MAPiCO: An Abstract Machine for the *PiCO* Calculus

Antal A. Buss, Mauricio Heredia, Gabriel Tamura
{abuss, mheredia, gtamura}@atlas.ujavcali.edu.co
Pontificia Universidad Javeriana - Cali

Abstract

This paper presents a description of the design and implementation of an abstract machine for the *PiCO* calculus. The aim of this calculus is to integrate the object-oriented, concurrent and constraint programming paradigms, to provide a sound basis for high-level programming languages presenting characteristics of these paradigms.

A set of simple transition rules is used to specify the machine operation. These rules are adapted from the transition rules of *PiCO* and require some supporting structures.

A general protocol is built into the abstract machine, that allows interaction with implementations of different constraint systems.

MAPiCO is implemented in Java and provides a target machine for the compilation of programs written in *Cordial* [QRT97], a visual language integrating concurrent objects and constraints.

Keywords: *PiCO*, Abstract Machines, Concurrent Programming, Constraint Programming, Concurrent Constraint objects, programming languages

1 Introduction

PiCO [ADQ+98] is a calculus that integrates the object oriented, concurrent, and constraint programming paradigms. It is based on the π^+-calculus [VDR97] which, in turn, is based on Milner's polyadic π-calculus [MPW92]. The basic processes of the π-calculus are extended with objects and constraints, the first including the notion of message-delegation and the latter adding (orthogonally) the notion of *constraint system* [Sar93]. These extensions provide a natural way to express more sophisticated communication schemes than are possible with the standard message-passing synchronization mechanism.

Several concurrent object calculi have been proposed recently, such as TyCO [Vas94], ϕ [Mil80] and Cardelli's imp-ς [AC96]. In these models, the interactions of concurrent processes (or objects) are synchronized through one of two mechanisms: the "use of channels" and "message-passing". However, the notion of constraint is not considered.

Constraints can be used to define a rich set of possible concurrent process interactions. Complex synchronization schemes can be defined with the basic operations *ask* and *tell*, through the sharing of global variables in processes. In the constraint language *Oz* [HM94, Smo94], first class procedures and first class cells are used to simulate objects within a concurrent constraint context. The powerful constraint calculus ($\rho-$calculus) of *Oz* allows other programming models (e.g. functional, objects) to coexist through syntactic encodings. However, although the fundamental concepts of the object-oriented paradigm can be simulated through (rather tricky) encodings in *Oz*, they are not primitive notions. This means that inefficiencies are likely to occur in practice when real object-oriented programs are written in this language.

The purpose of *PiCO*, on the contrary, is to include objects and constraints directly at the calculus level.

MAPiCO is an abstract machine that implements *PiCO*. It was designed to provide the fundamental requirements to "execute" *PiCO* processes by reducing them according to the transition rules and congruence relations defined in this calculus.

Section 2 of this article describes the syntax and operational semantics of *PiCO*. The design of the abstract machine is presented in section 3. Section 4 describes a simple example of constraint system that has been integrated to the machine. Section 5 discusses some implementation details and, finally, section 6 presents some conclusions and present lines for future work.

2 The *PiCO* Calculus

2.1 *PiCO* Syntax

As was mentioned above, *PiCO* is based on the $\pi-$calculus. The key notion in both is that of *process.* There exists three basic process types in *PiCO*: *messages, Objects,* and *constraints* (see table 1 [1]).

Normal Processes: N	::=	O	Inaction or null process
	|	$I \lhd m.P$	Message to I
	|	$(I, J) \rhd M$	Object I with delegation to J
Constraint Processes: R	::=	$!\phi.P$	Process *tell*
	|	$?\phi.P$	Process *ask*
Processes: P, Q	::=	$(\upsilon x)P$	New variable x in P
	|	$(\upsilon a)P$	New name a in P
	|	N	Normal Process
	|	$P \mid Q$	Concurrent composition
	|	$*P$	Replicated process
	|	R	Constraint Process
Object identifiers: I, J	::=	a	Name
	|	v	Value
	|	x	Variable
Collection of Methods M	::=	$[l_1 : (\widetilde{x_1})P_1 \& \ldots \& l_m : (\widetilde{x_m})P_m]$	
Message m	::=	$l : [\widetilde{I}]$	

Table 1: The *PiCO* Syntax

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 $(I, J) \rhd M$, where M is a collection of methods $[l_1 : (\widetilde{x_1})P_1 \& \ldots \& l_m : (\widetilde{x_m})P_m]$, can be thought of as an object *located at, identified by, or named by* I, whose methods $(\widetilde{x_1})P_1, \ldots, (\widetilde{x_m})P_m$ are labeled by a set of pairwise distinct labels $Labels(M) = \{l_1 \ldots l_m\}$. Activation of methods is performed by sending messages to objects as in $I \lhd l : [\widetilde{X}].P$. In this process, I is the destination object of a message named l with arguments $\widetilde{X}$. P is the continuation to be performed after reception of the message is acknowledged. Identifier J, the *delegation address*, represents the address of the process where messages should be delegated to when there are no appropriated methods in the original receiver to handle the message. Objects of the form $(I, I) \rhd M$ represent objects without delegation. For the sake of simplicity, they will be abbreviated as $I \rhd M$. In general, only objects of the form $(I, J) \rhd M$, such that it can be inferred that $I \neq J$, allow message delegation.

