An Intensional Programming Approach to Multi-agent Coordination in a Distributed Network of Agents*

Kaiyu Wan and Vasu S. Alagar

Department of Computer Science,
Concordia University,
Montreal, Quebec H3G 1M8, Canada
{ky_wan, alagar}@cse.concordia.ca

Abstract. We explore the suitability of *Intensional Programming Paradigm* for providing a programming model for coordinated problem solving in a multi-agent system. We extend our previous work on Lucx, an Intensional Programming Language extended with context as first class object, to support coordination activities in a distributed network of agents. We study coordination constructs which can be applied to sequential programs and distributed transactions. We give formal syntax and semantics for coordination constructs. The semantics for transaction expressions is given on top of the existing operational semantics in Lucx. The extended Lucx can be used for Internet-based agent applications.

Keywords: Multi-agent systems, coordinated transactions, Intensional Programming Language, coordination constructs.

1 Introduction

Our goal is to provide a programming model for a network of distributed agents solving problems in a coordinated fashion. We suggest an *Intensional Programming Language*, with *context* as first class objects and a minimal set of coordination constructs, to express the coordinated communication and computing patterns in agent-based systems. We give a formal syntax and semantics for the language and illustrate the power of the language with a realistic example.

Intensional Programming Paradigm. Intensional logic is a branch of mathematical logic used to precisely describe context-dependent entities. According to Carnap, the real meaning of a natural language expression whose truth-value depends on the context in which it is uttered is its *intension*. The *extension* of that expression is its actual truth-value in the different possible contexts of utterance. For an instance, the statement *"The capital of China is Beijing"* is intensional because its valuation depends on the context (here is the time) in which it is uttered. If this statement is uttered before 1949, the extensions of this statement are *False* (at that time, the capital was Nanjing). However, if it is uttered after 1949, the extensions of this statement are *True*. In the *Intensional Programming* (IP) paradigm, which has its foundations in Intensional Logic,

* This work is supported by grants from Natural Sciences and Engineering Research Council, Canada.

M. Baldoni et al. (Eds.): DALT 2005, LNAI 3904, pp. 205–222, 2006.

the real meaning of an expression is a function from contexts to values, and the value of the intension at any particular context is obtained by applying context operators to the intension. Basically, intensional programming provides intension on the representation level, and extensions on the evaluation level. Hence, intensional programming allows for a more declarative way of programming without loss of accuracy.

Lucid was a data-flow language which evolved into a Multidimensional Intensional Programming Language [1]. Lucid is a *stream* (i.e. infinite entity) manipulation language. The only data type in Lucid is a *stream*. The basic intensional operators are *first*, *next*, and *fby*. The four operators derived from the basic ones are `wvr`, `asa`, `upon`, and `prev`, where `wvr` stands for *whenever*, `asa` stands for *as soon as*, `upon` stands for *advances upon*, and `prev` stands for *previous*. All these operators are applied to streams to produce new streams. Example 1 illustrates the definitions of these operators (*nil* indicates an undefined value).

Example 1.

A	$= 1\ \ 2\ 3\ \ 4\ \ 5\ \ \ldots$
B	$= 0\ \ 0\ 1\ \ 0\ \ 1\ \ \ldots$
first A	$= 1\ \ 1\ 1\ \ 1\ \ 1\ \ \ldots$
next A	$= 2\ \ 3\ 4\ \ 5\ \ \ldots$
prev A	$=$ *nil* $1\ 2\ \ 3\ \ 4\ \ 5\ \ \ldots$
A fby B	$= 1\ \ 0\ 0\ \ 1\ \ 0\ \ 1\ \ \ldots$
A wvr B	$= 3\ \ 5\ldots$
A asa B	$= 3\ \ 3\ 3\ \ \ldots$
A upon B	$= 1\ \ 1\ 1\ \ 3\ \ 3\ \ 5\ \ \ldots$

The following program computes the stream $\langle 1,\ 1,\ 2,\ 3,\ 5,\ \ldots\rangle$ *of all Fibonacci numbers:*

```
result = fib
fib = 1 fby (fib + g)
g = 0 fby fib
```

Lucid allows the notion of context only implicitly. This restricts the ability of Lucid to express many requirements and constraints that arise in programming a complex software system. So we have extended Lucid by adding the capability to explicitly manipulate contexts. This is achieved by extending Lucid conservatively with *context* as a first class object. We call the resulting language *Lucx* (Lucid extended with *contexts*). Lucx has provided more power for representing problems in different application domains and given more flexibility of programming. We discuss Lucx, context calculus which is its semantic foundation, and multi-agent coordination constructs introduced in Lucx in Section 3.

Multiple-Agent Paradigm. By agent we mean software agents which can be personalized, continuously running and semi-autonomous, driven by a set of beliefs, desires, and intentions (BDI). Instead of describing each agent in isolation, we consider agent *types*. An agent type is characterized by a set of services. A generic classification of agent types, is *interface agent* (IA), *middle agent* (MA), *task agent* (TA), and *security agent* (SA). The MA type can be specialized into *arbitrator agent*, *match-maker agent*,

and *broker agent*. They play different roles, yet share the basic role of MA. All agents of an agent type have the same set of services. Agents are instances of agent types. An agent with this characterization is a black-box with interfaces to service its clients. It behaves according to the context and feedback from its environment. Two agents interact either directly or through intermediaries, who are themselves agents. An atomic interaction between two agents is a *query* initiated by an agent and received by the other agent at the interface which can service the query. An interaction among several agents is a collection of sequences of atomic interactions. Several methods are known in the field of distributed systems to characterize such behavior. Our goal is to explore intensional programming for expressing different interaction types, not in characterizing the whole collection of interactions.

In order to understand interaction patterns let us consider a typical business transactions launched by an agent. The agent acquires data from one or more remote agents, analyzes the data and computes some result, and based on the computed result invokes services from other agents. The agents may be invoked either concurrently or in sequence. In the latter case, the result of an agent's computation may be given as input to the next agent in the sequence. In the former case, it is possible that an agent splits a problem into subproblems and assigns each subproblem to an agent which has the expertise to solve it. We discuss the syntax and semantics for such coordination constructs in Lucx, under the assumption that an infrastructure exists to carry out the basic tasks expressed in the language.

A *configuration* is a collection of interacting agents, where each agent is an instance of an agent type. A configuration is simple if it has only one agent. An agent interacts with itself when it is engaged in some internal activity. More generally, the agents in a configuration communicate by exchanging messages through bidirectional communication channels. A channel is an abstract binding which when implemented will satisfy the specifications of the interfaces, the two ends of the channel. Thus, a configuration is a finite collection of agents and channels. The interfaces of a configuration are exactly those interfaces of the agents that are not bound by channels included in the configuration. In a distributed network of agents, each network node is either an agent or a configuration. Within a configuration, the pattern of computation is deterministic but need not be sequential.

There are three major language components in the design of distributed multi-agent systems:

- (ACL) agent communication language
- (CCL) constraint choice language
- (COL) coordination language.

