A proposal to merge Multiple Tuple Spaces, Object Orientation, and Logic Programming

Vincenzo Ambriola  Giovanni A. Cignoni
Laura Semini

Dipartimento di Informatica
Università di Pisa

Abstract

We define a new language that naturally satisfies the requirements of concurrency support, modularity, and declarativity. Although some classes of existing languages already cover a subset of this requirements, there is no example of a programming language entirely fulfilling them.

We propose to merge three programming paradigms, Multiple Tuple Spaces, Object Orientation, and Logic Programming, as a solution to our requirement list. Each paradigm is well characterized and satisfies one of the given requirements. Moreover, we claim that the merging will reveal new directions in the inception and the development of future programming languages.

1 Introduction

A programming language should be conceived for writing computer programs. In the Oikos Project [13] there was the need for a powerful language to implement the cooperation policies that rule the software development process.

Starting from these assumptions, three years ago we decided to design a new language. During this enterprise we realized that our language could
fulfill the needs of a community of users larger than the group of researchers involved in writing software process programs in the Oikos Project.

According to the principles of *Software Process Modelling* [8], cooperation policies should be expressed in a high level language that exhibits three properties: modularity, reusability, and concurrency support (possibly with a distributed implementation). From a more general point of view, these needs are shared by builders of large distributed systems.

Our previous experience with languages based on *Multiple Tuple Spaces* led us to define a strategy to simultaneously satisfy our requirements. The outcome of our quest was the idea to merge the Multiple Tuple Spaces paradigm with two other programming paradigms: *Object Orientation* and *Logic Programming*.

We also discovered some useful side effects that derive from the merging. For instance, dynamicity is a natural result of applying the Object Orientation perspective to the Multiple Tuple Spaces concept, and the use of unification (from Logic Programming) rather than pattern matching, for associative access to tuples, improves the declarativity of the language.

In the following sections we present our requirements (Section 2) and the programming paradigms that we choose to fulfill them (Section 3). In Section 4 we describe the computational model obtained by merging these paradigms. Then we present eta, the programming language that instantiates the proposed model (Section 5), its syntax (Section 6), and some semantic issues (Section 7). In Section 8 we discuss related works.

## 2 Requirements for a language

Our goal is to satisfy the needs of the programmer who will use the language to build large distributed software systems, as is the case in process-centered software development environments. Towards this aim, we design a programming language that fulfills a strict list of requirements. We want our language to be *concurrent, modular, and declarative*.

### 2.1 Concurrency

A concurrent language or, more precisely, a language that naturally supports concurrency must provide:
• syntactic constructs to define autonomous entities of computation, i.e.,
  the processes, that can be executed in true parallelism;

• mechanisms for process synchronization and cooperation: these mecha-
  nisms may belong either to the share approach, where processes refer
  to a common storage and synchronize by writing and reading on it, or
  to the communicate approach, where processes synchronize by sending
  and waiting for messages.

The expressive power of a concurrent programming language depends on
the mechanisms that the language provides to describe concurrent compu-
tations without using operating system calls. To control the inherent non-
determinism of concurrent computations, a concurrent language should also
provide mechanisms to express priority and mutual exclusion.

Parallelism is different from concurrency. For instance, a sequential lan-
guage may have a parallel implementation that increases the performance of
program execution still preserving the semantics. By the way, a concurrent
language is the needed tool to implement parallel algorithms and distributed
systems.

The large growth of local area networks is shifting the focus from central-
ized to distributed architectures. Moreover, distribution can be viewed as an
external constraint in the development of concurrent systems, i.e., when it is
practically impossible to share memory. Although shared memory can always
be implemented on a distributed environment, using a communication based
language that naturally supports distribution is an evident advantage. From
this point of view, in our requirement list concurrency covers the suitable for
distributed applications issue.

2.2 Modularity

A modular system is made of a collection of logically autonomous entities, the
modules, connected by a simple framework. Modularity is the key to develop-
ing large systems, to improving maintenance, and to supporting reuse. This
definition, with the listed benefits, well applies to software process programs
too.

• A module must be a syntactic entity of the language. The implement-
ation of the language must support separate compilation of modules
and dynamic loading of pre-compiled modules.

- The language must support information hiding. Each module declares a private part (used to keep the state of the module) and a public part (the interface) as the mean to express the interaction with the other modules of the system.

- The language must provide explicit constructs to define the module interface. The interaction mechanisms must refer to a well defined semantic schema, e.g., procedure call or message passing.

In our requirement list, modularity mainly covers the suitable for large programs issue.