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

Process $(\upsilon a)P$ declares a new (unique) name a, distinct from all external names, to be used in P. Similarly, $(\upsilon x)P$ declares a new variable x.

$*P$ (*replication*) means $P \mid P \ \ldots$ (as many copies as needed). A common instance of replication is $*(I, J) \rhd M$, an object which reproduces itself (i.e. it persists) when a requester communicates with 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. In particular, the store is used to control all potential communications. The meaning of the *Tell* and *Ask* processes is based on the notion of *constraint system* (see [Sar93]).

[1] In reality, here it is considered a subset of the *PiCO* calculus. A complete description is presented in [RAQ+98]

2.2 Operational Semantics

The behavior of a process P is defined by transitions between configurations of the form $\langle P; \top \rangle$ (that is, the process P in the context given by the store $\top$). A transition, $\langle P; S \rangle \longrightarrow \langle P'; S' \rangle$, means that the configuration $\langle P; S \rangle$ can be transformed into the configuration $\langle P'; S' \rangle$ in a single computational step, by using one of the reduction rules of the transition system. The reduction rules are presented next. [ADQ+98] presents a detailed description.

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

$$\text{COMM:}\quad \frac{S \models_\Delta I = I' \qquad |\widetilde{K}| = |\widetilde{x}|}{\left\langle I' \lhd l{:}[\widetilde{K}].Q \mid (I,J) \rhd [l{:}(\widetilde{x})P \& \ldots]; S \right\rangle \longrightarrow \left\langle Q \mid P\left\{\widetilde{K}/\widetilde{x}\right\}; S \right\rangle}$$

$$\text{DELEG:}\quad \frac{S \models_\Delta I' = I \wedge I' \neq J \qquad l \notin Labels(M)}{\left\langle I' \lhd l{:}[\widetilde{K}].Q \mid (I,J) \rhd M; S \right\rangle \longrightarrow \left\langle J \rhd l{:}[\widetilde{K}].Q \mid (I,J) \rhd M; S \right\rangle}$$

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

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

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

$$\text{DEC-V:}\quad \frac{x \notin fv(S),\ \langle P; S \gg \{x\} \rangle \longrightarrow \langle P'; S' \rangle}{\langle (vx)P; S \rangle \longrightarrow \langle P'; S' \rangle}$$

$$\text{DEC-N:}\quad \frac{a \notin fn(S),\ \langle P; S \gg \{a\} \rangle \longrightarrow \langle P'; S' \rangle}{\langle (va)P; S \rangle \longrightarrow \langle P'; S' \rangle}$$

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

Table 2: The *PiCO* Transition System

COMM specifies the reduction of a method invocation $I' \lhd l : [\widetilde{J}].Q$ for the object $(I, J) \rhd [l : (\widetilde{x})P \& \ldots]$. The store is used to decide whether the object is indeed the target of the message. Notice that message continuation Q is activated when the message is received.

DELEG describes message delegation. Let $I \lhd l : [\widetilde{K}].Q$ be a message sent to object $(I, J) \rhd M$ and suppose that no method with label l exists in M. In this case, the message is forwarded to the delegation address J. To avoid non-terminating executions, it is checked that the "new" address of the message is different from the previous one.

The ASK and TELL rules specify the reduction of constraint processes. The TELL process $!\phi.P$ adds the constraint ϕ to the store S and then activates its continuation P. The ASK process $?\phi.P$ queries the store for the constraint ϕ. The rule says that P can be activated if the current store S entails ϕ, or discarded when S entails $\neg\phi$. An ask process that cannot be reduced in a store S is said to be *suspended* by S. ASK processes add the "blocking ask" mechanism of 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 the 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 a new variable x to store S. DEC-N is defined in a similar way. Rule EQUIV simply says that P-equivalent configurations have the same reductions.

3 Design of the *MAPiCO* Abstract Machine

As was already said, the abstract machine must provide the data and control structures required to support the execution of the basic computational entities of the calculus, which are the processes. Specifically, a program is considered to be a set of concurrent processes.