These three languages have different design criteria. An ACL must support *interoperability* in agent community while providing the freedom for the agent to hide or reveal its internal details to other agents. A CCL must be designed to support agent problem solving by providing explicit representation of choices and choice problems. A COL must support transaction specification and task coordination among the agents. The two existing ACLs are *Knowledge Query and Manipulation Language* (KQML) and the FIPA agent communication language [5]. The FIPA language includes the basic concepts of KQML, yet they have slightly different semantics. Agents require a

content language to express information on constraints, which is encapsulated as a field within *performatives* of ACL. FIPA Constraint Choice Language (CCL) is one such content languages [6], designed to support agent problem solving by providing explicit representations of choices and choice problems.

In our works [2] [15], we have shown the suitability of Lucx, an *Intensional Programming Language* (IPL), for agent communication as well for choice representation. In this paper we extend Lucx with a small number of constructs to express task coordination. We are motivated by the following merits of Lucx.

1. Lucx allows the evaluation of expression at *contexts* which are definable as first class objects in the language. Context calculus in Lucx provides the basis for expressing dynamically changing situations.
2. Performatives, expressible as context expressions, can be dynamically introduced, computed, and modified. A dialog between agents is expressible as a *stream* of performatives, and consequently using stream functions such as *first*, *next*, and *fby*, a new dialog can be composed, decomposed, and analyzed based on the existing dialogs.
3. Lucx allows a cleaner and more declarative way of expressing the computational logic and task propagation of a program without loss of accuracy of interpreting the meaning of the program;
4. Lucx deals with *infinite entities* which can be any simple or composite data values.

2 Basics of Agent Coordination

A coordination expression, *Corde*, in its simplest form is a message/function call from one agent to another agent. A general *Corde* is a message/function from one configuration to another configuration. We introduce an abstract agent coordination expression $S.a(C)$ where S is a configuration expression, a is the message (function call) at S, and C is a context expression (discussed in Section 3). The context C is combined with the local context. If the context parameter is omitted it is interpreted as the current context. If the message is omitted, it is interpreted as a call to default method invocation at S. The evaluation of a *Corde* returns a result (x, C'), where x is the result and C' is the context in which x is valid. In principle, x may be a single item or a stream.

As an example, let B be a broker agent. The expression $B.sell(dd)$ is a call to the broker to *sell* stocks as specified in the context expression *dd*, which may include the stock symbol, the number of shares to be sold, and the constraints on the transaction such as date and time or minimum share price. The evaluation of the expression involves contacting agent B with *sell* and *dd* as parameters. The agent B computes a result (x, dd'), and returns to the agent A who invoked its services, for which the expression is $A.receive(C')$, where context C' includes x and dd'. In this example, dd' may include constraints on when the amount x can be deposited in A's bank account.

Composition Constructs. We introduce composition constructs for abstract coordination expressions and illustrate with examples. A coordination in a multi-agent system requires the composition of configuration expressions. As an example, consider an agent-based system for flight ticket booking. Such a system should function with minimal human intervention. An interface agent, representing a client, asks a broker agent

to book a flight ticket whose quoted price is no more than $300. The broker agent may simultaneously contact two task agents, each representing an airline company, for price quotes. The broker agent will choose the cheaper ticket if both of the quoted price are less than $300 and make a commitment to the corresponding task agent, then informs the interface agent about the flight information. If both prices are above $300, the broker agent will convey the information to the interface agent. The integrated activities of those agents to obtain the solution for the user is regarded as a transaction. Typically, the result from the interaction between two agents is used in some subsequent interaction, and results from simultaneously initiated interactions are compared to decide the next action. To meet these requirements, we provide many composition constructs,including sequential composition, parallel composition, and aggregation constructs. We informally explain the sequential and parallel composition constructs below.

The expression $E = S_1.a_1(C_1) > (x, C') > S_2.a_2(C_2)$ is a *sequential* composition of the two configuration expressions $S_1.a_1(C_1)$, and $S_2.a_2(C_2)$. The expression E is evaluated by first evaluating $S_1.a_1(C_1)$, and then calling S_2 with each value (x, C') returned by $S_1.a_1(C_1)$. The contexts C' is substituted for C_2 in the evaluation of $S_2.a_2(C_2)$.

The expression $S_1.a_1(C_1) \parallel S_2.a_2(C_2)$ is a *parallel* composition of the two configuration expressions $S_1.a_1(C_1)$, and $S_2.a_2(C_2)$. The evaluation of the two expressions are invoked simultaneously, and the result is the stream of values (x, C') returned by the configurations ordered by their time of delivery (available in C').

Example 2. *Let A (Alice) and B (Bob) be two interface agents, and M be a mediator agent. The mediator's service is to mediate a dispute between agents in the system. It receives the information from users and delivers a solution to them. In the expression*

$$((B.notifies > m_1) \parallel (A.notifies > m_2)) > M.receives(C') > (m_3, C'') > (B.receives \parallel A.receives)$$

the mediator computes a compromise (default function) m_3 for each pair of values (m_1, m_2) and delivers to Bob and Alice. Context C' includes $(m_1, m_2$ and the local context in M. Context C'' is a constraint on the validity of the mediated solution m_3.

The other constructs that we introduce are And, Or, and Xor constructs to enforce certain order on expression evaluations. The And ($\circ$) construct is to enforce evaluation of more than one expressions although the order is not important. The Or ($\wr$) construct is to choose one of the evaluations nondeterministically. The Xor ($\diamond$) construct defines one of the expressions to be evaluated with priority. In addition, we introduce Commit construct (*com*) to enable permanent state changes in the system after viewing the effect of a transaction. The syntax and semantics of these constructs in Lucx are given in Section 4. We can combine the where construct in Lucx with the above constructs to define parameterized expressions. Once defined, such expressions may be called from another expression.

3 An Intensional Programming Model for Distributed Networks of Agents

Lucx [2, 15] is a conservative extension of Lucid [1], an Intensional Programming Language. We have been exploring Lucx for a wide variety of programming applications.

In [14], we have studied real-time reactive programming models in Lucx. Recently we have given constraint program models in Lucx [15]. In this section we review these works for agent communication and content description.

3.1 An Overview of Intensional Programming Language: Lucx

Syntax and Semantic Rules of Lucx. The syntax of Lucx [2, 15], shown in Figure 1, is sufficient for programming agent communication and content representation. The symbols @ and # are context navigation and query operators. The non-terminals E and Q respectively refer to *expressions* and *definitions*. The abstract semantics of evaluation in Lucx is $\mathcal{D}, \mathcal{P}' \vdash E : v$, which means that in the definition environment $\mathcal{D}$, and in the evaluation context $\mathcal{P}'$, expression E evaluates to v. The definition environment $\mathcal{D}$ retains the definitions of all of the identifiers that appear in a Lucid program. Formally, $\mathcal{D}$ is a partial function $\mathcal{D}$: **Id** $\rightarrow$ **IdEntry**, where **Id** is the set of all possible identifiers and **IdEntry** has five possible kinds of value such as: *Dimensions*, *Constants*, *Data Operators*, *Variables*, and *Functions*. The evaluation context $\mathcal{P}'$, is the result of $\mathcal{P} \dagger c$, where $\mathcal{P}$ is the initial evaluating context, c is the defined context expression, and the symbol $\dagger$ denotes the overriding function. A complete operational semantics for Lucx is defined in [2, 15].

