eta

Everything but Assignment *

Vincenzo Ambriola    Giovanni A. Cignoni    Laura Semini

Dipartimento di Informatica
Università di Pisa
corso Italia 40, 56100 Pisa

Abstract

We define eta, a new programming language that combines multiple tuple space, object oriented, and logic programming models. Multiple tuple spaces provide a powerful model for the development of large systems as a collection of communicating components. The combination of multiple tuple spaces with logic programming results in a paradigm that well integrates modularity and declarativity.

The object oriented model brings new insights to enhance tuple space languages without affecting their basic characteristics. In the language eta we integrate these three models: eta is a multiple tuple space logic language that borrows the programming style of the object oriented paradigm; supplies a reliable communication policy; enhances control allowing mutual exclusion and priority.

Keywords Concurrency, multiple tuple spaces, logic programming, object orientation.

1 Introduction

Tuple space concurrent languages exploit, for cooperation among processes, the shared memory paradigm. The store (i.e. the tuple space) is a multiset of tuples representing the cooperation state. Processes communicate by reading and writing on it.

The first system truly based on the notion of a tuple space, logically shared among concurrent agents, is Linda [9, 8]. Initially conceived as a coordination language, Linda evolved into a well defined paradigm to model communication and synchronization among distributed processes, written in various languages (C, Fortran, Prolog). Linda provides primitive operations to read, update, and change a tuple space. In Linda

*This work has been partially founded by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo del CNR, Sottoprogetto 6
the tuple space is the medium for cooperation between processes that have their own private control and a local store.

Coordination in Linda resembles the model of concurrent constraint programming, where processes communicate and synchronize through the store. Indeed, there are many differences between the two models. For instance, shared variables, that provide a second medium of communication in the concurrent constraint model, are not present in Linda.

In Shared Prolog (SP) a tuple is a logical fact [5]. The interaction between the tuple space (called blackboard) and the agents is controlled by the unification algorithm. The presence of some configurations of the tuple space causes the agents to react with a transition. A transition (or action, or step) is an operation composed of a guarded access to the tuple space, a computation, and a write operation on the tuple space. Agents have a local store and a local control only within each single step.

In a far-reaching extension of the tuple space paradigm, systems are composed of a set (or hierarchy) of tuple spaces, instead of a single one. Tuples are sent through the system from one tuple space to another, exploiting explicit addressing. This extension orthogonally combines the shared memory and the message passing paradigms, providing a model for the modular development of large systems as collection of components. A tuple space is the local store of each component and components interact exchanging tuples. Languages based on multiple tuple spaces are Linda-3 [12], ESP [7], PoliS [10], and Paté [4].

In this paper we present a new language, called eta, that combines the multiple tuple space model with object orientation and logic programming following the ideas presented in [2]. We identify a component, i.e. a tuple space and its associated agents, with an object. As in the SP paradigm, we exploit unification for the interactions between the tuple space and the agents.

The object oriented paradigm supports the notions of information hiding, inheritance, and inter-objects interaction policies. This merge brings new insights on the multiple tuple space framework.

In Section 2 we discuss some features of existing multiple tuple space languages and we show how they can be improved. In Section 3 we present the language eta showing how it copes with some general notions of concurrent and object oriented languages. In Section 4 we supply the syntax and an informal semantics of eta. In Section 5 we analyse some features of eta with respect to the general object oriented model and in the Appendix we provide an example of eta programming.

## 2 Motivations

For the general characteristics of the languages based on multiple tuple spaces we refer to the literature [7, 4, 10, 12]. Here we focus on some features: program structure, protection mechanism, priority, and mutual exclusion. We show how the existing languages deal with these features and we propose an alternative solution.
Program structure

In ESP and Paté (PoliS and Linda-3 features are substantially equivalent) a program consists of a library of theories, each theory being the parametric definition of an agent behaviour. At run time, many theories are activated on each tuple space, most of the times at tuple space creation. At this purpose, an activation goal specifies: the tuple space name; its initial state; a final condition that, when satisfied, causes the termination of the tuple space itself (both optional); the theory instances corresponding to the activated agents.

This style of programming gives a high degree of freedom and allows the reuse of theory definitions in different contexts. It has at least two disadvantages: activation goals are dramatically cumbersome; it is unnatural to design theories out of any context, as theories describe the communication of a tuple space with the others.