For any process, the machine keeps both a static and a dynamic representation. The first is addressed through a basic instruction set and its corresponding binary encoding format. The second, through a set of machine registers that represents the state of execution of the process.

A process is executed by reducing it as specified by (an adaptation of) the rules of the *PiCO* transition system. In other words, the operational semantics of a process in the machine is specified by the rules of the calculus, adapted for the machine. This behaviour is bound to certain instructions of the machine instruction set. This so-called execution requires some supporting structures, namely structures for representing the storage space (for elements like variables, object definitions, argument passing and the constraints store) and control management (basically some queues to hold the representation of the processes in different states of execution).

The calculus is parameterized with a constraint system whose constraints are just (subsets of) first-order logic formulæ. The machine provides a "protocol" for interfacing with constraint systems based on this kind of constraints. The protocol consists in a specification about how the compiler should build and represent a formula, together with the definition of the signature of the *ask* and *tell* operations.

Finally, for efficiency reasons, only *normal* processes can be replicated in the machine. This should not limit significantly the expressive power of the machine in practice. In fact, the aproximated behavior of any replicated process can be achieved through (1).

$$*P \approx (va)(*(a,a) \rhd [rep : ()P | a \lhd rep[]] \mid a \lhd rep[]) \tag{1}$$

3.1 Process Representation

3.1.1 Dynamic Representation

The concurrent execution of processes in the machine is achieved by considering each one in turn for reduction.

To support the execution of one process, the machine has a set of some specific-purpose registers. This set fully describes the state of execution of a process. The process being held at a given time by the set of registers is called the actual process. Thus, the dynamic representation of a process is given by a tuple of the contents of the machine registers, that is, <PC,PV,PN,PA>. These registers are:

- PCA (code pointer): Points to the current instruction (in the program memory) of the actual process to be executed.
- PVA (variable pointer): Points to the first element (in the translation memory) of a variable-binding list. This list contains the address, in the constraint store, where a representation of each process variable is held.
- PNA (name pointer): Points to the first element (in the translation memory) of a name-binding list. This list contains the address in the constraint store where a representation of each process name is held.
- PAA (argument pointer): Similar to PVA and PNA, PAA points to the translation memory. This pointer has two uses. Firstly, to point to a list with the arguments and object receiver of a message when a communication is taking place. Secondly, to reference nodes in the construction of a syntax tree representing a constraint.
- PAUX (auxiliary pointer), used to manipulate and execute processes.

3.1.2 Static Representation

A process is represented statically by a sequence of machine instructions. This sequence of instructions is stored in the program memory, along with the sequences of all other concurrent processes in a program. Processes are identified by program-memory addresses.

Machine Instructions The format of the machine instructions consists of one operation code and from 0 to 3 operands. The instructions are classified in 3 groups: process definition/manipulation (table 3), object definition (table 4) and constraint definition (table 5).

In these tables, **addr** denotes a valid program address; **ind**, an index in a list (pointed to by PV or PN); **num** denotes a number (for objects this specifies the number of methods); and **nam**, a reference to a name. Before executing a method call, the object receiver must be pushed first (using *pushn* or *pushv*).

Mnemonic	Opcode	Src1	Description
ret	0		Nil process
par	1	addr	Concurrent composition
newv	2		New free variable
newn	3		New free name
pushv	4	ind	Push variables in the PA
pushn	5	ind	Push names in the PA
pop	18		Pop the first element from the stack PA (to be pushed on stack PV or PN)
call	20	nam	Call method nam in the top object of PA
tell	24		Tell the constraint pointed to by PA
ask	25		ask for the constraint pointed to by PA

Table 3: Instructions for Process Definition and Manipulation

Mnemonic	Opcode	Src1	Src2	Src3	Description
objvv	64	ind	ind	num	Object definition (src1,src2) with src1 and src2 being variables
objnn	65	ind	ind	num	Like objvv but src1 and src2 being names
objvn	66	ind	ind	num	Like objvv but src1 variable and src2 name
objnv	67	ind	ind	num	Like objvv but src1 name and src2 variable
objvvr	72	ind	ind	num	Like objvv but replicated
objnnr	73	ind	ind	num	Like objnn but replicated
objvnr	74	ind	ind	num	Like objvn but replicated
objnvr	75	ind	ind	num	Like objnv but replicated
meth	80	nam	addr		Method definition (with name,address)

Table 4: Instructions for Object Definition

It must be noted that the instructions for constraint definitions allow the representation of any first-order predicate so as to provide a mechanism that makes it possible the parameterization of the machine with different constraint systems.

3.2 Process Execution

3.2.1 Support Structures

Storage Support There are three main support structures for storage: the program memory, the constraints store, and the translation memory.

