstV_1 : (n)!(n = fn1).Group(self, fn_1, ln_1, fn_2, ln_2)\&$
$lastV_1 : (n)!(n = ln1).Group(self, fn_1, ln_1, fn_2, ln_2)\&$
$firstV_2 : (n)!(n = fn2).Group(self, fn_1, ln_1, fn_2, ln_2)\&$
$lastV_2 : (n)!(n = ln2).Group(self, fn_1, ln_1, fn_2, ln_2)\& \ldots]$

$GenerateGroup(k, i, j, r_1, r_2)$ generates group k given remaining durations r_1 and r_2 for $voice_1$ and $voice_2$, respectively. Here i and j represent notes which are being generated in $Voice_1$ and $Voice_2$, respectively. r_1 and r_2 are the remaining allotted durations in the group for each voice. So, i and j will start at $1/2 - r_1$ and $1/2 - r_2$ (w.r.t group), respectively. Initially, r_1 and r_2 are both set to $1/2$.

The first *ask* operation in process *GenerateGroup* says "if both r_1 and r_2 are equal to O then the current $i-1$ and $j+1$ are the last notes for the group in each voice. The second *ask* operation creates a new group k and sets the current i and j as its first notes. The remaining *ask* operations generate notes satisfying the appropriate constraints (to the right of each tell operation, there is a comment specifying the corresponding condition that it implements, e.g. "; cond.3" corresponds to the third condition described above).

Thus, given note i with duration x_2 and starting at $1/2-r_1$ and note j starting at $1/2-r_2$ (w.r.t the group's starting time) then i will sound at the same time as j whenever $r_1 - x_2 \leq r_2$ and $r_1 \geq r_2 > 0$.

$GenerateGroup(k, i, j, r_1, r_2) \stackrel{def}{=}$
$?(r_1 = r_2 = 0)(\ k\triangleleft : lastV_1[i-1] \mid k\triangleleft : lastV_2[j+1]\ )$
$\mid ?(r_1 = r_2 = 1/2).(Vx, y)Group(k, i, x, j, y) \mid$
$\mid ?(r_1 \geq r_2 > 0).(vx_1, x_2, x_3, z_1, z_2, z_3)$
$(\ Note(i, x_1, x_2, x_3)$
$\mid !(x_2 \leq r_1)$; cond. 2
$\mid ?(r_1 = 1/2).!x_3 = 127$; cond. 3
$\mid ?(r_1 < 1/2).!x_3 = 70$; cond. 3
$\mid (i+1)\triangleleft : pitch[z_1]$;next note's pitch
$\mid !(abs(x_1 - z_1) \in MELODY1)$; cond. 1
$\mid j\triangleleft : duration[z_2]$; $Voice_2$ note starting at $1/2 - r_2$
$\mid j\triangleleft : pitch[z_3]$
$\mid ?(r_1 - x_2 \leq r_2 \wedge x_2 > 1/16 \wedge z_2 > 1/8).$
$!(abs(x_1 - z_3) \in HARMONSET1)$;cond. 4
$\mid ?\neg(r_1 - x_2) \leq r_2 \wedge x_2 > 1/16 \wedge z_2 > 1/8).$
$!(abs(x_1 - z_3) \in HARMONSET2)$;cond.4
$\mid GenerateGroup(k, i+1, j, r_1 - x_2, r_2)\ )$
$\mid ?(0 < r_1 < r_2).(vx_1, x_2, x_3, z_1, z_2, z_3)$
$(\ Note(j, x_1, x_2, x_3)$
$\mid !(x_2 \leq r_2)$; cond. 2
$\mid ?(r_2 = 1/2).!x_3 = 127$; cond. 3
$\mid ?(r_2 < 1/2).!x_3 = 70$; cond. 3
$\mid (j-1)\triangleleft : pitch[z]$;next note's pitch
$\mid !(abs(x_1 - z) \in MELODY2)$; cond. 1
$\mid i\triangleleft : duration[z_2]$; $Voice_1$ note starting at $1/2 - r_1$
$\mid i\triangleleft : pitch[z_3]$
$\mid ?(r_2 - x_2) \leq r_1 \wedge x_2 > 1/8 \wedge z_2 > 1/8).$
$!(abs(x_1 - z_3) \in HARMONSET1)$;cond. 4
$\mid ?\neg(r_2 - x_2 \leq r_1 \wedge x_2 > 1/8 \wedge z_2 > 1/8).$
$!(abs(x_1 - z_3) \in HARMONSET2)$;cond. 4
$\mid GenerateGroup(k, i, j-1, r_1, r_2 - x_2)\ )$

$GenerateVoices(k, i, j)$ continuously generates groups until an external condition is met. $GenerateVoices(0, 1, -1)$ is the initial call for starting the whole program.

