Using Tuple Space Communication in Distributed Object-Oriented Languages

(In Proceedings of ACM OOPSLA'88, Special Issue of SIGPLAN Notices, Vol. 23-11, 1988 pp.276–284)

Satoshi MATSUOKA

matsu@is.s.u-tokyo.ac.jp

*Satoru KAWAI**

(*Currently with Department of Computer and Graphic Sciences, College of Arts and Sciences, The University of Tokyo)

kawai@graco.c.u-tokyo.ac.jp

Department of Information Science,

Faculty of Science, The University of Tokyo

Address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Telephone: +81-3-5800-6913

Abstract

When Object-Oriented languages are applied to distributed problem solving, the form of communication restricted to direct message sending is not flexible enough to naturally express complex interactions among the objects. We transformed the *Tuple Space Communication Model*[29] for better affinity with Object-Oriented computation, and integrated it as an alternative method of communication among the distributed objects. To avoid the danger of potential bottleneck, we formulated an algorithm that makes concurrent pattern matching activities within the Tuple Space possible.

1. INTRODUCTION

1.1 Object-Orientation, Distributed Systems and Distributed Problem Solving

The goal of computer systems is to solve problems. Complicated problems are decomposed into sub-problems. Decomposition may be from different viewpoints, such as procedural, functional, or *Object-Oriented.* At the same time, systems have become *distributed* due to lower CPU cost and faster communication speed. Linguistic supports for programming in the distributed systems are provided by *concurrent/distributed languages*. In the field of *distributed problem solving*, problem solving has become distributed among multiple knowledge bases.

In solving problems, one of the most attractive attributes of Object-Oriented languages is to allow natural modelling of physical entities in the user's problem domain. Each object can represent actual physical entity of the real world, such as a car or a human being, or more abstract concept, such as time or date. The behavior of an object is the definition on how it will react to external events. It is the simplified subset of the behavior of its counterpart in the real world. Concurrent interaction of the objects with each other is the metaphor of collective behavior of solver-entities in actual problem solving.

In the study of distributed systems, one objective is to provide powerful linguistic support for concurrent programming. Various distributed languages have been proposed as a result; they include: ADA [1], Cell [2], Concurrent Pascal [3], CSP [4], Distributed Processes [5], Euclid [6], PLITS [7], and Synchronizing Resources [8]. Since Object-Oriented computational model is based on message passing, we will concentrate on discussing main design issues of communication primitives [9] in those languages.

One issue is the *synchronization* of messages, in other words, whether message sends are *synchronous* (CSP, ADA), *asynchronous* (PLITS), or both (Synchronizing Resources).

Another issue is the *form* of communication. This addresses the the relationship between the senders and the receivers in the communication. They include:

- *symmetry* of communication — symmetric or asymmetric (master-slave)
- *correspondence* of communication — 1 to 1, 1 to *n*, *n* to 1, or *n* to *n*.
- *scope* of communication — to whom the message can be sent.

The issue that closely affects the correspondence and the scope of communication is the *naming scheme* or *addressing* of processes. [10] categorizes four types of possible naming schemes:

- *Implicit addressing* allows a process to communicate with a single parent process.
- *Explicit addressing* w makes the process name its target explicitly. It requires global knowledge of a name source containing the identites of all the processes in the system.
- *Global addressing* associates global names with local mailboxes.
- *Functional addressing* establishes connections based on the need to serve or request of service. The service *path* such as a *port* identify the process at the other end.

Here, we define *direct message send* as the form of communication whose correspondence is 1 to 1, whose scope is global, and whose naming scope is explicit.