Several aspects of modularity can be viewed under the concurrency perspective. The modular design of a software system often elicits the existing parallelism and identifies modules with concurrent entities. This approach, combined with a dynamic loading capability, suggests a dynamic language for the implementation of open systems, a language such that entirely new components can be added without halting the running system. Moreover, dynamic loading can be enhanced by distribution or load balancing options.

### 2.3 Declarativity

A programming language is declarative if its programs describe the results of a computation rather than the way such results are achieved, or, in other words if the programmer can write “what” he/she wants and not “how” to get it [12].

In some cases this property may be a limitation, e.g., when the programmer’s task is to implement “something” (a data structure, an algorithm, a whole system) given in a procedural way. In general, however, declarativity is an advantage for the programmer and for this reason, in our requirement list, declarativity covers the usable for programmers issue. In particular, we believe that this property is very useful when the programming goal is to build a concurrent application. In these cases, it is essential to describe events and reactions, since the procedural alternative would be to explicitly receive external inputs, test complex multiple conditions, and produce proper outputs.
3 Programming language paradigms

Our quest for a language suitable for Software Process Modelling started with the design, the implementation, and the experimentation of Multiple Tuple Spaces based languages [13]. Recently we formulated an elegant and natural strategy to satisfy the requirements imposed by our research in Software Process Modelling [3]. We decided to investigate on the advantages of merging the Multiple Tuple Spaces paradigm with Object Orientation and Logic Programming. Each of these paradigms is well characterized and fully satisfies one of the requirements listed above.

3.1 Multiple Tuple Spaces

Concurrent languages based on the Tuple Space paradigm exploit the shared memory approach. The store (i.e., the tuple space) is a multiset of tuples that represents the computation state. Autonomous computational entities, called agents, cooperate by reading and writing on the tuple space [5].

The presence of some specific configurations of the tuple space causes the agents to react with a transition. A transition is described by a rule, i.e., a guarded action on the tuple space, a computation, and a write operation on the tuple space.

A guarded action is a read or a read-and-delete operation associatively performed on the tuple space. The computation is purely functional, without side effects on the tuple space. Since the behaviour of the agents depends on the state of the tuple space, the tuple space and its agents can be seen as a closed reactive (to its own state) system.

The Multiple Tuple Spaces paradigm is a far-reaching extension of the Tuple Space paradigm: a system is composed of a set (in some cases a hierarchy [7]) of tuple spaces, instead of a single one. Tuples are sent through the system from one tuple space to another, exploiting explicit addressing. Tuple spaces are reactive components of a distributed system (they do not share anything). As in Tuple Space paradigm they react to changes to their state, but in this case, the state can be modified by external messages too.

This extension orthogonally combines the shared memory and the message passing paradigms, providing a powerful model to develop large systems as collections of components. Programming languages based on Multiple Tuple Spaces include Linda-3 [10], ESP [7], and Pâté [4]. A model dealing with
both these aspects provides two grains of parallelism: coarse grained parallelism between the tuple spaces, and fine grained parallelism between the agents belonging to each component. Each form of parallelism refers to a precise cooperation model: message passing and shared memory.

These features, both with respect to the parallelism grain and to the cooperation models, resemble multitasking implementations commonly used in the last generation of operating systems, like heavyweight and lightweight processes in Mach [16].

3.2 Object Orientation

The Object Orientation paradigm formalizes the notions of class of objects, information hiding, inter-objects interaction policies, and inheritance. They all refer to the wider concept of modular programming.

- The notion of class is an extension of the primitive concept of type. The procedures to operate on data structures belong to the type definition. The assignment of abstract names to collections of data and operations is the first step to achieve modularity.

- A module (a class) must hide the details of its internal implementation from the other components of the system ...

- ... offering only a defined interface to its functionalities. The interface establishes which functionalities are public, which are private, which are restricted to a subset of the other components.

- The development of new modules by extension and redefinition of existing ones is a powerful tool to build highly structured programs and to reuse existing software.

Far from being a “silver bullet”, Object Orientation is the best paradigm to support adaptation and maintenance, two issues of actual interest and research in software development [14] and, of more interest for us, in software process programming.
3.3 Logic Programming

Logic Programming is one of the most advanced and refined approaches for solving complex programming problems. It comes from the synergism between logic (declarativeness) and programming (procedurality) [17, 12]. The rôle of logic is to provide a formal framework to define the semantics of programs and to make them more understandable. The rôle of programming is to deal with the computational aspects of programs and, from the logicians’ point of view, to control how the proof (i.e., the program execution) proceeds.