The implementation technique of evaluation for Lucx programs is an interpreted mode called *eduction* [1]. Eduction can be described as *tagged-token demand-driven dataflow*, in which data elements (tokens) are computed on demand following a data-flow network defined in Lucid. Data elements flow in the normal flow direction (from producer to consumer) and *demands* flow in the reverse order, both being *tagged* with their current context of evaluation.

Context Calculus. Informally, a context is a reference to a multidimensional stream, making an explicit reference to the dimensions and the *tags* (indexes) along each dimension. The formal definition is given in [15]. The syntax for context is $[d_1 : x_1, \ldots, d_n : x_n]$, where $d_1, \ldots, d_n$ are dimension names, and x_i is the tag for dimension d_i. An atomic context with only one dimension and a tag is called *micro context*. A context with different dimensions is called *simple context*. Given an expression E and a context c, the Lucid expression E @ c directs the eduction engine to evaluate E in the context c. According to the semantics, E @ c gives the stream value at the coordinates referenced by c.

In our previous papers [2, 14], we have introduced the following context operators: the *override* $\oplus$ is similar to function override; *difference* $\ominus$, *comparison* $=$, *conjunction*

$$
\begin{array}{ll}
E ::= id & C ::= \{E_1, \ldots, E_n\} \\
\quad \mid E(E_1, \ldots, E_n) & \quad \mid Box[E_1, \ldots, E_n \mid E'] \\
\quad \mid \texttt{if } E \texttt{ then } E' \texttt{ else } E'' & \quad \mid [E_1 : E_1', \ \ldots, \ E_n : E_n'] \\
\quad \mid \# & Q ::= \texttt{dimension } id \\
\quad \mid E \texttt{ @ } C & \quad \mid id = E \\
\quad \mid \langle E_1, \ldots, E_n \rangle E & \quad \mid id(id_1, \ldots, id_n) = E \\
\quad \mid select(E, E') & \quad \mid Q \ Q \\
\quad \mid E \texttt{ where } Q &
\end{array}
$$

Fig. 1. Abstract syntax for Lucx

Table 1. Precedence Rules for Context Operators

syntax	precedence
$C ::= c \mid C = C$ $\mid C \supseteq C \mid C \subseteq C$ $\mid C \mid C \mid C/C$ $\mid C \oplus C \mid C \ominus C$ $\mid C \sqcap C \mid C \sqcup C$ $\mid C \rightleftharpoons C \mid C \rightharpoonup C$ $\mid C \downarrow D \mid C \uparrow D$	1. $\downarrow, \uparrow, /$ 2. $\mid$ 3. $\sqcap, \sqcup$ 4. $\oplus, \ominus$ 5. $\rightleftharpoons, \rightharpoonup$ 6. $=, \subseteq, \supseteq$

$\sqcap$, and *disjunction* $\sqcup$ are similar to set operators; *projection* $\downarrow$ and *hiding* $\uparrow$ are selection operators; *constructor* $[\_ : \_]$ is used to construct an atomic context; *substitution /* is used to substitute values for selected tags in a context; *choice* $\mid$ accepts a finite number of contexts and nondeterministically returns one of them. *undirected range* $\rightleftharpoons$ and *directed range* $\rightharpoonup$ produce a set of contexts. The formal definitions of these operators can be found in [15]. The right column of Table 1 shows the precedence rules for the context operators, listed from the highest to the lowest precedence. The formal syntax of context expressions is shown in the left column of Table 1. Parentheses will be used to override this precedence when needed. Operators having equal precedence will be applied from left to right. Rules for evaluating context expressions are given in [15].

Example 3. *The precedence rules shown in Table 1 are applied in the evaluation of the well-formed context expression* $c_3 \uparrow D \oplus c_1 \mid c_2$*, where* $c_1 = [x : 3, y : 4, z : 5]$, $c_2 = [y : 5]$*, and* $c_3 = [x : 5, y : 6, w : 5]$, $D = \{w\}$*. The evaluation steps are as follows:*

[Step1]. $c_3 \uparrow D = [x : 5, y : 6]$ [$\uparrow$ Definition]

[Step2]. $c_1 \mid c_2 = c_1$ *or* c_2 [$\mid$ Definition]

[Step3]. Suppose in Step2, c_1 is chosen,

$c_3 \uparrow D \oplus c_1 = [x : 3, y : 4, z : 5]$ [$\oplus$ Definition]

else if c_2 is chosen,

$c_3 \uparrow D \oplus c_2 = [x : 5, y : 5]$ [$\oplus$ Definition]

A context which is not a micro context or a simple context is called a non-simple context. In general, a non-simple context is equivalent to a set of simple contexts [2]. In several applications we deal with contexts that have the same dimension set $\Delta \subseteq DIM$ and the tags satisfy a constraint p. The short hand notation for such a set is the syntax $Box[\Delta \mid p]$.

Definition 1. *Let* $\Delta = \{d_1, \ldots, d_k\}$*, where* $d_i \in DIM$ $i = 1, \ldots, k$*, and p is a k-ary predicate defined on the tuples of the relation* $\Pi_{d \in \Delta} f_{dimtotag}(d)$*. The syntax*

$$Box[\Delta \mid p] = \{s \mid s = [d_{i_1} : x_{i_1}, \ldots, d_{i_k} : x_{i_k}]\},$$

where the tuple $(x_1, \ldots, x_k)$*,* $x_i \in f_{dimtotag}(d_i)$*,* $i = 1, \ldots k$ *satisfy the predicate p, introduces a set S of contexts of degree k. For each context* $s \in S$ *the values in* $tag(s)$ *satisfy the predicate p.*

Table 2. Precedence Rules for Box Operators

syntax	precedence
$B ::= b \mid B \mid B$ $\mid B \boxdot B \mid B \boxtimes B$ $\mid B \boxplus B \mid B \downarrow D$ $\mid B \uparrow D$	1. $\downarrow, \uparrow$ 2. $\mid$ 3. $\boxdot, \boxplus, \boxtimes$

Many of the context operators introduced above can be naturally lifted to sets of contexts, in particular for *Box*es. We have defined three operators exclusively for *Box*es. These are ($\boxtimes$, $\boxplus$, and $\boxdot$). They have equal precedence and have semantics analogous to relational algebra operators. Table 2 shows a formal definition of *Box* expression B, and precedence rules for *Box* expressions. We use the symbol D to denote a dimension set.

An Example of a Lucx Program. Consider the problem of finding the solution in positive integers that satisfy the following constraints:

$\mathtt{x}^3 + \mathtt{y}^3 + \mathtt{z}^3 + \mathtt{u}^3 = 100$
$\mathtt{x} < \mathtt{u}$
$\mathtt{x} + \mathtt{y} = \mathtt{z}$

The Lucx program is given below:

```
Eval.B1, B2, B3 (x', y', z', u') = N
where
  N = merge ( merge( merge(x, y), z), u) @ B1 ⊠ B2 ⊠ B3;
  where
    merge(x, y) = if (x <= y) then x else y;
    B1 = Box[X, Y, Z, U | x^3 + y^3 + z^3 + u^3 = 100, x ∈ X, y ∈ Y, z ∈ Z, u ∈ U];
    B2 = Box [ X, U | x < u, x ∈ X , u ∈ U ];
    B3 = Box [ X, Y, Z | x + y = z, x ∈ X , y ∈ Y , z ∈ Z];
  end
end
```

3.2 Agent Communication

Lucx can be used as an *Agent Communication Language* (ACL) [2]. Due to the static nature of the predefined *communicative acts* (CAs) in FIPA and performatives in KQML, it is not possible to express the dynamic aspects in agent's states and requirements. Thus, inter-interoperability is not fully achieved. In using Lucx as ACL this problem is remedied. The performatives are expressed as context expressions, and a context is a *first class object* in Lucx, hence we are able to dynamically manipulate performatives. The name of a performative is considered as an expression, and the rest of the performative constitute a *context* which can be understood as a *communication context*, with each field except the name in the message being a *micro context*. The communication context will be evaluated by the receiver, by evaluating the expression at the context obtained by combining the micro contexts. In some cases, the receiver may combine the communication context with its *local context* to generate a new context.