1.2 Concurrent/Distributed Object-Oriented Languages

In a distributed Object-Oriented environment, the objects may reside within different addressing spaces with no means of sharing, and communicate with each other by sending messages. In order to provide linguistic support for such an environment, there have been many proposals of concurrent/distributed Object-Oriented languages in the recent years; to date, they include: Act 1 [11], Act 3 [12], ABCL [13], CLIX [14], Concurrent Smalltalk [15, 16], Emerald [17], Orient84/K [18], POOL-T [19], and Vulcan [20]. Their common properties are to model the objects as concurrent entities which actively exchange messages with one another, namely *actors* [21] or *concurrent objects* [18]. [19] states that concurrency in Object-Oriented languages can be seen from the standpoint of conventional parallel programming, as characterized by [9]. [22] compares Object-Oriented languages against conventional languages for distributed systems. [23] focuses on the deadlocking problem that occurs when delegation is used as the mechanism of knowledge sharing in concurrent Object-Oriented languages. [24] investigates the difference between inheritance and subtyping, and the potential problems when incorporating them into a distributed Object-Oriented language. [25] discusses several design choices when class information is distributed in Smalltalk-80 [26].

1.3 Problems in Current Concurrent/Distributed Object-Oriented Languages

We believe that one of the essential characteristic of Object-Oriented systems is to provide means for the user to define objects so that he can implement the model of his particular problem naturally on the system. Most Object-Oriented languages are robust from this viewpoint: by using inheritance, entities can be grouped according to similarity in their properties, and only their peculiarities need to be distinguished with subclass relations. As a result, inheritance has been studied extensively, theory and practice.

However, most current distributed Object-Oriented languages assume that the interactions among the objects are done solely with direct message sends, despite many studies of distributed languages which state that explicit naming of processes is restrictive [9, 10], and proceeds to investigate more powerful and effective communication primitives.

In distributed problem solving, the interactions among the cooperative entities can vary in a number of complex ways. For example, the contract net protocol [27] requires broadcasting of the announcement of eligibility requirements to a set of potential contractors by the manager. If only direct message sends were available, the manager, in turn, must send a message to each contractor he has to his acquaintance (Figure 1.1). As a consequence, the manager must have explicit knowledge of all the contractors that can take part in the bidding. When a sub-contract is announced by the contractor, it must have a knowledge of potential sub-contractors as well, and so on. This spoils the initial intention of the contract net protocol, in that one of the favorable characteristics of the protocol was that automatic distribution of the problem is achieved through content-directed procedure invocation.

The disadvantage further manifests itself when a new solver is added; then a painstaking process of adding the solver to the acquaintance set of each manager and solver must be performed (Figure 1.2). This calls for enumeration of all managers and solvers in the system, requiring maintenance of some global set holding such information. Such maintenance usually requires explicit programming, introducing various unnecessary details into the program. The result is that we have drifted far away from the original intent of Object-Orientation, where 'natural programming of problem at hand is possible.'

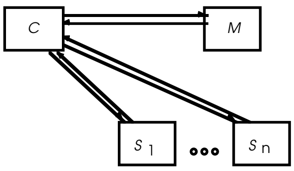

Figure 1.1

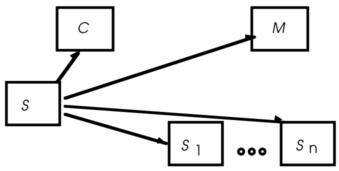

Figure 1.2

We claim that the root of some of the problems encountered when attempting to distribute current Object-Oriented systems [25, 28] lies in the restrictiveness of communication. Direct message sending may be appropriate for direct communication to tightly coupled objects, but lacks robustness in many respects. Extending the synchronization of communication alone is only a partial solution. We need a new model of communication that has high affinity with the Object-Orientation, while powerful enough to resolve all the problems we have presented.

2. THE TUPLE SPACE COMMUNICATION MODEL

2.1 Basic Concepts of Tuple Space Communication Model

Tuple Space Communication, alternatively *Generative Communication*, was first proposed by D. Gelernter as a fundamental communication model for *Linda* [29], a language for distributed systems. Individual processes communicate with each other via a medium called the *Tuple Space*. The list of formal and actual arguments given in the send request forms a *Tuple*. The sender process inserts the Tuple into the Tuple Space. Each Tuple is a unique, independent existence in the Tuple Space. The receiver process gives its own list of arguments in its Tuple withdrawal request. Withdrawal occurs when there is a Tuple *matching* the receiver's specific request; otherwise, the receiver process waits until such a Tuple becomes available in the Tuple Space. The receiver obtains the necessary information from elements of the Tuple where the formal argument of the receiver matched the actual element of the Tuple