The experience of three years of programming in these languages, within the Oikos project at the University of Pisa [1, 3], has proven the fallacy of this approach. The reuse of parametric theory definitions is a rare event. Almost always, theories are designed having in mind the tuple space on which they are successively activated.

We contend that it is more natural to treat a tuple space and the set of its associated agents as a unique programming entity. This idea leads to the notion of object.

Protection mechanism

One of the common features of multiple tuple space languages is that an agent can send tuples to any tuple space provided that its address is known. Furthermore, in Linda-3 each agent can freely access the tuples of any tuple space both in read and consume mode. These language design choices stem from the attempt to maintain the philosophy of (single) tuple space based languages, where the tuple space is a communication medium that allows the autonomous behaviour of (uncoupled) agents.

As a consequence, an agent cannot check whether either data will eventually reach their destination or communication failed, e.g. because of an error in the address. Similarly, it is not possible to prevent a tuple from reaching a tuple space. When a tuple reaches a tuple space where it is unexpected, i.e. in which it is ignored by the agents, the tuple is not processed and the sender will never receive an answer or an acknowledgement.

The interactions between different tuple spaces are de facto inspired by the message passing paradigm, and communication policies, appropriate for this paradigm, must be supplied.

Priority

In SP, as well as in ESP, an agent is composed of a set of guarded rules. At each step, one of the rules whose guard is satisfied is nondeterministically chosen and the corresponding action is performed. This amounts to say that an agent selects the action to be performed according to the don’t care nondeterminism model of classical logic concurrent languages [14].

In Paté, control strategies such as sequencing and alternation of agent actions are declaratively expressed in terms of legal activation sequences, by path expressions [4].
However, the language is still missing a way of expressing a priority relation between rules.

**Mutual exclusion**

Agents operate on a tuple space concurrently accessing the same tuples. In some situations an agent need to lock a part of the tuple space. For instance, the agent may want to perform a sequence of actions on a “private space”, or to guarantee a mutual exclusion between two or more actions of different agents. Mutual exclusion is especially needed when the predicate *all* is involved in the guard of an agent (see Paragraph 4.6 and the use of *all* in the example supplied in the appendix).

### 3 The language eta

The language *eta* is designed to build (logically and physically) distributed systems, composed of a collection of concurrent communicating objects. Object states consist of a passive component, the tuple space, and of an active component, a set of *threads*. The former is a multiset of logical facts (atoms), while the latter corresponds to agents and resembles the agents of a Paté system.

We present the characteristics of *eta* with respect to a general classification criterion, showing how it copes with the notions of dynamicity, communication, information hiding, modularity, and others.

#### 3.1 Dynamicity

Dynamic languages allow the design of open systems, where it is possible to add to a system an entirely new component that was not even thought at system start up. This incremental growth should not compromise the behaviour of the system itself. In an open system, a component is added without halting the running system, compiling, and activating the new version.

In *eta*, where systems are composed of collections of objects, we achieve dynamic features with the definition and the activation of new objects at run time.

#### 3.2 Communication

In *eta*, objects communicate by sending and receiving messages, using object names as addresses. The language is asynchronous in the sense that, after a send, objects do not stop waiting for an answer.

Messages are atoms that are added to the receiver object’s state and not operation or procedure invocations like in most of the object oriented languages. This allows to model, at semantic level, the interactions between two objects as a *rendez-vous*: after sending a message, an object is blocked until either the message has reached 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, combined
with path expressions (see Paragraph 3.7), it forces the sender of a message to wait for an answer to its request. It is hence possible to program totally synchronous communications, yielding to a (potentially) synchronous language.

3.3 Information Hiding

An object provides an interface to the outside world in terms of the messages that both accepts as input by other objects and produces (and sends through the system). Any message sent to an object that does not consider it a possible input, is treated as an error, and the sender is acknowledged. In this way we prevent unexpected tuples from reaching a tuple space, as wanted. Moreover, as the interface is the only visible public part of an object, we can freely modify object behaviour, just keeping invariant the interface itself.

3.4 Modularity

A program in \textit{eta} is a library of class definitions, where a class is, as usual, a parametric definition of objects. At run time, new objects are activated simply instantiating the parameters of a class. This one-to-one relationship between system modules - the physical objects composing a system- and programming modules -class definition instances- lifts the modular structure of the multiple tuple spaces paradigm at the programming level, simplifying the programmers work.