Logic programming couples the power of a single inference rule (resolution) with an extremely general form of pattern matching (unification). Prolog is the most successful realization of this computational model.

4 Merging the paradigms

Our idea is to specialize the object oriented model to the case of Multiple Tuple Spaces languages, treating a tuple space and its agents as a unique programming entity (an object).

4.1 Logic Programming + Multiple Tuple Spaces . . .

The combination of tuple spaces with logic programming yields a paradigm that well integrates cooperation and declarativity. Shared Prolog [5] belongs to this model, whose characteristics are:

- a tuple is a logical fact, a Prolog atom, and the interaction between the tuple space and the agents is controlled by the unification algorithm;
- a restricted set of Prolog (with no side effects on the tuple space and on the Prolog knowledge base) is used to perform the functional computation associated to each rule.

The expressive power of a tuple space language is enhanced by unification: agent descriptions can include complex conditions on the tuple space, yet expressed in a very concise manner. The cooperation and the computation facets of an agent (the interaction with the tuple space and the computation
fired by the specified configurations of the tuple space, respectively) can be uniformly described by the same formalism.

ESP and Pâte are two examples of languages that extend the single tuple space of SP to the Multiple Tuple Spaces paradigm. A common feature of these languages is the ability to write (send) tuples to other tuple spaces. Pâte also provides a declarative way to control rule firing sequences inside agents by means of path expressions [4].

4.2 ... + Object Orientation ...

Objects provide an interface that, in a context where objects are concurrent entities, is the specification of the interaction with other objects of the system.

As a concurrent object, the interface of a tuple space is defined by the tuples accepted as input and those produced as output. In our model, we prevent unexpected tuples from reaching a tuple space using unification (from Logic Programming) to control which tuples can pass through the declared object interface.

With respect to inheritance, we follow the approach to define new tuple spaces by extending their interface and redefining part of the rules that specify the object behaviour.

4.3 ... = eta

We have shown how three independent programming models can be merged in a new paradigm. The appealing features of this merging prompted us to make the new paradigm concrete by defining a programming language. The language eta (Everything but Assignment) is a Multiple Tuple Spaces and Logic language that borrows the programming style of the object oriented paradigm [3].

In defining eta, our goal is the design of a language for the development of large, real, and distributed systems. Consequently, we take into account the notions of control and reliability. For instance, considering the communications between tuple spaces, we want to detect whether either data will eventually reach their destination or the communication failed, e.g., because of an address error, an error of the support, or a violation of the input interface of the receiver. Similarly, we require primitives to express control
information: sequencing, as well as alternation, priority, and mutual exclusion of agent actions. This requirement leads to a further improvement of the path expression mechanism already exploited in Pâté.

In the next section we present the characteristics of eta, showing both how it instantiates the proposed paradigm, and how it accomplishes our goal, enhancing control and reliability.

5 The language eta

A program in eta is a collection of class definitions, where each class defines the behaviour of a set of possible objects.

An object is an instance of a class. The state of an object consists of a passive part (a multiset of tuples) and an active part (a multiset of threads). Tuples have the form of Prolog atoms. Objects communicate by sending and receiving messages, exploiting explicit addressing: messages are tuples added to the receiver object’s state, and addresses are object names. The event of receiving a message causes the incoming tuple to be added to the object state. This behaviour differs from that of most object oriented languages where the arrival of a message is interpreted as an operation or a procedure invocation.

5.1 Interface and information hiding

In eta each object provides an interface to the external world in terms of the tuples accepted as input from other objects and the tuples produced as output and sent through the system.

\[
\begin{align*}
\text{class} & \quad \text{air\_company}(\text{Name}) \\
\text{input} & \quad \{\text{req}(\text{Name}, \ldots)\} \\
\text{output} & \quad \{\text{booked}(\ldots)\} \\
& \quad \cdots \cdots \cdots \cdots \cdots \cdots \cdots \\
& \quad \text{interface} \\
& \quad \text{private part}
\end{align*}
\]

The name of a class is a term. The functor of the term identifies the class, while the arguments are free variables (parameters of the class) that are
instantiated when an object is activated. **Input** and **output** specify the tuples accepted as input, or sent to other objects, respectively. In the example, the object `air.company(c.name)` only accepts tuples that unify with `req(c.name, _)` and produces tuples of the form `booked(_, _)`.

Any message not accepted as input by the addressee raises a run-time error. The sender is acknowledged of this event. This mechanism is a simple way to prevent unexpected tuples from reaching a tuple space. Moreover, as the interface is the only visible and public part of an object, its behaviour can be freely changed by keeping the interface invariant.