Gelernter defines three basic operations of the Tuple Space communication. The operation out(P_1, ... , P_j) outputs the Tuple consisting of elements P_1, ... , P_j into the Tuple Space. The operation in(P_1, ... , P_j) withdraws a Tuple matching P_1, ... , P_j from the Tuple Space. The operation read(P_1, ... , P_j) reads the content of the Tuple matching P_1, ... , P_j but does not extract it from the Tuple Space. A single Tuple may match several receivers waiting with the in() or the read() operations, and it is logically non-determined as to which one will match and extract the Tuple. Alternatively, multiple Tuples may match a single in() or a read() by a receiver; in this case, only a single Tuple may be extracted. All the operations are <u>atomic</u>, i.e., a single Tuple may not partially instantiate a receiver's request, but rather, can only be extracted as a whole.

The first necessary condition for a Tuple to match the receiver operation is that its *arity*, i.e., the number of elements, matches that of the receiver request. Next, individual elements of the Tuple must match the corresponding arguments of the receiver operation. There are four possible combinations of formal and actual arguments. Below, the LHS of the expression denotes the argument of the out() operation (which becomes the element of the Tuple), and the RHS denotes the argument of the in() or read() operation.

- <u>formal × formal</u> — Formals never match formals. No value passing occurs. (Alternatively, a match could be defined if the their types match.)
- <u>formal × actual</u> — A match occurs if the types of both arguments match. No value passing occurs (*inverse structured naming*).
- <u>actual × formal</u> — A match occurs if the types of both arguments match. The formal argument of the receiver is instantiated with the value of the sender argument.
- <u>actual × actual</u> — A match occurs if the types <u>and</u> the values of both arguments match. Global identifiers can be given by both sides to specifically select a Tuple in the communication.

Gelernter identifies several distinguishable properties of the Tuple Space communication. *Communication orthogonality* means that the sender and receiver are equivalent in their functionality, and that both the sender and the receiver need no direct prior knowledge of each other. This is contrary to message sends in Object-Oriented languages, where the receiver needs to be a member of the sender's acquaintance set, and the receiver must be explicitly named in the message send statements. *Space uncoupling*, *time uncoupling*, and *distributed sharing* are derived directly from this property. *Free naming* means that the elements of a Tuple can be used to name it with a structured identifier, whereas other programming languages impose a single naming scheme for identifying the receiver. This allows the communication operation to have support for *continuation passing* and *structured naming*.

2.2 Transformation of the Tuple Space Communication Model in Distributed Object-Oriented Computing

We claim that the problems in distributed Object-Oriented languages can be resolved with Tuple Space communication integrated as a fundamental communication functionality among the objects. However, the original model of Tuple Space communication is deficient in several respects if it is simple-mindedly blended into Object-Oriented languages. Fundamental transformation of the semantics of the communication model must be made in order to correct the deficiencies and extend its power. The transformed model merged with Object-Oriented model of computation forms a new model, whose power is sufficient to serve as a solutions to the problems presented. For each transformation, we show the deficiencies present in the original model and how they are corrected.

Transformation 1: *Object-Orientation of Tuples and Tuple Spaces*

In the actor model, messages themselves are considered to be first-class objects. This is necessary in facilitating message delegation of opaque objects. In many Object-Oriented languages, however, messages are usually not regarded as first-class objects. In Smalltalk-80, for example, the message selector and message arguments are objects, whereas the message itself is not an object[1]. One cannot, for instance, send a message to a message.

By contrast, in Tuple Space communication, objects communicate using Tuples; as a result, it is natural to regard Tuples as objects. The functionality of Tuples can then be defined in a manner consistent with other objects in the system. The advantages are:

- An object can retain the message even though it may not have the ability to understand it: Later on, by gaining the ability, the message can be processed.
- An object may delegate the message freely to other objects, regardless of its ability to understand it.
- A Tuple is subject to garbage collection.