Moreover, modularity provides, at system level, a framework to develop large systems exploiting the client-server architecture, and modularity at programming level, coupled with inheritance, naturally supports reuse.

3.5 Reactiveness

Threads connected to an object react to given state configurations by reading and consuming some of the contained atoms. Then they carry out an internal computation and produce an output.

Threads behaviour is described by rules: rules have a name and are composed of a guard, a body, and a postcondition.

\[
name: \quad \text{guard} \quad \text{body} \quad \text{postcondition}
\]

The rule name is a term that uniquely identifies the rule within the object. The guard is a sequence of read (\textit{Read-guard}) and input conditions (\textit{In-guard}): the former is a compound goal on the object state, the latter defines the set of atoms to be removed from the state. A guard is satisfied when two conditions are satisfied: there exists a substitution that unifies a subset of the tuples contained in the object state with the rule guard; the atoms specified by the \textit{In-guard} occur in the object state and can be removed. The \textit{body} is an atomic Prolog goal, evaluated with respect to the Prolog program associated to the object (see Section 4). The \textit{postcondition} is a list
of atoms to be written in the object state, or activation requests, or atoms to be sent to other objects. The commit operator ‘\|’ separates the precondition from the body. The commit operator has a semantics similar to that of classical logic concurrent languages. If the rule is fired, i.e. the commit is taken, then no backtracking is possible. This is often called don’t care nondeterminism [14].

3.6 Declarativity

In \texttt{eta} the interaction between the state and the threads of an object is controlled by the unification algorithm. Unification enhances the expressive power of the language as complex conditions on the tuple space can be expressed in the rules in a declarative and concise manner.

Variables scope plays an important role in the language: variables appearing as object parameters bind the variables occurring in the rules and in the object interface. Viceversa, the scope of any variable appearing in a rule, and not in the class parameters, does not extend out of the rule itself. Finally, bindings in the Prolog program are limited to the single clauses, as usual.

3.7 Control

Path expressions have been introduced in Paté as a mechanism to constrain a wild non-determinism in rules activations [4]. In \texttt{eta} they are extended to specify critical regions as well as sequencing, alternation, and rule priority.

A path expression is similar to a regular expression. It specifies a set of paths over the alphabet of rule names extended with \( \epsilon \), the empty path expression and \( \{ \gamma_1, \ldots, \gamma_n, \ldots \}^{\{+,-,\}} \), special terms used to mark critical regions. According to this form of control, a rule can fire when both its firing satisfies the path expression and its guard is satisfied by the current state.

3.8 Parallelism

The language \texttt{eta} provides two grains of parallelism: fine grained parallelism among the threads of an object and coarse grained parallelism among the objects of an \texttt{eta} system. Each form of parallelism refers to a precise cooperation model: shared memory and message passing.

These \texttt{eta} features, both with respect to parallelism grain and to cooperation models, look like multitasking implementations commonly used in last generation operating systems, as lightweight and heavyweight processes in Mach [15]. In \texttt{eta} these features are independent from any language implementation: they are formalized by language constructs rather than achieved by system calls.

3.9 Reliability

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

At the language level, it is possible to supply, in rule postconditions, default atoms
to be written in the object state, in case of failure of the body evaluation; failure of
the communication; failure in the activation of a new object.

### 3.10 Inheritance

In *eta* a class definition can inherit from another one, exploiting an *isa* declaration.
We adopt an extensional mode of inheriting 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 the messages that belong to a
superset of those that are sent and received by an object of the superclass, while the
reactions to an identical message can be different.

### 4 Syntax and Semantics

In *eta* a program is a collection of object class definitions, where a class definition has
the following structure:

```plaintext
Class  class_name  [isa class_name]
input  \{atom\}*
output \{free atom\}*
contents \{free atom\}*
initial \{atom\}*, \{thread\}*
critical \{region_name : \{atom\}\}*
thread \{thread_name : path_expression\}*

rules
\begin{align*}
  &rule_1 \\
  \vdots \\
  &rule_n
\end{align*}

with

prolog_program

end
```

In case of class definition by inheritance, all these parts are optional, while at least
*thread*, *rules*, and one between *input*, *output*, and *contents* must be specified when
defining an entirely new class.