The program memory is a linear addressable (usually through register PCA) area of memory where the program instructions (static representation of processes) are stored.

The constraints store contains every constraint asserted by *tell* processes. Its behavior depends on some specific constraint system and therefore it cannot be described here. However, the constraint system is assumed to obey the established protocol mentioned at the beginning of section 3.

The translation memory holds variable and name bindings, the actual arguments in method invocations, and the tree representation of constraints.

The structure for the variable and name bindings is handled through machine registers PVA and PNA, respectively. It has two components: (1) *name*, used to hold a reference to the bound variable (or name), and (2) a

Mnemonic	Opcode	Src1	Src2	Description
termc	32	num		Constant term
termv	33	ind		Variable term
termn	34	ind		Name term
termf	35	fun	num	Function with arity
atom	36	pred	num	Atom with arity
sentenc	40	conn		Sentence with a connective (or, and)
sentenq	41	quant		Sentence with a quantifier or negation

Table 5: Instructions for Constraint Definition

reference to the next binding structure of that kind (that is, variable or name structure).

The structure for actual arguments has three components: (1) *name*, a reference to the argument, (2) *kind*, which indicates whether the argument is a variable or name, and (3) a reference to the next actual argument. This structure is handled by using machine register PAA.

Finally, the structure for a node in the tree representation of constraints has five components, as illustrated in figure 1: (1) *parent*, a pointer to the parent of the node, (2) its value, (3) the node class (function, atom, sentence), (4) number of children, given by the arity of the node class, and (5) the list of children. This structure is also handled through the machine register PAA.

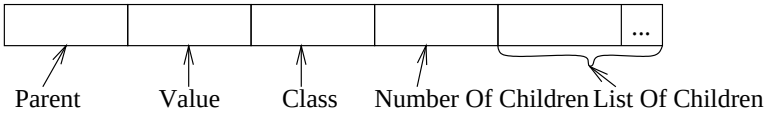

Figure 1: Constraint Frames in *MAPiCO*

On execution, the combined structures for variable and name bindings of multiple concurrent processes may turn into an inverted tree, each (dynamic representation of a) process having, through its PV (PN) tuple component, its own "point of access" to its variable (name) binding structure. The path from this "point of access" up to the root of the tree represents all the variable (name) bindings of the process. This inverted tree grows down a branch each time a new variable (name) is created in a some process whose PV (PN) points to that branch. Branches expand when a parallel composition process is reduced, since one of the processes involved in the reduction replaces the original (parent) process and the other must be made its brother in the tree by creating a new process node for it. The process replacing its father "inherits" its branches. The PV and PN pointers of the just created brother are also made to point to the same nodes as the replaced father used to point to. Figure 2 illustrates this.

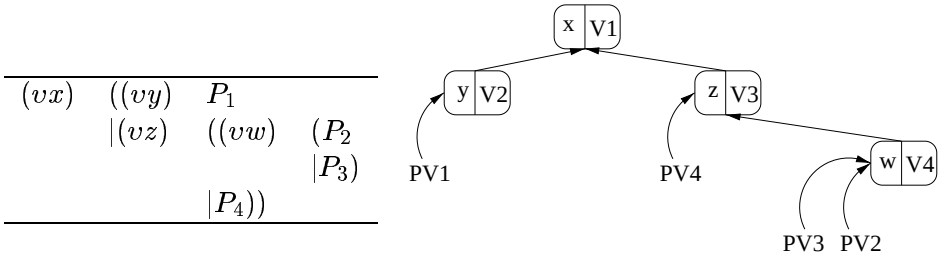

Figure 2: Variable Binding Structure in a Program

Support for Concurrent Execution Concurrent execution is implemented through the definition of four process queues. Each of these queues holds the dynamic representation of processes in different states of execution, as classified by the type of the process:

- Rq (Run queue): holds all kind of processes in the state "ready for execution".
- Oq (Suspended Objects queue): holds replicated objects and objects for which communication is pending.
- Msq (Suspended Messages queue): holds those messages whose target object does not exist or whose condition of communication cannot be established by the store (see rules ObjComm, ObjDel and ObjWODel in section 3.2.2).
- Aq (Suspended *Asks* queue): holds those *ask* processes for which the corresponding asked formula cannot be entailed by the store.

The execution of processes and how they change of state is ruled by the machine reduction rules, which are described in the next section. Figure 3 also helps to illustrate this.