$GenerateVoices(k, i, j) \stackrel{def}{=}$
$(vz)\ ext - cond\triangleleft : status[z].$
$?z ='' continue''.(\ GenerateGroup(k, i, j, 1/2, 1/2)$
$\mid (vi, j)(\ k\triangleleft : last - v1[i] \mid k\triangleleft : last - v2[j] \mid$
$Generatevoices(k+1, i+1, j-1)\ )$
$\mid GenerateVoices(0, 1, -1)$

7 *Cordial*: A visual language for PiCO

Our research project includes the development of visual programming tools for musical composition. *Cordial* ([LQT97]) 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. A first version of a visual editor for Cordial has been implemented in Java. A translator to (the abstract machine of) PiCO is currently under way. 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 [AA96]. 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. A method definition (see figure 2), is a collection of messages (abstracted away in figure 2 as a cloud), conditionals and formal arguments (the circles in figure 2). Links connecting two elements define relations. The translation into PiCO process of the class definition in figure 1 can be (roughly) defined by a function T as:

$Class1\ (self, attr1, attr2) \stackrel{def}{=}$
$!\ r1 \wedge r2\ [attr1, attr2].self\ \triangleright\ M$

where $M \equiv [met_2 : T(met2)\&met_2 : T(met2)\&met_2 : T(met3)]$.

Figures 3 and 4 show a message and a conditional, respectively. These can be roughly translated by T into processes $a \triangleleft met_2 : [b]$ and $?g_1.T(b_1)|?g_2.T(b_2)|...|?g_n.T(b_n)$, respectively.

8 Conclusions

We defined PiCO, an orthogonal extension of TyCO to handle 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.

We defined the operational semantics 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 illustrated the implementation of a non trivial musical example in the calculus. Finally, we mentioned the relationship between PiCO and a higher level visual language for music composition.

REFERENCES

[AA96] G. Assayag and C. Agon. Openmusic architecture. In *Proceedings of ICMC'96*, Hong Kong, China, 1996. MIT press.

[AC94] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Proceedings of Theoretical Aspects of Computer Software*. Springer Verlag, 1994.

[BR98] A. Bonnet and C. Rueda. *Situation: Un Langage Visuel Basee sur les Contraintes pour la Composition Musicale*. HERMES, Paris, France, 1998. To appear in: Recherches et Applications en Informatique Musicale. M. Chemillier and F. Pachet (Eds).

[CD96] P. Codognet and D. Diaz. Compiling constraints in clp(fd). *J. Logic Programming*, 27:1:1–199, 1996.

[JM95] N. Joachim and M. Müller. Constraints for Free in Concurrent Computation. In Kanchana Kanchanasut and Jean-Jacques Lévy, editors, *Asian Computing Science Conference*, Lecture Notes in Computer Science, vol. 1023, pages 171–186, Pathumthani, Thailand, December 1995. Springer-Verlag.

[LQT97] C. Rueda L.O. Quesada and G. Tamura. The visual model of cordial. Research Report AV-97-03, Grupo Avispa Universidad Javeriana-Cali, Via a Pance, Cali, Colombia, 1997. available electronically:http://sofia.univalle.edu.co/pub/Avispa/.

[Mil80] Robin Milner. A calculus of communicating systems. Lecture Notes in Computer Science, LNCS 92, 1980.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science,, October 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag , 1993.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MNS97] M. Müller, J. Niehren, and G. Smolka. Typed Concurrent Programming with Logic Variables. Technical report, Programming Systems Lab, Universität des Saarlandes, 1997.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proc. of 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[Nie96] Joachim Niehren. Functional computation as concurrent computation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM Press, January 1996.