### 4.1 Class_name

A class name is a term. The functor identifies the class, while the arguments are
free variables that are instantiated when an object is activated. Variables appearing
as parameters in a class name can bind variables appearing in all the parts but the
Prolog program.
4.2 Input, output, contents

These parts specify the atoms that an object of the class can accept as input from other objects, contain during the execution, or send to other objects, respectively. Input, output, and contents of an object can be statically derived from rule definitions. However, we contend that their explicit declaration is a readable interface of the object itself. In contents we allow only free atoms, as we want to statically verify if they meet rule postconditions.

4.3 Initial

Initial specifies the sequence of atoms that represent the object initial state and the sequence of the initial threads, i.e. those activated with the object.

4.4 Critical

A critical region is identified by a name that ranges over the alphabet \( \{\gamma_1, \ldots, \gamma_n, \ldots\} \) and specifies a set of atoms. Atoms specified in a critical region cannot be consumed by other threads than the one who entered the critical region itself.

4.5 Thread

A thread consists of a name and a path expression. The thread path expression specifies critical regions, sequencing, alternation, and rule priority, according to the syntax given below. We use three syntactic categories: \( \text{Exp} \), \( \text{Rule} \), and \( \gamma_i^{\{+,-\}} \), where \( \text{Rule} \) denotes a rule name, \( \text{Exp} \) a path expression, and \( \gamma_i \) ranges over the alphabet of the critical region names.

\[
\text{Exp::} \quad = \quad \epsilon \quad \quad \text{empty} \\
\quad | \quad \text{Rule} \quad \quad \text{rule name} \\
\quad | \quad \text{Exp} \text{Exp} \quad \text{sequencing} \\
\quad | \quad \text{Exp}\mid \text{Exp} \quad \text{choice} \\
\quad | \quad \text{Exp} \triangleright \text{Exp} \quad \text{priority} \\
\quad | \quad \text{Exp}^* \quad \text{iteration} \\
\quad | \quad [\text{Exp}] \quad \text{optionality} \\
\quad | \quad (\text{Exp}) \quad \text{precedence} \\
\quad | \quad \text{Rule}\{\text{Exp}; \text{Exp}; \text{Exp}; \text{Exp}\} \quad \text{failure} \\
\quad | \quad \gamma_i^+ \text{Exp} \gamma_i- \quad \text{critical region}
\]

The grammar generates the set of path expressions over the alphabet of rule names extended with \( \{\gamma_1, \ldots, \gamma_n, \ldots\}^{\{+,-\}} \) and \( \epsilon \), the empty path expression. A path expression is similar to a regular expression, with the following exceptions: the priority operator modifies choice allowing to express a priority between two alternative paths; the optionality construct is a shorthand for \( \text{Exp} | \epsilon \); the failure operator takes into account the possibly different outcomes of a rule evaluation (success or failure of the body and of the communication).
Every rule name must appear at least in one of the threads of the object, and can appear more than once in each thread.

A thread terminates when it reaches the end of one of the (finite) paths described by its path expression. When all the threads of an object are terminated, the object itself terminates.

4.6 Rules

Rule syntax is indeed more complex than the one we have given in Paragraph 3.5, where we skipped many details, such as error handling and primitives. A rule takes the form:

\[
\text{name} : \ \text{read.guard \{in.guard\}} \\
| \text{body} \\
| \text{out} \\
; \ \text{body} \ \text{failbody} \\
\text{data} \ \text{faildata} \\
\text{target} \ \text{failtarget}
\]

Rule guard

Both read.guard and in.guard are sequences of atoms. Read.guard may contain primitives like =, ?=, \, ==, =, >, is, var, ... derived from Prolog, to express relations between the different facts composing a guard, and primitives of eta:

- self(O), father(O). Variable O is bound to the object name or to the name of the object creator, respectively.
- not A succeeds iff A does not unify with any atom of the state. In case of success, not A does not produce any substitution.
- all(A, L), all(A, L, G). In the first case L is bound to the list of all the atoms of the object state that unify with A, while in the second case atoms in L must also satisfy goal G.

The last two primitives can appear in an In-guard too. In this case the selected atoms are removed from the state. For guard evaluation we refer to [11].

Some syntactic sugar: in the precondition we use curly brackets to denote tuples removed (In.guard), in the postcondition the same brackets denote tuples to be written.