The first property is essential for maintaining time-uncoupling property when the actions of the objects can be altered dynamically, such as the *enhancement* mechanism in [30]. The second property allows transparent delegation, a non-elegant task in some Object-Oriented languages. The third property implies that a Tuple used in one communication may be re-used in another communication.

Transformation 2: *Communication Orthogonality between Sender and Receiver Tuples*

In the original model, the sender creates a Tuple by giving the *specification* of the Tuple in the out() operation (Figure 2.1.A). The receiver, in turn, gives a specification of the Tuple it wants to match in the in() or the read() operation, then extracts the matching Tuple (Figure 2.1.C). In this model, the creation of the Tuple is only done implicitly by the sender. Here, violation of communication orthogonality is apparent:

---

1 In Smalltalk-80 , class Message exists for the use by a simulator that is invoked by the debugger.

the sender may pass the Tuple to other senders who have no knowledge of the entire specification of the Tuple, and they may use the Tuple with no modification or only partial modification of the elements they do know about. However, on the receiver side, the entire receiver specification of the Tuple must be revealed and shared among the receivers[1].

In order to overcome the deficiency, we modify the model so that both the sender and the receiver output Tuples into the Tuple Space (Figure 2.2.A). We distinguish them by referring to Tuples created by the senders *sender Tuples*, and those created by the receivers *receiver Tuples*. The sender and receiver Tuples mutually check each other for a match. If a match occurs, unification of the Tuples occur by the formal elements of the receiver Tuple being instantiated by the corresponding actual elements of the sender Tuple (Figure 2.2.B). Then, the unified Tuple is returned to the receiver (Figure 2.2.C). Orthogonality is maintained with this model, as both the sender and the receiver Tuples may be shared among the clients; still, the flow of information is still one way. Also the distinction of the Tuples is made only by the roles they play in the communication; a Tuple serving as a receiver Tuple may serve as a sender Tuple in another situation, or vice-versa.

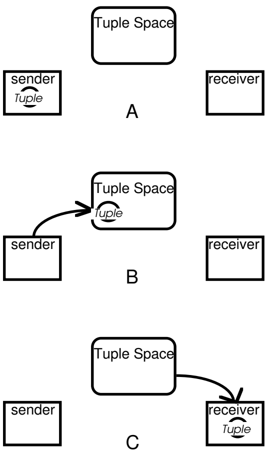

Figure 2.1

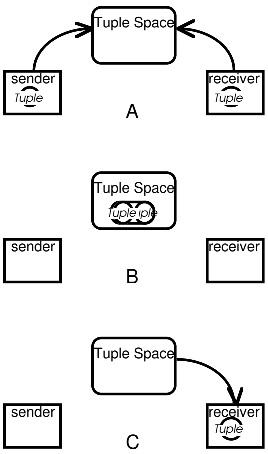

Figure 2.2

We illustrate how the transformed model is advantageous over the original for a certain case in distributed problem solving. There is a client object *C* with a set of problems to be solved. *C* communicates with problem solver manager *M* via Tuple Space communication, requesting to solve some of the problems. *M* has a set of problem solvers $S_1, ..., S_n$ to his acquaintance. When *M* receives the entire problem from *C*, it breaks the problem down into sub-problems, and assigns them to $S_1, ..., S_n$.

During the course of problem solving, *C* and *M* communicate, exchanging partial results. There may be situations where S_1, ..., S_n directly participate in the communications in order to receive broadcasted information from *C*. But if Tuples cannot be passed as objects, S_1, ..., S_n cannot directly wait for messages from *C*, as they cannot predetermine the specification of the Tuples used in the communication between *C* and *M*. Instead, they must rely on *M* to translate and broadcast to them (Figure 2.3.A). This restricts the structure of distributed control to be hierarchical, which is not necessarily always the most efficient [31].

By allowing the receiver Tuples to be passed as objects, *M* can broadcast to the solvers beforehand the partial specifications of the Tuples it intends to use in the communications with *C*. From then on, the solvers can directly receive messages intended for *M* from *C* , without having total knowledge of the Tuples used in the communications (Figure 2.3.B)[2] .