### 5.2 Behaviour and control

The behaviour of an object is defined by its threads. A thread is the orthogonal combination of rules and path expressions: rules describe actions in terms of reactive behaviour, path expressions express control on rules activations.

A rule is composed of a guard, an outside computation, an interaction, and a postcondition.

<table>
<thead>
<tr>
<th>booking:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{req(_, Flight, N, R, C), seats(Flight, Free)} map(Flight, M)</code></td>
</tr>
<tr>
<td>`</td>
</tr>
<tr>
<td><code>{booked(Flight, Seats)}@C</code></td>
</tr>
<tr>
<td><code>{seats(Flight, Free')}</code></td>
</tr>
<tr>
<td><code>body</code></td>
</tr>
<tr>
<td><code>{seats(Flight, Free), req(Fight, N, any, C)}</code></td>
</tr>
<tr>
<td><code>target</code></td>
</tr>
<tr>
<td><code>{seats(Flight, Free'), tryLater(C, Flight, Seats)}</code></td>
</tr>
<tr>
<td><code>interface</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>in guard</th>
<th>rule name</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>guards`</td>
</tr>
<tr>
<td>`</td>
<td>commit operator`</td>
</tr>
<tr>
<td>`</td>
<td>body`</td>
</tr>
<tr>
<td>`</td>
<td>outside`</td>
</tr>
<tr>
<td>`</td>
<td>interaction`</td>
</tr>
<tr>
<td>`</td>
<td>computation`</td>
</tr>
<tr>
<td>`</td>
<td>success`</td>
</tr>
<tr>
<td>`</td>
<td>body failure`</td>
</tr>
<tr>
<td>`</td>
<td>target failure`</td>
</tr>
<tr>
<td>`</td>
<td>postcondition`</td>
</tr>
<tr>
<td>`</td>
<td>interface`</td>
</tr>
</tbody>
</table>

The **guard** is a sequence of read (**read guard** and input conditions (**in guard**): the former is a compound goal on the object state, the latter defines the multiset of atoms to be removed from the state. The **outside computation** has no side-effects on the object state: the **body** defines a functional calculus; the **interaction** defines the tuples to be sent to external objects. Finally, the **postcondition** defines a list of tuples to be written in the object state and the
treatment of possible failures that occur during the execution of the body or of the interaction.

In the example, the action described by rule booking is taken when a request from a client $C$, for $N$ places ($R$ specifies preferences, e.g., window), on flight number Flight, is found. The tuple recording the number of free seats on that flight is removed to be updated, while the map of the flight is only accessed in read mode. The body searches the requested seats. In case of success, the booked acknowledgement is sent to the client. Otherwise, the request is reformulated, ignoring the preferences. Tuples in the target or interface parts are written in the object state in case of failure of the communication with the client (see Sections 7.3.3 and 7.3.5).

Path expressions are a declarative way to specify control and constrain a wild non-determinism in rules activations [4]. According to this form of control, the activation of a rule is subjected to two conditions: it must satisfy path expression constraints, and the current state of the object must satisfy the rule’s guard. No backtracking on rule activations is allowed according to a “don’t care” nondeterminism policy [15].

A path expression is similar to a regular expression over the alphabet of rule names, with the following exceptions:

- a priority operator $\prec$ extends the choice operator, allowing to express a priority between two alternative paths;

- an optionality construct is provided, as a shorthand for $Exp | \epsilon$ (with $\epsilon$ the empty path);

- a failure operator $\text{rule}\{Exp; Exp; Exp; Exp\}$ takes into account four possible different outcomes of a rule evaluation (success or failure of the body and of the interaction).

Path expressions can also specify semaphores (see Section 7.2).

We extend our working example on the class $\text{air\_company}(\_)$, providing an example of thread:
A thread terminates when it reaches the end of one of the (finite) paths of its path expression. When all the threads of an object are terminated, the object terminates.

5.3 Modularity and reusability

A class definition can inherit from another one, exploiting an is-a declaration.

<table>
<thead>
<tr>
<th>class</th>
<th>company(Name, Over)</th>
<th>interface : extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>is-a</td>
<td>air_company(Name)</td>
<td></td>
</tr>
<tr>
<td>output</td>
<td>{overbooked(_)}</td>
<td>private part : overriding</td>
</tr>
<tr>
<td></td>
<td>...........................................</td>
<td></td>
</tr>
</tbody>
</table>

We adopt an extensional flavour of inheritance for the object interfaces, while we allow overriding for the object internal behaviour, uniformly with the encapsulation aspects of the language. An object can send and receive only messages that belong to a superset of those sent and received by an object of the superclass, while reactions to an identical message can be different.