Rule body

Rule body is an atomic Prolog goal. It is instantiated by the substitution computed by guard evaluation and then evaluated with respect to the Prolog program accordingly to Prolog usual semantics. In case of success the first computed answer substitution binds the variables of the postcondition, otherwise no substitution is computed, the postcondition is skipped, and failbody is written in the object state.
Rule postcondition
Rule postcondition consists of an \textit{out} followed by the (optional) parts \textit{body}, \textit{data}, \textit{target} each of them containing an atom. The \textit{out} is a sequence of:

- atoms to be written on the object state;
- expressions of the kind \texttt{atom@target} whose meaning is that the atom must be sent to object target;
- thread activations;
- expressions of the kind \texttt{new}(O) whose meaning is that a new object \(O\) must be activated;
- expressions of the kind \texttt{list}([L]) or \texttt{list}([L]@target) whose meaning is that the atoms contained in \texttt{list} \(L\) must be written in the object state or sent to object target, respectively.

The parts \textit{body}, \textit{data}, \textit{target} specify the atoms to be written in the object state in case of failure of the body evaluation (\textit{body}); in case of a communication failure, caused by an interface violation (\textit{data}) or by the non existence of the target object (\textit{target}); in case of failure of an object activation because of the non existence of a corresponding class (\textit{target}).

4.7 Isa
When a class inherits from another one, say \texttt{A\textunderscore isa}\texttt{B}, the parts \textit{input}, \textit{output}, \textit{contents} and \textit{initial} of the two classes are disjoint, with the meaning that in \textit{A} we simply extend \textit{B}. On the contrary, we allow overriding of rules, threads, critical regions, and Prolog clauses (predicate definitions), i.e. we can redefine them in \textit{A}.

5 Discussion
Object orientation usually implies a structure over program data. In logic languages, object orientation leads to a structured knowledge base [6, 13]. An \texttt{eta} object has a static knowledge base, which defines the semantics of the rule bodies. The tuple space contents, which is dynamic and open to external inputs, affects, with rule definitions, the reactive behaviour of an object. Both these aspects belong to the definition of an object and exploit the advantages of the object oriented model, in particular of the inheritance.

The way to access an object in \texttt{eta} is, as usual, by sending messages through a rigid interface. Messages are atoms that are added to the receiver object’s state, that is shared among concurrent threads. As a consequence, we miss the usual and full correspondence between the object interface, i.e. the set of exported methods, and its functional behaviour. However, from a programming point of view it is straightforward to define objects with this feature and, from a semantic point of view, it is not difficult to explicit this correspondence in the general case.


6 Conclusions and Future Works

The \textit{eta} implementation is designed to exploit a network of Unix workstations. This choice has been motivated by the great diffusion and the acceptable price of distributed environments of this kind.

The realization of the \textit{eta} run time support will benefit from the experience gained during the implementation of Paté and is more than an affordable task. The approach is to compile \textit{eta} objects in programs written in a object oriented language (actually C++). Each \textit{eta} object runs as a multithreaded Unix process. Evaluation of Prolog goals, interprocess communication, and multithreading are supported by custom predefined libraries.

As extensions of the language, we plan to introduce multiple inheritance and a notion of \textit{eta} system that allows to statically define and check cooperation properties among objects.

Acknowledgements

We gratefully thank Roberto Deias and Angela Discenza for various helpful discussions.

References


Appendix

The example describes a zoo.

Animals (bears) growl when hungry, wait some time (PatienceTime), and then either some food has arrived or they growl once more. After eating, if the food was enough with respect to their diet (to.eat(0)), they yawn and sleep, otherwise, they growl to have some more food.

Wardens take care of animals, preparing food when they growl. The time they spend in this operation depends on their Laziness and reduces if a warning (hurry_up) is received.

The zoo management controls the work of the wardens, engages new wardens if they are not enough, and buys new animals.
Class \texttt{bear}(Name, Diet, SleepTime, PatienceTime, ZooMan, Warden)

input \{food(honey, Quantity)\}

output \{growl(Name),
yawn(Name),
animal(Type, Name, Id)\}

contents \{to\_eat(Amount)\}

initial \{to\_eat(Diet)\},

\{startup\}

thread \texttt{startup}: \texttt{init}

\texttt{behaviour}: \((\texttt{eat} > \texttt{growl})^* \texttt{sleep} \texttt{awake})^*

rules

init : \texttt{self(Id)}

| \{animal(bear, Name, Id)\} @ ZooMan
\{animal(bear, Name, Id)\} @ Warden
\{behaviour\}

eat : \{food(\_, Quantity), to\_eat(Amount)\}
\texttt{eat}(Amount, Quantity, NewAmount).
\{to\_eat(NewAmount)\}

growl : \texttt{to\_eat}(Amount, Amount > 0)
\texttt{wait}(PatienceTime).
\{growl(Name)\} @ ZooMan
\{growl(Name)\} @ Warden

sleep : \{to\_eat(0)\}

| \{yawn(Name)\} @ ZooMan

awake : | \{sleep(SleepTime),
\{to\_eat(Diet)\}

with \texttt{eat}(Amount, Quantity, NewAmount) : –

Amount > Quantity,
NewAmount is Amount – Quantity,
suspend(Quantity).