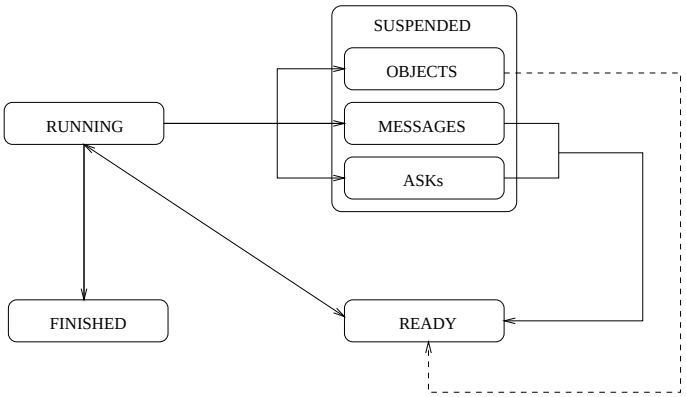

Figure 3: State Transition Diagram for Processes in Execution

3.2.2 Machine Reduction Rules

The machine reduction rules are "low level" transcriptions of the transition system rules of *PiCO*. Therefore, these rules specify the execution of the processes in the machine, also in terms of configuration transformations.

In the rules described in this section, an empty queue is represented by $\bullet$; the operator :: denotes queue concatenation; $\oslash$ means that there are no variable bindings, and T is an empty store.

A configuration of the machine is represented by the tuple:

$$\langle Thread, HBind, HAux, ObjQ, MsgQ, AskQ, RunQ, Store \rangle$$

Thread, *HBind* and *HAux* correspond to the PC, (PV, PN) and PA components of the dynamic representation of the actual process.

The transformation of configurations is expressed in the following way:

$$\left\langle \vec{P}, B, H, Oq, Msq, Aq, Rq, S \right\rangle \longrightarrow \left\langle \vec{P}', B', H', Oq', Msq', Aq', Rq', S' \right\rangle$$

Each rule examines the actual process to see if its conditions apply. If they do, the rule transforms the current configuration into its ending configuration, thus reducing the process.

The $SCHED$ rule has no condition and specifies the reduction of the null process, so it is simply removed from the run queue, enabling the next process in this queue to be considered:

$$\text{SCHED:} \langle nil, B, H, Oq, Msq, Aq, (B', H', \vec{P}) :: Rq, S \rangle \longrightarrow \langle \vec{P}, B', H', Oq, Msq, Aq, Rq, S \rangle$$

The process $(\nu x)\vec{P}$ or $(\nu a)\vec{P}$ creates a new binding from x to L where L is a new variable (generated by the machine):

$$\text{NEW-REF}: \frac{L \ New}{\langle (\nu x)\vec{P},B,H,Oq,Msq,Aq,Rq,S\rangle \longrightarrow \langle \vec{P},B+(x\mapsto L),H,Oq,Msq,Aq,Rq,S\rangle}$$

Here $B+(x\mapsto L)$ means a new reference to the constraint store in the translation memory.

If the process is a parallel composition $(P|Q)$, the machine splits the composition, placing Q at the end of the run queue:

$$\text{PARALLEL}: \frac{}{\langle (P|Q),B,H,Oq,Msq,Aq,Rq,S\rangle \longrightarrow \langle P,B,H,Oq,Msq,Aq,Rq::(B,H,Q),S\rangle}$$

To reduce a *tell* process $!\phi.P$, the constraint ϕ is sent to the store and all suspended messages or suspended *ask* processes are moved to the run queue.

$$\text{TELL}: \frac{\forall L_i \in \widetilde{L} \ \ L_i \in Dom(B)}{\left\langle !\phi.\vec{P},B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle \vec{P},B,\emptyset,Oq,\bullet,\bullet,Rq::Msq::Aq,S\wedge\phi(\widetilde{L})\right\rangle}$$

To reduce an *ask* process $(?\phi.\vec{P})$, the constraint store must be queried. There are 3 possible cases:

1. If the *store* entails $\phi(\widetilde{L})$, the machine continues with the execution of the thread $\vec{P}$:

$$\text{ASK1}: \frac{S\models_\Delta \phi(\widetilde{L}) \quad \forall L_i \in \widetilde{L} \ \ L_i \in Dom(B)}{\left\langle ?\phi.\vec{P},B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle \vec{P},B,\emptyset,Oq,Msq,Aq,Rq,S\right\rangle}$$

2. If the *store* entails $\neg\phi(\widetilde{L})$ the machine removes the *ask* process, enabling the next process in the run queue to be reduced:

$$\text{ASK2}: \frac{S\models_\Delta \neg\phi(\widetilde{L}) \quad \forall L_i \in \widetilde{L} \ \ L_i \in Dom(B)}{\left\langle ?\phi.\vec{P},B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle nil,B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\right\rangle}$$

3. If from the store neither $\phi(\widetilde{L})$ nor $\neg\phi(\widetilde{L})$ can be entailed, the *ask* process is suspended by placing it in the suspended Asks queue.