6 Syntax

In this section we introduce the formal syntax of eta, using the BNF notation. The design choices made during the definition of the language have been oriented to modularity: a module corresponds to a syntactic entity and the different parts of each entity are loosely connected. Following this approach, we split the definition of eta in three parts, each described in the next three subsections. In Subsection 6.1 we give the general structure of a
class definition, leaving undefined the production rules for threads and rules. They will be provided in Subsections 6.2 and 6.3, respectively.

Syntactic categories are capitalized. The symbols ::= and | belong to the BNF notation, while { } [] ( ) , : . | @ > < * + - belong to the language. Few syntactic categories are not capitalized and not expanded, for those we refer to the standard Prolog syntax.

Keywords of eta, as well as primitives, are in boldface. The former set includes: class, is-a, input, output, contents, initial, critical, threads, rules, with, end, body, interaction, and target; the latter includes: getlist, putset, new, not, father, and self.

6.1 Classes

In the next figure we provide the syntax for the most general class definition. Actually, a class definition can be a proper subset of the listed parts (only class and end are always required), with the following constraints: in the case of class definition by inheritance, is-a is mandatory; at least init, thread, rules, and one among input, output, and contents must be specified when defining an entirely new class.

Variables appearing as parameters in a class name can bind variables appearing in all the parts but contents, output, and with. In contents and output we allow only free atoms, as we want to statically verify if they meet rule postcondition or interaction; atoms in input can contain variables bound to the class parameters, or constants.

The set of atoms that represent the object initial state and the set of initial threads, i.e., those activated with the object, are specified by initial. A critical region specifies a set of atoms and is identified by a name that ranges over the alphabet of the semaphores.

When a class is derived from another one, say A is-a B, the parts input, output, contents, and initial of the two classes are disjoint: in A we simply define the extentsions with respect to B. On the other hand, we allow the overriding of rules, threads, critical regions, and Prolog clauses (predicate definitions).
6.2 Threads

A thread consists of a name and a path expression. We define the syntax of path expressions using three syntactic categories: `Rname`, `Exp`, and `Semaphore`. `Rname` denotes a rule name, `Exp` a path expression, and `Semaphore` a semaphore name: the grammar generates the set of path expressions over the alphabet of rule and semaphore names.
Threads ::= ThName : Exp | Threads , Threads
Exp ::= ε % empty
  | RName % rule name
  | Exp* % iteration
  | Exp Exp % sequencing
  | Exp | Exp % choice
  | Exp > Exp % priority
  | Exp* > Exp | Exp* ≺ Exp % ""
  | [ Exp ] % optionality
  | ( Exp ) % precedence
  | RName { Exp ; Exp ; Exp ; Exp } % failure
  | Semaphore+ Exp Semaphore− % critical region

RName ::= constant

6.3 Rules

A rule has a name and two parts separated by the commit operator |. Both parts can be empty.

Rules ::= Rule | Rules Rules
      Rule ::= r ; % empty rule
         | RName : Guard | Post % normal rule
         | RName : Guard
         | RName :

The first part, Guard, is the rule guard. The second, Post, is a conjunction of the outside computation and the postcondition of the rule.

Guard ::= { In }
      | Read
      | { In } Guard
      | Read { In } Guard
In ::= Atom | Get | In , In
Read ::= Atom | Get | PrologPrim | EtaPrim | Read , Read

Get ::= getlist(Atom , X) | getlist(Atom , X , PrologPrim)
PrologPrim ::= Atom = Atom | Atom != Atom | ... (\( = == \) is var ...)
EtaPrim ::= self(X) | father(X) | not Atom

X ::= variable
Both \textit{Read} and \textit{In} are sequences of atoms and \textit{Get} primitives. The read guard (\textit{Read}) may contain both \textit{eta} primitives (\textit{EtaPrim}) and primitives derived from Prolog (\textit{PrologPrim}), to express relations between the facts of the guard.

\begin{center}
\begin{verbatim}
Post ::= Body
      Interaction
      Success
      FailBody
      FailInteraction
      FailTarget