[Pie96] B. C. Pierce. Programming in the pi-calculus: An experiment in programming language design. Tutorial notes on the PICT language. Available electronically, 1996.

[PR95] F. Pachet and P. Roy. Mixing constraints and objects: A case study in automatic harmonization. In *Proceedings of TOOLS Europe'95*, pages 119–126, Versailles, France, 1995. Prentice-Hall.

[PRT93] B. C. Pierce, D. Rémy, and D. N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.

[PT94a] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan*, November 1994.

[PT94b] B. C. Pierce and D. N. Turner. PICT user manual. Technical report. Available electronically, 1994.

[PT96] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical report; available electronically, 1996.

[RMW92] J. Parrow R. Milner and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.

[Smo94a] G. Smolka. 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, February 1994.

[Smo94b] G. Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, September 1994. Springer-Verlag.

[Tur95] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Laboratory for Foundations of Computer Science,, 1995.

[Vas94] V. T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *Proc. of 8th European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, 1994.

[VDR97] F. Valencia, J. F. Diaz, and C. Rueda. The π^+-calculus. Research Report AV-97-02, Grupo Avispa Universidad Javeriana, Universidad del Valle -Cali, 1997. available electronically: http://sofia.univalle.edu.co/pub/Avispa/Pi+/.

[Wal91] D. Walker. π-calculus semantics of object-oriented programming languages. In Takayasu Ito and Al-

bert Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991. Available as Report ECS-LFCS-90-122, University of Edinburgh.

[Wal95] D. Walker. Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.

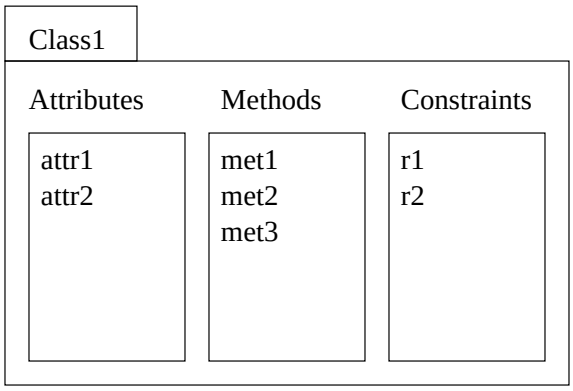

Figure 1. Class definition in Cordial

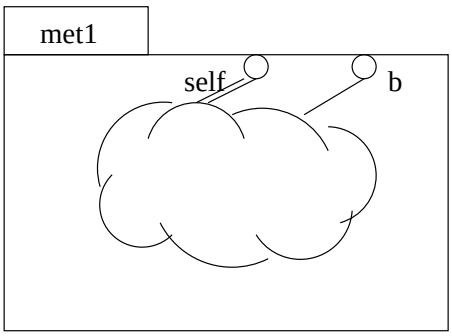

Figure 2. Method definition in Cordial

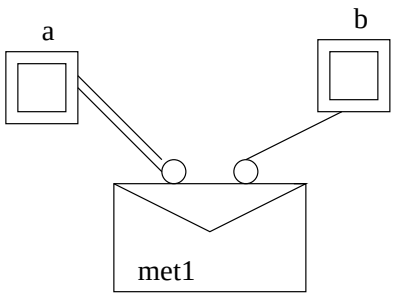

Figure 3. Message definition in Cordial

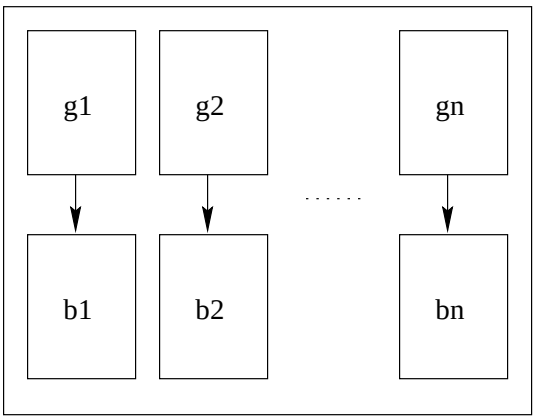

Figure 4. Conditional definition in Cordial