$$\text{ASK3}: \frac{S\not\models_\Delta \phi(\widetilde{L}) \quad S\not\models_\Delta \neg\phi(\widetilde{L}) \quad \forall L_i \in \widetilde{L} \ \ L_i \in Dom(B)}{\left\langle ?\phi.\vec{P},B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle nil,B,\phi(\widetilde{L}),Oq,Msq,Aq::(B,\phi(\widetilde{L}),?\phi.\vec{P}),Rq,S\right\rangle}$$

To reduce a message process referencing an object which is not yet in the suspended objects queue, the message process is suspended in the suspended messages queue:

$$\text{MsgEnQ}: \frac{S\models_\Delta I=I' \quad (I',J')\rhd M\notin Oq}{\left\langle I\lhd l_i:[\widetilde{K}].\vec{P},B,\widetilde{x_i}\mapsto\widetilde{L},Oq,Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle nil,B,\widetilde{x_i}\mapsto\widetilde{L},Oq,Msq::(B,\widetilde{x_i}\mapsto\widetilde{L},I\lhd l_i:[\widetilde{K}].\vec{P}),Aq,Rq,S\right\rangle}$$

On the contrary, when the process is a message $I\lhd l_i:[\widetilde{K}].\vec{P}$ to an object that is in the suspended objects queue and the message label l_i belongs to the set of object methods, the object is removed from this queue and the method body of $\vec{P}_i$, with a new binding to $\widetilde{K}$, is placed at the end of the run queue:

$$\text{MsgComm}: \frac{S\models_\Delta I=I' \quad o\le i\le m \quad |\widetilde{K}|=|\widetilde{x_i}| \quad (\widetilde{K}\mapsto\widetilde{L})\in B}{\left\langle I\lhd l_i:[\widetilde{K}].\vec{P},B,\widetilde{x}\mapsto\widetilde{L},Oq::(B',\emptyset,(I',J')\rhd[l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle \vec{P},B,\emptyset,Oq,Msq,Aq,Rq::(B'+(\widetilde{x_i}\mapsto\widetilde{L}),\emptyset,\vec{P_i}),S\right\rangle}$$

If the target object of the message $(I',J')\rhd[l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]$ (with $S\models_\Delta I'\neq J'$) exists, that is, $(S\models_\Delta I=I')$, but the invoked method l_i is not defined in it, the message is then forwarded to J', its delegation address, and the the original target object is kept in the suspended objects queue:

$$\text{MsgDel}: \frac{S\models_\Delta I=I' \quad S\models_\Delta I'\neq J' \quad i\ge m \quad (\widetilde{K}\mapsto\widetilde{L})\in B}{\left\langle I\lhd l_i:[\widetilde{K}].\vec{P},B,\widetilde{x_i}\mapsto\widetilde{L},Oq::(B',\emptyset,(I',J')\rhd[l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle nil,B,\widetilde{x_i}\mapsto\widetilde{L},Oq::(B',\emptyset,(I',J')\rhd[l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq,Aq,Rq::(B,(\widetilde{x_i}\mapsto\widetilde{L}),J'\lhd l_i:[\widetilde{K}].\vec{P}),S\right\rangle}$$

Similarly, if the target object $(I', J') \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ (with $S \models_\Delta I' = J'$, that is, it has no delegation address) exists but the invoked method l_i is not defined in it, the message is then suspended in the suspended messages queue:

$$\text{MsgWODel: } \frac{S \models_\Delta I = I' \quad S \models_\Delta I' = J' \quad i \geq m \quad (\widetilde{K} \mapsto \widetilde{L}) \in B}{\left\langle I \lhd l_i : [\widetilde{K}].\vec{P}, B, \widetilde{x_i} \mapsto \widetilde{L}, Oq :: (B', \emptyset, (I', J') \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]), Msq, Aq, Rq, S \right\rangle \longrightarrow \left\langle nil, B, \widetilde{x_i} \mapsto \widetilde{L}, Oq :: (B', \emptyset, (I', J') \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]), Msq :: (B, \widetilde{x_i} \mapsto \widetilde{L}, I \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle}$$

In case there are no messages to communicate with $(I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ in the suspended messages queue, then the object is placed back in the suspended objects queue for later execution:

$$\text{ObjEnQ} : \frac{S \models_\Delta I = I' \quad I' \lhd m.\vec{P} \notin Msq}{\left\langle (I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}], B, \emptyset, Oq, Msq, Aq, Rq, S \right\rangle \longrightarrow \left\langle nil, B, \emptyset, Oq :: (B, \emptyset, (I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]), Msq, Aq, Rq, S \right\rangle}$$