Body ::= $\epsilon \mid \text{Atom}$.
Interaction ::= $\epsilon \mid \text{Out} \oplus \text{Term} \mid \text{new}(\text{Term}) \mid \text{Interaction} \mid \text{Interaction}$
Success ::= $\epsilon \mid \text{Out} \oplus \text{Th}\text{Name} \mid \text{Success} \mid \text{Success}$
FailBody ::= $\epsilon \mid \text{body}\{ \text{Atoms} \}$
FailInteraction ::= $\epsilon \mid \text{interaction}\{ \text{Atoms} \}$
FailTarget ::= $\epsilon \mid \text{target}\{ \text{Atoms} \}$

Out ::= \{ Atom \}
   | \text{putset}(X)
   | \text{putset}(X, Atom)
   | \text{putset}(X, Atom, PrologPrim)
\end{verbatim}
\end{center}

\textit{Post} consists of the body, the interaction, and the postcondition. The body (\textit{Body}) is a Prolog atomic atom. \textit{Interaction} is a sequence of send (\textit{@}) and activations of new objects (\textit{new}(\textit{Term})). The postcondition consists of a \textit{Success} followed by the parts \textit{FailBody}, \textit{FailInteraction}, and \textit{FailTarget}. \textit{Success} is a sequence of write declarations (atoms or sets of atoms to be written on the object state) and thread activations. Expressions of the kind \texttt{putset()} specify lists of atoms.

\section{Semantics}

An \textit{eta} system is a set of executing objects that are dynamically created. System execution consists in the concurrent execution of the objects, i.e., the evaluation of the rules defined in the object class, according to the constraints expressed by the path expression of every thread occurrence. Rules
in different thread occurrences are evaluated concurrently. A strong fairness property on rule selection is guaranteed, according to the results presented in [9].

7.1 Interface: synchronization and reliability

At the semantic level, we model the interaction between two objects as follows: after sending a message, either the message reaches the destination or a failure in the communication is acknowledged. This synchronization constraint has a very low impact on the system parallelism and represents a sufficient condition to build reliable systems. Moreover, exploiting path expressions, we can force the sender of a message to wait for an answer to its request. Synchronous communications can be easily programmed.

Communication between objects can fail: the addressee may not exist or it may not consider an incoming message as a possible input. The dynamic characteristics of \( \eta \) make impossible to detect these failures at compile-time: they are handled by the run-time support of the language.

7.2 Behaviour: threads

A thread defines all the feasible sequences of actions. A path expression associated with each thread defines alternation, sequencing, and priority constraints on rule activation, while the rules define the reactive behaviour of actions. Path expressions can also define semaphores: atoms specified in a critical region cannot be consumed by other threads than the one which entered the critical region. Critical regions are especially needed when the predicate get\textbf{list} is involved in the guard of a rule.

In the next subsection we show, by means of examples, how to derive a partial order over the set of rule names by making priorities explicit, while in Section 7.2.2 we define a finite state machine that recognizes a path expression.

According to those definitions we say that a rule \( r \), which occurs in a path expression \( p \), can be evaluated in a state \( (s_1, s_2) \), where \( s_1 \) is the state of the object and \( s_2 \) is the state of the finite state machine that recognizes \( p \), if:

1. the rule guard is satisfied in \( s_1 \) (see Section 7.3);
2. there is an outgoing edge from \( s_2 \) labeled \( r \);
(3) \( r \) is maximal, with respect to the partial order induced by the priorities, in the poset of rules satisfying (1) and (2).

Multiple occurrences of the same rule can appear in a path expression. We define a map that uniquely renames each occurrence of a rule name in the path expression. In the next subsections we assume that path expressions are properly renamed.

### 7.2.1 Priorities

We call \( S \) the function that defines the partial order (over rule names) induced by the priority operators of a path expression.

\[
S : \text{Exp} := \bigwedge_{\text{partial order}} \text{Exp} \times \text{Exp}
\]

Although the definition of \( S \) is out of the scope of this paper, for the sake of completeness in the presentation of \( \text{eta} \) we provide the following examples:

\[
S((a \succ b) \prec c) = \{(a, b), (c, a), (c, b)\}
\]

\[
S(\alpha^+ ab \prec c \succ d^* \prec e \alpha^- \beta^+ f g^* \succ h \beta^-) = \{(d, e), (c, a), (c, d), (c, e), (g, h)\}
\]