The syntax of a message in Lucx from agent A is of the form $\langle E_A, E'_A \rangle$, where E_A is the message name and E'_A is a context expression. In an implementation E_A corresponds

to a function. The context E'_A includes all the information that agent A wants to convey in an interaction to another agent. A response from agent B to agent A will be of the form $\langle E_B, E''_B \rangle$, where E''_B will include the reference to the query for which this is a response in addition to the contexts in which the response should be understood. A conversation between two agents A and B is of the form $\langle \alpha_A;\ \beta_B \rangle$, where $\alpha_A = \langle E_A, E'_A \rangle$, and $\beta_B = \langle E_B, E''_B \rangle$. The semantics of a conversation is given in [2]. A *dialog* between two agents A, and B is a stream of conversations.

The operational semantics of Lucx is the basis for query evaluation. Consider queries that demand some form of response. The query from agent A $\langle E_A, E'_A \rangle$ to agent B is evaluated as follows:

1. agent B obtains the context $F_B = E'_A \oplus L_B$, where L_B is the local context for B.
2. agent B evaluates $E_A @ F_B$.
3. agent B constructs the new context E''_B that includes the evaluated result and information suggesting the context in which it should be interpreted by agent A, and
4. sends the response $\langle E_B, E''_B \rangle$ to agent A.

The above semantics should be changed for evaluating queries that do not necessarily demand some form of response. The query from agent A may be evaluated at any local context of B, and the result of evaluation may trigger an appropriate action in B. For instance, let B is a publisher agent which receives a material for publication in the form of a query of this type from a mobile agent A. A mobile agent roams around the web, collects information and delivers to his clients. Agent B may decide to process and publish the information delivered by A periodically or at a time that it "knows" to be most appropriate. Thus, steps 3 and 4 in the above semantics should be modified as follows: 3'. agent B "determines" the context E''_B for processing the information (evaluated in step 2), and 4'. processes the information at the context E''_B.

Example 4. *A query from agent* PTAc *about the Hotel information which was encoded as "ask-one" performative represented in Lucx as the expression* $E \ @ \ E'$, $E' = E_1 \oplus E_2 \oplus E_3 \oplus E_4 \oplus E_5$.

$\mathtt{E@[E1} \oplus \mathtt{E2} \oplus \mathtt{E3} \oplus \mathtt{E4} \oplus \mathtt{E5]}$
where
$\mathtt{E} = \text{"}\mathtt{ask-one}\text{"};\quad \mathsf{E}_1 = [\mathtt{sender} : \mathsf{PTAc}];\quad \mathsf{E}_2 = [\mathsf{content} : (\mathsf{Infor}_{\mathsf{Hotel}})];$
$\mathsf{E}_3 = [\mathtt{receiver} : \mathsf{HBA}];\quad \mathsf{E}_4 = [\mathsf{reply-with} : \mathsf{Infor-Hotel}];$
$\mathsf{E}_5 = [\mathtt{language} : \mathtt{LPROLOG}];$
end

3.3 Lucx as a Content Choice Language

CCL is designed to support agent problem solving by providing explicit representations of choices and choice problems. According to the requirements stated in [6], it should be possible in CCL to represent the sets of choices to be made, define operations that can be performed on the choices, declaratively state the relationships among choices, and introduce simple propositional statements. Lucx can be used for agent-based problem solving in the following four aspects that are normally attributed to a CCL [15].

1. *Modeling*: Choice Problem is modeled as CSP (Constraint Solving Problem) in Lucx, say by an agent A;

2. *Information Gathering*: Agent A either sends the whole CSP to several other agents or has the knowledge to decompose the CSP into several sub-CSPs, where each sub-CSP is solvable by an agent; after decomposition it sends to those agents and gets their feedback; Lucx, as ACL, can be used here;
3. *Information Fusion*: Agent A incorporates the feedbacks from other agents using context calculus and *Box* operations.
4. *Problem Solving*: Agent A may run simple problem solving algorithm such as the General CSP solver, or send the CSP components to problem solving agents, get their solutions, and unify it.

Example 5 illustrates the first step *Modeling* using Lucx as a CCL. That is, agent PTAc asks agent HBA whether those hotels (Marriott, Hilton, Sheraton) are available:

Example 5. $\mathtt{E@[E1 \oplus E2 \oplus E3 \oplus E4 \oplus E5]}$

```
where
  E = "ask − one";   E1 = [sender : PTAc];   E2 = [content : B1′];
    where B1′ = [Hc | h ∈ {Marriott, Hilton, Sheraton}]; end
  E3 = [receiver : HBA];   E4 = [reply − with : Hotel − Infor];
  E5 = [language : Lucx];
end
```

Example 6 illustrates the second step *Information Gathering* using Lucx as a CCL. That is, as a reply, agent HBA tells agent PTAc that Marriott and Hilton are available:

Example 6. $\mathtt{E'@[E'_1 \oplus E'_2 \oplus E'_3 \oplus E'_4 \oplus E'_5]}$

```
where
  E′ = "tell";   E1′ = [sender : HBA];   E2′ = [content : B1′′];
    where  B1′′ = [Hc | h ∈ {Marriott, Hilton}];   end
  E3′ = [receiver : PTAc];   E4′ = [in − reply − to : Hotel − Infor];
  E5′ = [language : Lucx];
end
```

An example of *Information Fusion* is the expression E_7 that appears in Example 8, Section 5. The expression E_8 in the same example illustrates *Problem Solving*.

4 Introducing Coordination Constructs in Lucx

In this section we conservatively extend Lucx with coordination constructs and give their formal syntax and semantics.

A *Corde* expression $S.m(C)$ arises when an agent A invokes S through method m in a context C. Using the Lucx notation introduced in Section 3.2 and the query in Example 5 it is easy to map this expression to a query $\langle E_A, E'_A \rangle$, where E_A represents m as the (ACL) performative name, and S, and C are respectively encapsulated as a context expression E'_A. That is, the representation of $S.m(C)$ is a Lucx performative. The result of evaluation of a *Corde* expression is also represented as a performative. We take a performative as a primitive service for a distributed agent system.

A *transaction* is a dialog between several agents. It is a distributed computation in a distributed agent systems, consisting of many steps of primitive services. Yet a transaction need not to be successful. We smoothly integrate primitive services to represent a transaction. Agent coordination is the set of transactions in the system.