In case there is some message $I' \lhd l_i : [\widetilde{K}].\vec{P}$ in the suspended messages queue, and furthermore, the receiver object $(I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ exists and the invoked method l_i is defined in it, then the communication can take place: the message is removed from the suspended messages queue, its continuation $\vec{P}$ is enqueued in the run queue, followed by the method body $\vec{P}_i$, with new bindings to $\widetilde{K}$:

$$\text{ObjComm} : \frac{S \models_\Delta I = I' \quad 0 \leq i \leq m \quad |\widetilde{K}| = |\widetilde{x_i}| \quad (\widetilde{K} \mapsto \widetilde{L}) \in B'}{\left\langle (I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}], B, \emptyset, Oq, Msq :: (B', (\widetilde{x_i} \mapsto \widetilde{L}), I' \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle \longrightarrow \left\langle nil, B, \emptyset, Oq, Msq, Aq, Rq :: (B', \emptyset, \vec{P}) :: (B + (\widetilde{x_i} \mapsto \widetilde{L}), \emptyset, \vec{P}_i), S \right\rangle}$$

The following rule specifies the situation where a message $I' \lhd l_i : [\widetilde{K}].\vec{P}$ in the suspended messages queue is trying to communicate with an object $(I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ with delegation $(S \models_\Delta I \neq J)$, but l_i is not defined in the object. The message is moved from the suspended messages queue to the end of the run queue, after changing its object receiver by the one indicated in the delegation address (i.e. $J \lhd l_i : [\widetilde{K}].\vec{P}$). The original object is inserted back in the suspended objects queue:

$$\text{ObjDel: } \frac{S \models_\Delta I = I' \quad S \models_\Delta I \neq J \quad i \geq m \quad (\widetilde{K} \mapsto \widetilde{L}) \in B'}{\left\langle (I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}], B, \emptyset, Oq, Msq :: (B', (\widetilde{x_i} \mapsto \widetilde{L}), I' \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle \longrightarrow \left\langle nil, B, \emptyset, Oq :: (B, \emptyset, (I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]), Msq, Aq, Rq :: (B', \widetilde{x_i} \mapsto \widetilde{L}, J \lhd l_i : [\widetilde{K}].\vec{P}), S \right\rangle}$$

Finally, if there is a message $I' \lhd l_i : [\widetilde{K}].\vec{P}$ to an object $(I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ without delegation (i.e. $(S \models_\Delta I = J)$) in the suspended messages queue , and l_i is not defined in the target object, the object is placed in the suspended objects queue:

$$\text{ObjWODel: } \frac{S \models_\Delta I = I' \quad S \models_\Delta I = J \quad i \geq m \quad (\widetilde{K} \mapsto \widetilde{L}) \in B'}{\left\langle (I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}], B, \emptyset, Oq, Msq :: (B', (\widetilde{x_i} \mapsto \widetilde{L}), I' \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle \longrightarrow \left\langle nil, B, \emptyset, Oq :: (B, \emptyset, (I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]), Msq :: (B', (\widetilde{x_i} \mapsto \widetilde{L}), I' \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle}$$

Replicated processes always remain in the suspended objects queue or suspended messages queue, correspondingly.

3.2.3 Initial and Final Configurations

The machine starts in a configuration with its process queues empty, no bindings, no constraints, and as actual process the "bootstrap process loader". This process has only a dynamic representation, which is $< 0, nil, nil, nil >$. It is called the "bootstrap process loader" because its PC points to 0, the address in program memory of the first of all concurrent processes composing the program in execution. The static representation of these processes are chained in program memory through the instruction encoding the concurrency operator. The execution of this process will "load and execute" the rest of the processes. The initial configuration is then:

$$\langle < 0, nil, nil, nil >, \emptyset, \emptyset, \bullet, \bullet, \bullet, \bullet, T \rangle$$

The final configuration is reached when the actual process is *nil* and the Run Queue is empty:

<operation>	::=	ask <constraint> |
		tell <constraint>
<constraint>	::=	<variable> in <range><continuation> |
		<variable> = <variable>
<variable>	::=	iden
<range>	::=	<value><continuation_range>
<value>	::=	number |
		Min(<variable>) |
		Max(<variable>)
<continuation_range>	::=	: <value><continuation_range> |
		<empty>
<continuation>	::=	+ {number}|
		<empty>

Table 6: BNF for the Constraint System Formulæ

$$\langle nil, B, H, Oq, Msq, Aq, \bullet, S \rangle$$

Since the run queue is empty, any remaining processes in the suspended objects, messages, or asks queues are left suspended (they can not be further reduced).

4 A Possible (Simple) Constraint System

This section briefly describes a simple finite-domain constraint system that has been integrated into *MAPiCO*. This constraint system is based on [CD96] and is still under development.