### 7.2.2 Finite state machine

A deterministic finite state machine that corresponds to a path expression is built in two steps: first the path expression is translated into an equivalent (defining the same set of paths) regular expression, according to the mapping defined in the next figure, then the finite state machine that corresponds to the regular expression is constructed [2].
\[
\begin{align*}
\text{exp}_1 > \text{exp}_2 & \quad \Rightarrow \quad \text{exp}_1 \langle \text{exp}_2 \\
\text{exp}_1^* > \text{exp}_2 & \quad \Rightarrow \quad \text{exp}_1^* \langle \text{exp}_2 \\
\text{exp}_1^* \times \text{exp}_2 & \quad \Rightarrow \quad \text{exp}_1^* \langle \text{exp}_2 \\
\text{r}[\text{exp}_1; \text{exp}_2; \text{exp}_3; \text{exp}_4] & \quad \Rightarrow \quad \text{r}_\text{succ} \text{exp}_1 \text{r}_\text{body} \text{exp}_2 \text{r}_\text{target} \text{exp}_3 \text{r}_\text{interface} \text{exp}_4 \\
\text{sem}^+ \text{exp} \text{sem}^- & \quad \Rightarrow \quad (\text{sem}^+, r) \text{exp} (\text{sem}^-, r)
\end{align*}
\]

### 7.3 Rule evaluation

The evaluation of a rule is sequentially performed in four atomic steps: guard, body, interaction, and postcondition evaluation.

#### 7.3.1 Guard evaluation

The guard evaluation is defined by:

\[
eval : \text{Guard} \times \Sigma \rightarrow 2^{\Sigma \times S}
\]

The function \( \eval \) takes a rule guard (let call \( \text{In} \) the in-guard and \( \text{Rd} \) the read-guard), and the current object state \( \sigma \), and returns the set of pairs \((\sigma', \theta)\) satisfying:

- \( \forall A \in \text{Rd}. \exists B \in \sigma \text{ s.t. } A\theta \text{ is an instance of } B; \)
- \( \forall \text{getlist}(A, L) \in \text{Rd}. L\theta = \text{ list of atoms in } \sigma \text{ unifying with } A\theta; \)
- \( \forall \text{getlist}(A, L, C) \in \text{Rd}. L\theta = \text{ list of atoms } C \text{ in } \sigma \text{ such that } \exists \theta'. C\theta' = A\theta' \text{ and } \vdash G\theta'; \)
- \( \vdash G\theta, \) for every Prolog primitive \( G \in \text{Rd}; \)
- \( \sigma \supseteq \sigma' \text{ and } \sigma' = \text{ generalization of } \text{In}\theta \)
  \( \forall \)
  \( \{ B \text{ s.t. member}(B, L\theta) \text{ with } \)
  \( \text{getlist}(A, L) \in \text{In} \text{ and } A\theta \text{ unifies with } B \text{ or } \)
  \( \text{getlist}(A, L, C) \in \text{In}, \exists \theta'. B\theta' = A\theta' \text{ and } \vdash G\theta' \} \)
  \text{ with } \forall \text{ multiset union; } \sigma' \text{ is the multiset of atoms that must be removed from the store;}
- \( \text{no atom in } \sigma \text{ unifies with } A\theta \text{ for } \lnot A \in \text{Rd}; \)

19
\[ \textit{self}(O) \in Rd \Rightarrow O\theta \text{ is the name of the object where the rule is executed;} \]

\[ \textit{father}(O) \in Rd \Rightarrow O\theta \text{ is the name of the object that activated the one where the rule is executed;} \]

\[ \theta \text{ does not bind any variable appearing only in a negated atom;} \]

\[ \text{given a choice } C \text{ of atoms of the object state unifying with the guard, } \theta \text{ is the restriction, to the variables of the guard, of the most general unifier of } C \text{ and the guard itself.} \]

Note that because of the non-determinism in the selection of the atoms in the store, different choices of atoms can lead to different unifying substitutions.

If the outcome of \textit{eval} is the empty set, then the guard is not satisfied, while if it is satisfied, a pair \((\theta, \sigma')\) is selected and the guard is executed.

### 7.3.2 Body evaluation

Body evaluation is a standard Prolog evaluation of the atom in the body (instantiated with the substitution produced by the guard evaluation) with respect to the Prolog program associated with the object. Body evaluation is an outside computation without side effects on the tuple space and without modifying the knowledge base of the Prolog program (assert and retract primitives are not allowed). Possible outcomes of the body evaluation are either success (the first computed success substitution) or fail. Only in case of success the resulting substitution binds the variables of the postcondition.

### 7.3.3 Interaction evaluation

The interaction is evaluated only if the evaluation of the body is successful. Let \( \theta_b \) and \( \theta_h \) be the substitutions computed by guard and body evaluation, respectively, then interaction is instantiated by \( \theta_b \theta_h \) and then executed. Each interaction consists either of the communication to an external object or of the creation of a new object. The whole interaction is successfully executed if and only if it is possible to execute all its interactions.