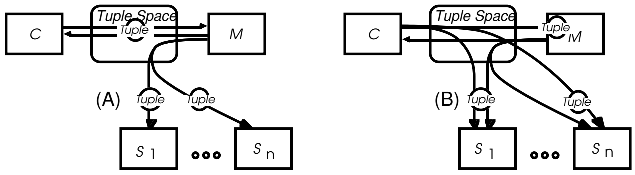

Figure 2.3 Distributed Problem Solving: Schematics of the original communication model (**A**) versus the proposed model (**B**) applied to distributed problem solving. Notice how S_1, ..., S_n can directly receive messages intended for *M* in (B).

Transformation 3: *Distribution of Tuple Space*

Gelernter's original model assumed that the Tuple Space is a single, unique global meta-entity in the system. By contrast, we assume that Tuple Spaces are objects, and are multiply distributed throughout the entire system. Two objects can communicate via a certain Tuple Space if and only if the Tuple Space is an acquaintance to both objects. This solves several problems inherent in Gelernter's model:

- Inefficiency — If there are huge number of objects exchanging enormous number of Tuples via a single global Tuple Space, the Tuple Space is likely to become a bottleneck. By distributing the Tuple Space, the flow of Tuples can be distributed evenly within the system, avoiding bottlenecks to some extent.
- Lack of effective naming scope — System-wide naming convention must be employed for identifiers that are elements of Tuples when different groups of non co-operating objects communicate simultaneously via a single Tuple Space. Such naming convention is not only extraneous (and is thus undesirable) to the specification of programming languages, but also must be hard-coded into the program. By having multiple distinct Tuple Spaces assigned to each group, such convention becomes unnecessary.
- Lack of security — If the Tuple Space is global, a malicious object can easily insert a Tuple with the wrong information, or extract a Tuple crucial to the computation. Distributing multiple Tuple Space overcomes this problem easily by secluding the acquaintance of the Tuple Space within the group.

By treating the Tuple Space as objects, we additionally obtain the following characteristics:

- Tuple Spaces can be created dynamically when necessary.
- Tuple Spaces can become first-class objects.
- Tuple Spaces can be subject to garbage collection.

---

[1] Such sharing is impossible in Linda, because the specification of the Tuples must be hard coded in the program.

[2] Of course, various minor conventions must be established.

- The functionality of the Tuple Space can be extended using inheritance.

The ability to pass a Tuple Space as an element of a Tuple is an essential characteristic of our proposal, due to the fact that an object can only gain acquaintance to a newly created Tuple Space through messages in an distributed environment.

2.3 The Tuple Space Communication Kernel

The fundamental functionality of the Tuple Space communication is defined by the *Tuple Space Communication Kernel*. It consists of *communicating objects*, which serve as media of communication among the distributed objects. *Kernel Communication Classes* are classes of communicating objects The functionality provided by the Kernel is minimum, but can be extended using inheritance (see section 3).

In formulating the definition of the Kernel, Smalltalk-80 [26] is used as the base language in our prototype system. As Smalltalk-80 already provides its own message sending mechanism, the system is hybrid, where message sends to objects that are direct acquaintance to the sender are done by conventional means, and those to objects that are distributed within the system are done with Tuple Space communication. We can, however, construct a system where communication among the objects are done solely via Tuple Spaces; in [32] it is shown that the expressive power of Tuple Space communication can simulate the conventional direct message sends. (Note that semantics of message sends can be different from Smalltalk. For example, all Smalltalk message sends suspend the sender until a response returns from the receiver, whereas the put: message to the Tuple Space does not.)

Essential Kernel Communication Classes are as follows:

- Class Tuple (Tuple)

Tuples are instances of class Tuple. Each Tuple is an arbitrary sequenced collection of element objects. The objects are concurrent objects, and possess the functionality to perform pattern matchings within the Tuple Space. (Hereafter, we adopt a syntax so that the Tuples can be literally used in the program: elements of a Tuple are sequentially listed separated by spaces; furthermore, the entire list is parenthesized, e.g., (#foo 123 bar).)

- Class Formal (Formal Arguments)

Instances of Formal represent formal arguments; they only hold class information which is used in the communication.