4.1 Coordination Constructs in Lucx

The coordination constructs are n-ary constructs whose operands are performatives. We introduce a unary construct for "committing" the state changes in the evaluation of an expression.

Sequential Composition Construct $\gg$. The expression $a \gg b$ defines the sequential composition of performatives a and b. The content field of each performative is actually used to pass value/results between two agents, so the passed result is not shown in the syntax, however the value passing is implicitly supported in the semantics. Given two performatives a, b, the sequential composition $a \gg b$ is evaluated by first evaluating the performative a and using the result of its evaluation, which is encapsulated in the content field of the response performative of a, in evaluating the performative b. In general, the expression $a_1 \gg a_2 \ldots \gg a_k$ denotes the execution of performative a_{i+1} with the result of execution of a_i as an input, for $i = 1, \ldots, k-1$.

Parallel Composition Construct $\|$. The expression $a \parallel b$ defines a parallel composition of performatives a and b. Given two performatives a, b, the parallel composition $a \parallel b$ is evaluated by simultaneous execution of performatives a and b. In general, the evaluation of the expression $a_1 \parallel a_2 \parallel \ldots \parallel a_k$ will create k threads of computation, one for each performative. The result of evaluation is the merging of the results in time order.

Composition with no order $\circ$. The *and* ($\circ$) operator is used to force the conjoined evaluation of several expressions, without imposing any order on the evaluation. Given two performatives a, b, the expression $a \circ b$ defines that performatives a and b should be evaluated by the receiver agent, however the order of evaluation is not important. The result of evaluation is the set of results produced by the evaluation of performatives a and b. In general, the expression $a_1 \circ a_2 \circ \ldots \circ a_k$ defines that all the performatives a_i, $i = 1, k$ should be evaluated by the receiver agent.

Nondeterministic Choice Construct $\wr$. Given two performatives a, b, the expression $a \wr b$ defines that one of the performatives be evaluated nondeterministically. In general, $a_1 \wr \ldots \wr a_k$ denotes the evaluation of a nondeterministically chosen performative from the k operands. If the performative a_i is the nondeterministic choice, the result from the evaluation of the performative a_i is the result of evaluating the expression $a_1 \wr \ldots \wr a_k$.

Priority Construct $\diamond$. Given two performatives a and b, the expression $a \diamond b$ defines that performative a should be evaluated first, and if it succeeds, the performative b is to be discarded; otherwise, performative b should be evaluated by the receiver agent. In general, the expression requires that the performatives be evaluated deterministically in the order specified until the first successful evaluation of a performative. The result of evaluating the expression $a_1 \diamond \ldots \diamond a_k$ is that of the first successful evaluation.

Commit Construct *com*. This is a unary construct whose operand is a coordination expression. The result of evaluating $com(e)$ is that the state changes that happened during the evaluation of the coordination expression e are made permanent. The expression $a \gg (b \parallel c)$ will produce a result, however the state changes that happened due to the modifications of contexts will be ignored. The expression $com(a \gg (b \parallel c))$ will

produce the result of evaluating the expression $a \gg (b \parallel c)$, as well as make the state changes permanent.

Construct Binding. All the constructs have the same precedence, and hence the expression is evaluated from left to right. To enforce a particular order of evaluations, parenthesis may be used.

Properties and Examples. From the operational definitions for composition constructs we can derive the following properties:

1. The expression $e \gg e$ refers to two invocations of the performative e. The context may change after the first evaluation of e. The result of evaluating the expression is the result produced by the second invocation of e.
2. The construct $\gg$ is not commutative, but left associative.
3. The construct $\parallel$ is both commutative and associative.
4. The evaluation of the expression $e \parallel e$ makes two copies of e and simultaneously evaluates them. Hence, the invocation of expressions e, $e \gg e$, and $e \parallel$ have different effects.
5. With our semantics the evaluated result of the expression $e \parallel f$ consists of all possible outputs from invocations to e and f. If it is necessary to have them ordered according to their times of arrival at the host agent, the Lucx functions such as *before* can be used. Other Lucx functions can be used to gather (1) only the first value, (2) a tuple combining the first one from each, (3) the first value that satisfies a predicate, and (4) a tuple (x, y), where x is a result from an invocation to e, y is a result from an invocation to f such that the pair x, y satisfies a constraint.
6. The sequential construct does not distribute over the parallel construct. That is, $e \gg (f \parallel g) \neq e \gg f \parallel e \gg g$. In the evaluation of $e \gg (f \parallel g)$, the performative e is evaluated once, and the evaluation of expression $f \parallel g$ starts after that. In evaluating the expression $e \gg f \parallel e \gg g$, there are two parallel invocations to e, and the performatives f, g are invoked only after the corresponding results are received.
7. The parallel construct does not distribute over the sequential construct. That is, $(e \parallel f) \gg g \neq e \gg g \parallel f \gg g$
8. The commit construct distributes over other constructs. For an instance, $com(a \gg (b \parallel c)) = com(a) \gg (com(b) \parallel com(c))$

Example 7. *Let us consider a small example: a mediator agent M receives the diaries of a number of agents $A_1, \ldots, A_n$ and fixes a conflict-free meeting time for them. Let e_i denote the performative from A_i to M, and e'_i be the response performative from M to A_i. We give three different solutions:*

1. *The expression, $(e_1 \parallel \ldots \parallel e_n) \gg (e'_1 \parallel \ldots \parallel e'_n)$, when evaluated will give all possible conflict-free meeting times, assuming that agent M has the skill to compute it.*
2. *The expression, $(e_1 \circ \ldots \circ e_n) \gg (e'_1 \parallel \ldots \parallel e'_n)$, when evaluated may give an optimal conflict-free meeting time, assuming that agent M has the resources to save the constraints in the performative, formulates it to a CSP, and solves it. That is, the mediator must be a CSP solver.*

3. *The expression* $(e_1 \diamond \ldots \diamond e_n) \gg (e'_1 \parallel \ldots \parallel e'_n)$*, when evaluated will give the earliest conflict-free meeting time.*

4.2 Formal Syntax and Semantics of the Extended Lucx

The new syntactic rules are M rules shown in Figure 2. The new semantic rules are shown in Figure 3. The semantic rule $\mathbf{M}_{\textbf{sequential}}$ is valid whether or not the result of executing M is required for executing M'. Moreover, the semantic rule suggests that the performative M' must be evaluated only after the evaluation of M even when M and M' do not share data. The semantic rules $\mathbf{M}_{\textbf{parallel}}$ and $\mathbf{M}_{\textbf{choice}}$ are easy to understand. The semantic rule $\mathbf{M}_{\textbf{composition}}$ suggests that M and M' can be evaluated in any order without affecting the outcome. In the semantic rule $\mathbf{M}_{\textbf{priority}}$, we use *false* to suggest that the evaluation fails. Notice that partial evaluation in eduction procedure is not failure. Because of the distributive property, we can write the commit expression *com* M, where M is a coordination expression, as an expression in which each atomic component is *com* E, where E is a performative. Since we have already given the semantics for evaluating performatives, we contend that no separate semantics for *com* M is necessary.