Currently, the system allows just two kinds of constraints:

- Domain constraints, such as $x \ in \ R$.
- Equality constrains, such as $x = y$.

The BNF of the formulae handled by this constraint system is shown in table 6.

The store is implemented as a set of domain constraints and a graph representing equality constraints. The state of the store may be consistent or inconsistent. The state is inconsistent when it has been given contradictory information. An inconsistent store can entail anything and can thus be considered a sort of run time failure. In this case, every *ask* operation returns true and every *tell* operation fails. The answers of an inconsistent store thus gives no useful information.

The store also contains a list of suspended *tell*s. The reason is that the algorithms for efficient testing of consistency are usually incomplete. For example, in the constraint system under consideration, a constraint $X \ in \ Min(Y)$ told to a store in which Y is not present, is accepted, but left suspended. This list is iterated each time a *tell* is actually performed to see if any suspended *tell* constraint can now be asserted in the new store.

5 Implementation of the Abstract Machine

The implementation strategy for *MAPiCO* differs from those used in [Lop98] and [PT96]. The TyCO Abstract Machine uses a linear memory to allocate the heap and temporary data. It uses word-code (w-code) to code basic instructions. PICT, on the other hand, translates the source code into the C language and compiles the result using a standard C compiler.

MAPiCO is implemented in Java. Java classes are used to handle memory areas, processes scheduling and all data structures needed for execution, as was explained in section 3. A first version of a *PiCO* compiler,

translating into *MAPiCO*, has also been implemented in Java. *PiCO* is in turn the underly model of *Cordial*, a visual language integrating object-oriented and constraint programming, designed for musical composition applications. At present, *Cordial* compiles into *PiCO* which itself compiles into the abstract machine presented here. A first version of *Cordial* written also in Java can be obtained electronically by public ftp from: ftp://atlas.ujavcali.edu.co/pub/grupos/avispa/cordial/

6 Conclusions and Future Work

Formal calculi are used to specify formally the semantics of programming languages. Based on *PiCO*, a calculus that integrates the concurrent, constraint and object oriented paradigms, an abstract machine has been designed and implemented, thus allowing to calculate automatically the "meaning" of a program, if translated to this abstract machine.

The abstract machine is defined by reduction rules that "specialize" those of *PiCO*. In this sense the abstract machine could also be regarded as a "low level" calculus. An interesting issue that is intended to pursue in the near future is to show the formal equivalence of these two calculus.

The transition rules of *PiCO* were simplified to allow replication of normal processes only. This simplification is not likely to affect the expressive power of the calculus in practice. In fact, an acceptable encoding of the replication of other processes was given. The simplification, on the contrary, can have a strong effect over the efficiency of the implementation (e.g. produces a lot less less unnecessary processes).

Projects under development include extensions of the machine for general input/output and schemes for flexible exploration of solutions consistent with a constraint store, and integration of different constraint systems.

PiCO is being used to define the semantics of a high-level programming language called *Cordial*, which is implemented in Java. *MAPiCO* and its low-level compiler from *PiCO* have also been implemented in Java.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[ADQ+98] Gloria Alvarez, Juan Francisco Diaz, Luis O. Quesada, Camilo Rueda, and Frank D. Valencia. Pico: A calculus of concurrent constraint object for musical applications. ECAI98, Workshop on Constraint & the Arts, Brighton, England, 1998.

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *The Journal of Logic Programming*, 1996.

[HM94] M. Henz and M. Muller. Programming in oz. *DFKI Documentation series*, 1994.

[Lop98] Luís Lopes. The tyco abstract machine implementation. Technical report, Universidade do Porto, January 1998.

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

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

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.

[PT96] Benjamin C. Pierce and David Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation; available electronically, 1996.

[QRT97] Luis Omar Quesada, Camilo Rueda, and Gabriel Tamura. The visual model of cordial, 1997. XXIII Conferencia Latinoamericana de Informática.

[RAQ+98] Camilo Rueda, Gloria Alvarez, Luis Omar Quesada, Gabriel Tamura, Frank Valencia, Juan Francisco D'iaz, and Gerard Assayag. Integrating Constraints and Concurent Objects in Musical Applications: A Calculus and its Visual Language. Submitted to *Constraints*, Kluwer Academic Publishers, 1998.

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

[Smo94] Gert 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.

[Vas94] Vasco T. Vasconcelos. *Typed Concurrent Objects.* PhD thesis, Keio University, November 1994.

[VDR97] Frank Valencia, Juan Francisco Diaz, and Camilo Rueda. The π^{+}-calculus, 1997. XXIII Conferencia Latinoamericana de Informática.