- Class TS (Tuple Space)

Tuple Spaces are instances of class TS, and are distributed throughout the system. (There is a issue regarding the identity of the returned Tuple against the original receiver Tuple, but we omit the discussion for brevity: for details, see [32].)

2.4 Functionality of the Kernel Communication Classes

1. Pattern Matching Functionality of Class Tuple

The existence of inheritance complicate the rules of pattern matching. For classes $C1$ and $C2$, define $C1 > C2$ meaning $C1$ is superclass of $C2$ [33]. Here, we use an informal definition: let $f(C)$ be the functionality of C, or a set of messages that an instance of C understands. Then, we assume that the relation

- $$C1 \geq C2 \Leftrightarrow f(C1) \subseteq f(C2) \tag{2.1}$$

holds. In addition we define a function *class*(*e*), which denotes the class of the object:

- $$class(e) \equiv \begin{cases} e \text{ is an instance of Formal} & \rightarrow \text{class of object which } e \text{ can replace} \\ otherwise & \rightarrow \text{class of object } e \end{cases} \tag{2.2}$$

The rules of pattern matching for each formal-actual combination of elements of sender and receiver Tuples are as follows. (We denote the element of the sender Tuple with e_s and that of the receiver with e_r.)

- actual × formal

  The receiver expects the functionality of the argument passed by the sender to be at least equivalent, i.e., $f(class(e_s)) \supseteq f(class(e_r))$. This implies $class(e_s) \leq class(e_r)$. Upon instantiation, e_r is replaced by e_s in the receiver Tuple.

- formal × actual

  The sender sends the message to a receiver whose Tuple satisfies the specification of its own. Using a similar argument as in the above case, we obtain $class(e_s) \geq class(e_r)$.

- actual × actual

  Both objects must have a functionality for determining equivalences with other objects. A match is made if e_s and e_r are determined to be equivalent. The problems are that: a) whether reflexivity, commutativity, and associativity are satisfied by equivalence determining functionality of both objects, and b) equivalence does not necessarily imply neither $class(e_s) \geq class(e_r)$ nor $class(e_s) \leq class(e_r)$. we do not have good solutions for resolving the latter; at present, we require either $class(e_s) \geq class(e_r)$ or $class(e_s) \leq class(e_r)$.

- formal × formal

  Even in our extended model, this sender-receiver pair never matches, because we cannot say whether assuming either $class(e_s) \geq class(e_r)$ or $class(e_s) \leq class(e_r)$ is correct. (The sole exception to this rule is the generic Formal any, which can match arbitrary objects, including formals.)

2. Functionality of Class Formal

- Formal ofClass: <class>
- Formal any

  The former creates an instance of Formal which can be instantiated by the class given in the argument. The latter creates a generic instance of Formal which can be instantiated by an object of any class, except other Formals (they can only be matched).

With the pattern matching rules as defined, instances of Formal can be instantiated with an actual object whose class is at least functionally equivalent. When instantiation occurs, only local references to the object should change.

3. Functionality of Class TS

- <Tuple Space> put: <Tuple>

  This operation is identical to the out() operation of Linda. The sender object invokes the send operation by inserting the specified sender Tuple into the Tuple Space. This operation is asynchronous, i.e., the sender is not blocked.

- <Tuple Space> set: <Tuple>

  This operation is similar to put:, except that it is synchronous — the sender is blocked until a matching receiver Tuple is found in the Tuple Space[1].

- <Tuple Space> get: <Tuple>

  This operation is identical to the in() operation of Linda. The receiver object invokes the receive operation by inserting the specified receiver Tuple into the Tuple Space. This operation is synchronous — the receiver is blocked until a matching sender Tuple is found, upon which the instantiated Tuple is extracted and returned to the receiver.

- <Tuple Space> read: <Tuple>

  This operation is identical to the read() operation of Linda. It is similar to get:, except that the instantiated Tuple is not extracted from the Tuple Space, but a copy is created and returned to the receiver.