$$
\begin{array}{lll}
M ::= M \gg M' & E ::= id & C ::= \{E_1, \ldots, E_n\} \\
\quad \mid M \parallel M' & \quad \mid E(E_1, \ldots, E_n) & \quad \mid Box[E_1, \ldots, E_n \mid E'] \\
\quad \mid M \circ M' & \quad \mid \texttt{if } E \texttt{ then } E' \texttt{ else } E'' & \quad \mid [E_1 : E'_1,\ \ldots,\ E_n : E'_n] \\
\quad \mid M \wr M' & \quad \mid \# & Q ::= \texttt{dimension } id \\
\quad \mid M \diamond M' & \quad \mid E \texttt{ @ } C & \quad \mid id = E \\
\quad \mid com\ M & \quad \mid \langle E_1, \ldots, E_n \rangle E & \quad \mid id(id_1, \ldots, id_n) = E \\
\quad \mid E & \quad \mid select(E, E') & \quad \mid Q\ Q \\
 & \quad \mid E \texttt{ where } Q &
\end{array}
$$

Fig. 2. Abstract syntax for the extended Lucx

5 Example

A general, but incomplete, description of the travel planning problem [15] is as follows: *Caroline would like to meet Liz in London for one of exhibition preview receptions at the Tate Gallery. These will be held at the beginning of October. Both Liz and Caroline have other appointments around that time, and will need to travel to London from their homes in Paris and New York.*

We suppose that there is an agent-based system making choices on when Liz and Caroline meet. Several agents assist each participant: a Personal Travel Assistant Agent (PTA) will communicate with Hotel Broker Agent (HBA), Air Travel Agent (ATA), and Diary Agent (DA). That is, the PTA for Caroline (PTAc) will get the hotel information from HBA, flight information from ATA, and meeting time from DA. After collecting and combining the information, it sends the information to Problem Solving Agent (PSA). The PSA will also receive the collected information from PTA for Liz (PTAl). The PSA computes the final solution and sends the solution to PTAs. PTAc communicates with HBA, ATA, and DA to make commitments. Once all these commitments are acknowledged, PTAc informs Caroline of the exact meeting time. As we remarked, the

$$\mathbf{M_{sequential}}: \frac{\mathcal{D},\mathcal{P}\vdash M:v \qquad \mathcal{D},\mathcal{P}\dagger[M\mapsto v]\vdash M':v'}{\mathcal{D},\mathcal{P}\vdash M\gg M':v'}$$

$$\mathbf{M_{parallel}}: \frac{\mathcal{D},\mathcal{P}\vdash M:v \qquad \mathcal{D},\mathcal{P}\vdash M':v'}{\mathcal{D},\mathcal{P}\vdash M \parallel M':v\dagger v'}$$

$$\mathbf{M_{choice}}: \frac{\mathcal{D},\mathcal{P}\vdash M:v \qquad or \qquad \mathcal{D},\mathcal{P}\vdash M':v}{\mathcal{D},\mathcal{P}\vdash M\wr M':v}$$

$$\mathbf{M_{priority}}: \frac{\mathcal{D},\mathcal{P}\vdash M:v \qquad or \qquad \mathcal{D},\mathcal{P}\vdash M:false \;\; \mathcal{D},\mathcal{P}\dagger[M\mapsto false]\vdash M':v}{\mathcal{D},\mathcal{P}\vdash M\diamond M':v}$$

$$\mathbf{M_{composition}}: \frac{\begin{array}{l}\mathcal{D},\mathcal{P}\vdash M:v \qquad \mathcal{D},\mathcal{P}\dagger[M\mapsto v]\vdash M':v' \\ or\ \mathcal{D},\mathcal{P}\vdash M':v \qquad \mathcal{D},\mathcal{P}\dagger[M'\mapsto v]\vdash M:v'\end{array}}{\mathcal{D},\mathcal{P}\vdash M\circ M':v'}$$

Fig. 3. New Semantic rules for the Extended Lucx

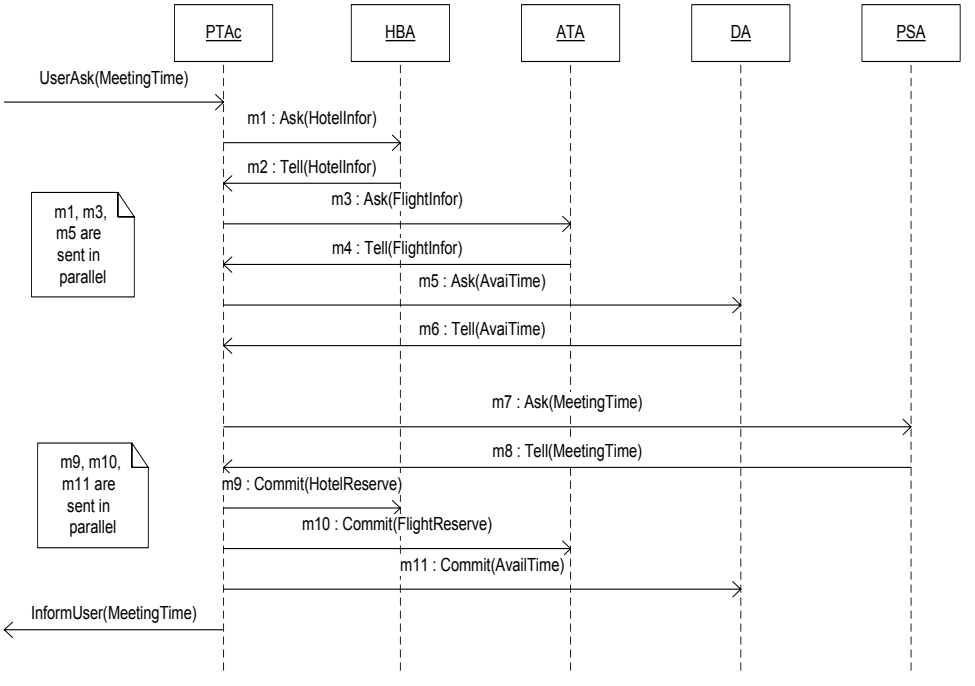

Fig. 4. Message Passing Sequence Diagram for TPE

agents themselves are not important, only their usage of Lucx is important. According to the above description, the message passing between agents is shown in Figure 4, and the Lucx program for the transaction is shown in Example 8:

Example 8. $(\mathtt{m}_1 \gg \mathtt{m}_2) \parallel (\mathtt{m}_3 \gg \mathtt{m}_4) \parallel (\mathtt{m}_5 \gg \mathtt{m}_6) \gg \mathtt{m}_7 \gg \mathtt{m}_8 \gg \mathtt{m}_9 \parallel \mathtt{m}_{10} \parallel \mathtt{m}_{11}$
where
$\mathtt{m}_1 = \mathtt{E}_1 @ [\mathtt{E}_{11} \oplus \mathtt{E}_{12} \oplus \mathtt{E}_{13} \oplus \mathtt{E}_{14} \oplus \mathtt{E}_{15}]$;

$$
\begin{array}{l}
\quad \text{where} \quad \mathtt{E}_1 = \text{"}\mathtt{ask-one}\text{"}; \quad \mathtt{E}_{11} = [\mathtt{sender} : \mathsf{PTAc}]; \quad \mathtt{E}_{12} = [\mathtt{content} : \mathtt{B}_1]; \\
\quad \text{where} \quad \mathtt{B}_1 = [\mathtt{H}_c \mid \mathtt{h} \in \{\mathtt{Marriott}, \mathtt{Hilton}, \mathtt{Sheraton}\}]; \quad \mathtt{end} \\
\quad \mathtt{E}_{13} = [\mathtt{receiver} : \mathsf{HBA}]; \quad \mathtt{E}_{14} = [\mathtt{reply-with} : \mathtt{Hotel-Infor}]; \\
\quad \mathtt{E}_{15} = [\mathtt{language} : \mathtt{Lucx}]; \ \mathtt{end} \\
\mathtt{m}_2 = \mathtt{E}_2 @ [\mathtt{E}'_{11} \oplus \mathtt{E}'_{12} \oplus \mathtt{E}'_{13} \oplus \mathtt{E}'_{14} \oplus \mathtt{E}'_{15}]; \\
\quad \text{where} \quad \mathtt{E}_2 = \text{"}\mathtt{tell}\text{"}; \quad \mathtt{E}_{21} = [\mathtt{sender} : \mathsf{HBA}]; \quad \mathtt{E}_{22} = [\mathtt{content} : \mathtt{B}''_1]; \\
\quad \text{where} \quad \mathtt{B}'_1 = [\mathtt{H}_c \mid \mathtt{h} \in \{\mathtt{Marriott}, \mathtt{Hilton}\}]; \quad \mathtt{end} \\
\quad \mathtt{E}_{23} = [\mathtt{receiver} : \mathsf{PTAc}]; \quad \mathtt{E}_{24} = [\mathtt{in-reply-to} : \mathtt{Hotel-Infor}]; \\
\quad \mathtt{E}_{25} = [\mathtt{language} : \mathtt{Lucx}]; \ \mathtt{end} \\
\ldots \\
\mathtt{m}_7 = \mathtt{E}_7 @ [\mathtt{E}'_{71} \oplus \mathtt{E}'_{72} \oplus \mathtt{E}'_{73} \oplus \mathtt{E}'_{74} \oplus \mathtt{E}'_{75}]; \\
\quad \text{where } \mathtt{E}_7 = \text{"}\mathtt{ask-one}\text{"}; \ \mathtt{E}_{71} = [\mathtt{sender} : \mathsf{PTAc}]; \ \mathtt{E}_{72} = [\mathtt{content} : \mathtt{B}'_1 \boxtimes \mathtt{B}'_2 \boxtimes \mathtt{B}'_3]; \\
\quad \text{where} \quad \mathtt{B}'_1 = [\mathtt{H}_c \mid \mathtt{h} \in \{\mathtt{Marriott}, \mathtt{Hilton}\}]; \\
\mathtt{B}'_2 = [\mathtt{F}_{\mathtt{from}_c} \mid \mathtt{f}_{\mathtt{from}c} \in \{[\mathtt{T}_{\mathtt{fl}} : \mathtt{10am}, \mathtt{T}_{\mathtt{fa}} : \mathtt{13pm}, \mathtt{F}_n : \mathtt{AC32}], [\mathtt{T}_{\mathtt{fl}} : \mathtt{16pm}, \mathtt{T}_{\mathtt{fa}} : \mathtt{19pm}, \mathtt{F}_n : \mathtt{AC38}]\}; \\
\quad \mathtt{B}'_3 = [\mathtt{T}_{3c} \mid \mathtt{t}_{3c} \in \{\mathtt{Oct.3-10am}, \mathtt{Oct.6-14pm}\}]; \quad \mathtt{end} \\
\quad \mathtt{E}_{73} = [\mathtt{receiver} : \mathsf{PSA}]; \quad \mathtt{E}_{74} = [\mathtt{reply-with} : \mathtt{Meeting-Time}]; \\
\quad \mathtt{E}_{75} = [\mathtt{language} : \mathtt{Lucx}]; \ \mathtt{end} \\
\mathtt{m}_8 = \mathtt{E}_8 @ [\mathtt{E}_{81} \oplus \mathtt{E}_{82} \oplus \mathtt{E}_{83} \oplus \mathtt{E}_{84} \oplus \mathtt{E}_{85}]; \\
\quad \text{where} \quad \mathtt{E}_8 = \text{"}\mathtt{tell}\text{"}; \quad \mathtt{E}_{81} = [\mathtt{sender} : \mathsf{PSA}]; \quad \mathtt{E}_{82} = [\mathtt{content} : \mathtt{B}''_1 \boxtimes \mathtt{B}''_2 \boxtimes \mathtt{B}''_3]; \\
\quad \text{where} \quad \mathtt{B}''_1 = [\mathtt{H}_c \mid \mathtt{h} \in \{\mathtt{Marriott}\}]; \\
\quad\quad \mathtt{B}''_2 = [\mathtt{F}_{\mathtt{from}_c} \mid \mathtt{f}_{\mathtt{from}c} \in \{[\mathtt{T}_{\mathtt{fl}} : \mathtt{10am}, \mathtt{T}_{\mathtt{fa}} : \mathtt{13pm}, \mathtt{F}_n : \mathtt{AC32}]\}; \\
\quad\quad \mathtt{B}''_3 = [\mathtt{T}_{3c} \mid \mathtt{t}_{3c} \in \{\mathtt{Oct.3-10am}\}]; \quad \mathtt{end} \\
\quad \mathtt{E}_{83} = [\mathtt{receiver} : \mathsf{PTAc}]; \quad \mathtt{E}_{84} = [\mathtt{in-reply-to} : \mathtt{Meeting-Time}]; \\
\quad \mathtt{E}_{85} = [\mathtt{language} : \mathtt{Lucx}]; \ \mathtt{end} \\
\mathtt{m}_9 = \mathtt{E}_9 @ [\mathtt{E}_{91} \oplus \mathtt{E}_{92} \oplus \mathtt{E}_{93} \oplus \mathtt{E}_{94} \oplus \mathtt{E}_{95}]; \\
\quad \text{where} \quad \mathtt{E}_9 = \text{"}\mathtt{commit}\text{"}; \quad \mathtt{E}_{91} = [\mathtt{sender} : \mathsf{PTAc}]; \quad \mathtt{E}_{92} = [\mathtt{content} : \mathtt{B}''_1]; \\
\quad \text{where} \quad \mathtt{B}''_1 = [\mathtt{H}_c \mid \mathtt{h} \in \{\mathtt{Marriott}\}]; \quad \mathtt{end} \\
\quad \mathtt{E}_{93} = [\mathtt{receiver} : \mathsf{HBA}]; \quad \mathtt{E}_{94} = [\mathtt{in-reply-to} : \mathtt{Hotel-Reserve}]; \\
\quad \mathtt{E}_{95} = [\mathtt{language} : \mathtt{Lucx}]; \ \mathtt{end} \\
\ldots \\
\mathtt{end}
\end{array}
$$

6 Conclusion

We have been exploring Intensional Programming Paradigm as a viable programming medium for different application domains. In this paper we have discussed our recent research results in enriching Lucx [2, 15], an Intensional Programming Language, with coordination constructs for programming multi-agent coordination in a distributed network of agents. We have given the formal syntax and operational semantics for the coordination constructs in Lucx. The language, thus extended, preserves the original syntax and semantics of Lucx which itself is a conservative extension of Lucid.