\texttt{eat}(Amount, Quantity, 0) : –

Amount < Quantity,
suspend(Quantity).

\texttt{wait}(PatienceTime) : –
suspend(PatienceTime).

\texttt{sleep}(SleepTime) : –
suspend(SleepTime).

end
Class normalbear(Name, Diet, SleepTime, PatienceTime, ZooMan) isa bear
input {food(meat, Quantity), food(vegetables, Quantity)}
end

Class toonbear(Name, Diet, SleepTime, PatienceTime, ZooMan) isa bear
input {food(sandwich, Quantity), food(cake, Quantity)}
end

Class warden(Name, Rest, Laziness, ZooMan)
input {animal(Name, Id), growl(Name, Id), hurry-up}
output {food(Food, Quantity), warden(Name, Id, 0)}
contents {food(Food, Quantity), dislike(Name, Food), lazy(CLaziness)}
initial {lazy(Laziness), food(banana, 6), food(honey, 4), food(meat, 8), food(sandwich, 6), food(vegetables, 6), food(cake, 8)},
{startup}
thread startup : init
  task : (work > hurry-up > rest)*
rules
  init : self(Id)
     | {warden(Name, Id, 0)}@ZooMan
   | {task}
  work : {growl(Name)}
     | animal(Name, Id), food(Food, Quantity),
     | not dislike(Name, Food),
     | lazy(CLaziness),
     | prepare(Food, Quantity, CLaziness).
     | food(Food, Quantity)@Id
     |
    data {dislike(Name, Food)}
  hurry-up : {hurry-up, lazy(CLaziness)}
     | {speed-up(CLaziness, NLaziness).}
     | {lazy(NLaziness)}
     |
    body {lazy(CLaziness)}
  rest : suspend(Rest).
with prepare(_, Quantity, Laziness) :-
    Time is Quantity × Laziness, suspend(Time).
speed-up(CLaziness, NLaziness) :-
    CLaziness > 1, NLaziness is CLaziness - 1.
Class zoo-management(MaxRate, MaxAnimal)

input
{warden(WName, WId, 0), animal(Type, AName, Id),
growl(AName), yawn(AName)}

output {hurry_up}

contents {take care(WName, AName),
hungry(AName, HRate),
warden(WName, WId, ANum)}

initial {}
{control, manage}

critical γ : {warden(–, –)}

thread control : (watch > (γ⁺, create animal γ⁻))¹
manage : (growl > yawn > (γ⁺ create warden γ⁻))¹

rules

watch :
{hungry(AName, HRate)} HRate > MaxRate,
take care(WName, AName),
{warden(WName, WId, –)}

{hurry_up}@WId
{hungry(AName, 0)}

create animal :
{warden(WName, WId, N)} N < MaxAnimal,
self[ZooMan]
{buy animal(ZooMan, WId, Animal, N, M, AName),
warden(WName, WId, M),
take care(WName, AName)}
new(Animal)

growl :
{growl(AName), hungry(AName, HRate)}
New HRate is HRate + 1.
{hungry(AName, New HRate)}

yawn :
{yawn(AName), hungry(AName, –)}

{hungry(AName, 0)}

create warden :
self[ZooMan]
all[warden(–, N), WList, N < MaxAnimal]
WList = [ ]
engage warden(ZooMan, Warden).
new(Warden)

with engage warden(ZooMan, warden(Name, Rest, Laziness, ZooMan)) :-
gen unique name(Name),
random(Rest),
random(Laziness),
buy animal(ZooMan, WId,
normal bear(AName, Diet, SleepTime, PatienceTime, ZooMan, WId),
N, M, AName) :-
gen unique name(AName),
random(Diet),
random(SleepTime),
random(PatienceTime),
M is N + 1.

end