A communication fails for a target failure (the target object is not reachable because it does not exist, the communication channel is interrupted, or the host is down) or for an interface failure (the target exists, but it does not
accept the tuples sent because of a violation of its interface). The creation of a new object fails if the required physical resources are not available (target failure) or if none of the classes in the library matches the request (interface failure).

7.3.4 Putset primitives

Special attention must be given to the primitives \( \textit{putset}(L) \), \( \textit{putset}(L, A) \), and \( \textit{putset}(L, A, G) \). The atoms written in the object state are: in the first case those contained in list \( L \); in the second case only those unifying with \( A \); in the third case the atoms in \( L \) both unifying with \( A \) and satisfying goal \( G \).

7.3.5 Postcondition evaluation

Postcondition evaluation depends upon the outcome of body and interaction evaluations. We distinguish four possible cases:

• Successful evaluation of body and interaction: the success part of the postcondition is instantiated by \( \theta_a \theta_b \) (outcomes of guard and body evaluation, respectively) and executed;

• Failure in the evaluation of the body: the body part of the postcondition is instantiated by \( \theta_a \) and executed (the interaction is not executed);

• Successful evaluation of body and target failure of the interaction: the target part of the postcondition is instantiated by \( \theta_a \theta_b \) and executed;

• Successful evaluation of body, no target failure, but an interface failure of the interaction: the interface part of the postcondition is instantiated by \( \theta_a \theta_b \) and executed.

8 Related works

We present and discuss some programming paradigms that integrate two out of the three models we merged. Then, we briefly present the Actors model, that derives from the Object Oriented model and that shows some similarities with the paradigm we propose.
8.1 Logic Programming & Object Orientation

Logic languages, and in particular Prolog, are not modular as they do not support reusability and information hiding. Moreover, they are not well suited to distributed problems, nor can they deal with an explicit notion of state. In models merging logic programming and object-orientation these limitations are partly overcome [6]. An object is a logical theory (a set of clauses) and a method is a clause. According to these choices, input and output method parameters are not statically defined, but dynamically determined using unification. More than one definition can be associated to each method and the correct method is singled out by using backtracking. Objects communicate by message passing and a message is the request to evaluate a goal. Inheritance relation is interpreted as combination of logic theories.

The merging of Logic Programming and Object Orientation well integrates declarativity and modularity. However, systems built in this model are not history-sensitive and are still unable to deal with a notion of system state.

8.2 Object Orientation & Multiple Tuple Spaces

In the object oriented model, an object communicates with other objects, while in the tuple space model an agent (thread) communicates with other agents through the tuple space. In the object oriented model an object has to react, with a method invocation, to the received messages, while in Multiple Tuple Spaces an agent can react, with a transition, to the received tuples (temporal decoupling).

An existing proposal of integration of these two paradigms defines a new communication model where tuple spaces are dynamically created objects that communicate by sending tuples [11]. However, there is no concrete example of a programming language inspired by this model.

8.3 Actors

The computational model that we present in this paper shares many characteristics with the Actors model. An actor is “a computational agent which has a mail address and a behavior. Actors communicate by message pass-
ing and carry out their actions concurrently” [1]. The communication is asynchronous, unless otherwise specified.

The main difference between the two paradigms relies on the receive policy. Actors have an associated buffer for incoming messages, which are processed one at a time. In Multiple Tuple Spaces, on the contrary, incoming messages are added to the state of the receiving object and are accessed in an associative way. This feature enhances the expressive power of the paradigm, as the reaction to a message can be contrained, in a uniform and declarative manner, to other conditions of the state, exploiting the unification algorithm and the atomicity of the guard evaluation.

9 Conclusions

We have identified the requirements of concurrency support, modularity, and declarativity for a language suited to describe cooperation and, more generally, to build large distributed systems. We have proposed the merging of Multiple Tuple Spaces, Object Orientation, and Logic Programming as a uniform paradigm that satisfy our requirements. Moreover, each paradigm satisfies one of the listed requirements and their merging does not affect this property. Our claim is that the resulting programming paradigm will be suited for building large distributed applications according to the principles of high level programming.

The language eta is a concrete example of language inspired to this paradigm. The implementation of a compiler and a distributed run-time support for eta is now the focus of our research.

Acknowledgements

We would like to thank Angela Discenza for her contribution to the definition of eta. We also thank the people of the Concurrent Systems Group at the Washington University, St. Louis, for their helpful suggestions.
References