The two major aspects governing a practical realization of a MAS are

- the design and implementation of agents in a MAS, in particular their reasoning ability based on their belief and knowledge against those of the agents with whom they interact, and
- communication and protocols in a distributed architecture of MAS;

Ours is a contribution to the second aspect: we have focused on the language constructs for representing the control and coordination aspects in MAS. Hence, our work should be viewed as complementing the first aspect, which are discussed in [4, 8].

In [4] agents are represented as *Ordered Choice Logic Programs* (OCLP) for modeling their knowledge and reasoning abilities. The agents use answer set programming for representing their reasoning capabilities. In interactions, an agent sets higher preference to its own beliefs and rules than to suggestions and responses received from other agents. In this formalism decisions and situation dependent preferences can be explicitly stated. In our work situations are represented as contexts and situation dependent preferences are obtained by evaluating expressions at contexts. Moreover the semantics of conversation, as given in Section 3.2, uses the overwrite operator $\oplus$ to enforce preference to the belief and knowledge of an agent than to the responses of other agents.

A distributed architecture for MAS endowed with a *social layer* is discussed in [8]. The agents in the MAS are guided by a set of administrative agents which collectively employ a blackboard system for managing the norms. The global states of the MAS are stored in a tuple space, allowing the administrative agents to manage them in a distributed manner. This approach gives rise to a rule- based programming language to manage the global states. The paper does not discuss the features of such a programming language. In [14], while discussing the merits of Lucx for real-time reactive programming, we have shown that transition systems can be formally represented in Lucx. We believe that the global states of the MAS can be represented and manipulated in Lucx as transition systems. A more thorough investigation is necessary to explore and validate our claim in this regard.

Our language describes the coordination aspects at a higher level of abstraction than the scripting languages discussed in [11]. According to a definition given in [11] a scripting language introduces and binds a set of software components that collaborate to solve a particular problem. For the sake of comparison with our work, we may replace the phrase "component" with "agent" in this definition. Examples of scripting languages include Bourne Shell [3], Tcl [10], Perl [13], Python [12], and Javascript [7]. Higher-level abstractions are quite cumbersome to implement in these scripting languages. Except Bourne Shell no other scripting language supports concurrency. Most importantly, scripting languages have no formal semantics. As a consequence it is not possible to reason about the overall coordination behavior of the MAS if any one of the above scripting language were to be used to describe the MAS collaboration. The coordination constructs in Lucx have formal semantics. Lucx also serves both as ACL and CCL [2]. In addition we claim that agent computations and internal decision making can also be programmed in Lucx, provided we develop within Lucx a reasoning system.

Agent programming languages that are in practice today require a detailed program to express the control flow in the transactions. In many programming languages, it may be possible to express sequential as well as concurrent transactions, but it is not possible to express who the participants are in the transaction. In our language, a *Corde* expression expresses the different agents that are active in a collaborative transaction. Moreover our language is expressive in the sense that very complex pattern of transactions can be expressed in one Corde expression. One of our future work is on the

implementation environment and tool support for agent collaboration. It is too early to speculate now on the performance and cost-effectiveness of Lucx programs for agent collaboration.

There is a huge amount of literature on network models for distributed computing and most of them can be applied to multi-agent coordination. We are motivated by the need to simplify the semantics of agent coordination. So we have designed a small number of coordination constructs, for which formal operational semantics could be given. We permit arbitrary sequential and parallel compositions of *Corde* expressions. This enables us to express complex transaction activities among agents as well-formed Lucx expressions. Using the semantics it seems possible to determine the equivalence of arbitrary coordination expressions. We can use priority and sequential constructs to describe temporal aspects of coordination. We do not model resource sharing explicitly in the language. The rationale for this decision is that in the implementation of the parallel construct, we can use the solutions that have been proposed by the distributed computing research community.

In introducing the coordination constructs in Lucx, we are motivated by the recent work of Misra [9]. Yet, there are deep semantic differences in the two approaches. In our work an atomic *Corde* is a performative which is a context expression in Lucx. It includes the service requirements, in addition to a request for service. This contrasts with the term *site* [9], which is a general term for a *service*, including function names. There is a need to investigate the full set of semantic differences between Lucx constructs and the *Orc* expressions of Misra, and the suitability of Lucx for wide area computing.

Acknowledgement

We want to express our sincere thanks for the referees whose suggestions have greatly improved the readability of the paper.

References

1. E. Ashcroft, A. Faustini, R. Jagannathan, W. Wadge. *Multidimensional, Declarative Programming.* Oxford University Press, London, 1995.
2. Vasu S. Alagar, Joey Paquet, Kaiyu Wan. *Intensional Programming for Agent Communication.* Proceedings of DALT'04, Lecture Notes in Computer Science, Springer-Verlag, Vol. 3476, Page 239-255.
3. S. Bourne. *An Introduction to the UNIX Shell.* Bell Systems Technical Journal, 57(6):1971-1990, 1978.
4. Marina De Vos, Tom Crick, Julian Padget, Martin Brain, Owen Cliffe, and Jonathan Needham. *A Multi-Agent Platform using Ordered Choice Logic Programming.* Proceedings of DALT'05, Lecture Notes in Computer Science, Springer-Verlag (this volume).
5. FIPA Semantic Language Specification. *FIPA Specification repository*, FIPA-specification identifier XC00008G, September 2000 Foundation for Intelligent Physical Agents, Geneva, Switzerland.
6. FIPA CCL Content Language Specification. *FIPA TC C, Document Number: XC00009B* www.fipa.org/specs/fipa00009/XC00009B.html,2001/08/10
7. D. Flanagan. *Javascript: The Definitive Guide.* O'Reilly & Associates, 2nd edition, 1997.

8. A. García-Camino, J. A. Rodríguez-Aguilar, C. Sierra, and W. Vasconcelos. *A Distributed Architecture for Norm-Aware Agent Societies.* Proceedings of DALT'05, Lecture Notes in Computer Science, Springer-Verlag (this volume).
9. Jayadev Misra. *A Programming Model for the Orchestration of Web Services* Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM04), Beijing, China, Sep 2004.
10. J. K. Ousterhout. *Tcl and the Tck Toolkit.* Addison-Wesley, 1994.
11. Jean-Guy Schneider, Markus Lumpe, and Oscar Nierstrasz. *Agent Coordination via Scripting Languages* In Coordination of Internet Agents, Omicini, Zambonelli, Klusch and Tolksdorf eds., Springer-Verlag, 2001, ISBN 3-540-41613-7, pp. 153-175.
12. G. van Rossum. *Python Reference Manual.* Technical Report, Corporation of National Research Institute (CNRI), 1996.
13. L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl.* O'Reilly & Associates, 2nd edition, 1996.
14. K. Wan, V.S. Alagar, J. Paquet. *Real Time Reactive Programming Enriched with Context.* ICTAC2004, Guiyang, China, September 2004, Lecture Notes in Computer Science,3407,Page 387-402, Springer-Verlag.
15. Kaiyu Wan. *Lucx: Lucid Enriched With Contexts*, Ph.d Thesis, Department of Computer Science, Concordia University, Montreal, Canada, January 2006.