2.5 Simple Examples Using Tuple-Space Communication

We present a solution to the classic example of Dining Philosophers problem due to E. Dijkstra. The solution in [34], written in Concurrent Pascal [3], creates a set of abstract objects whose internal states become true if forks are available. But introduction of such objects that do not appear in the problem is not 'natural'. Furthermore, the forks are modeled as processes in the program; but in reality, forks are static objects to be acquired, not to respond to messages inquiring if they are 'available' or not. Since method invocation upon message receive has semantics identical to dynamically bound procedure calls [35], the same problem would arise for Object-Oriented languages as well, even if objects become critical regions themselves and asynchronous message send were used [18, 13].

The solution in Tuple Space Smalltalk demonstrates the effectiveness of Object-Oriented Tuple Space communication. Philosophers are active objects, and alternately think and dine. They are instances of class Philosopher. By contrast, forks are static entities, and are instances of class Fork; they can only be acquired by the philosophers. There are n instances of Philosopher and Fork. The arbitration of acquisition of forks by the philosophers depends solely on their mutual cooperation. i.e., availability of forks should be judged only by the philosophers themselves. Philosophers do not communicate directly with each other, but instead, the announcement of availability of forks is broadcasted. We do not model the program so that arbitration is controlled by tables, rooms, and such.

We let the forks be the elements of a Tuple. The ith element of the Tuple is an instance of Fork if the fork is available, else, it is nil. The ith philosopher uses ith and $(i + 1 \bmod n)$th forks to dine. The philosophers communicate with each other through a Tuple Space aTS. At the beginning of the program, a single Tuple of arity n, whose elements are all instances of Fork, is inserted into aTS. Each active philosopher, in turn, executes the following program:

```
action     (for the ith philosopher)
    | tuple leftFork rightFork |
    [
        :
        "think"
        :
        tuple <- ( a tuple whose ith and (i + 1)th
                   elements are Formal ofClass: Fork
                   and the rest are Formal any ).
        aTS get: tuple.
        leftFork <- tuple at: i.
        rightFork <- tuple at: i + 1 \\ n.
        tuple at: i put: nil; at: i + 1 \\ n put: nil.
        aTS put: tuple.
        :
        "dine with leftFork and rightFork"
        :
        tuple <- ( a tuple whose ith and (i + 1)th
                   elements are nil
                   and the rest are Formal any).
        aTS get: tuple.
        tuple at: i put: leftFork; at: i + 1 // n put: rightFork.
        aTS put: tuple.
    ] loop
```

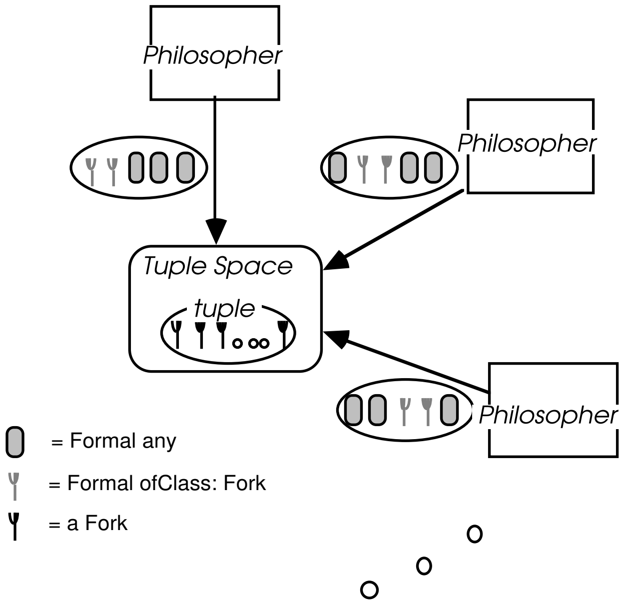

Figure 2.4 — The Dining Philosophers Problem:

At the first half of the loop, the Tuple received should contain the forks requested by the philosopher. After extracting the forks and replacing them with nil, the Tuple is inserted back into aTS. After dining with the forks, the Tuple is received again so that the forks can be put back for use by other philosophers. The atomicity of Tuple Space

[1]It is possible to express set: with the combination of put: and get:

communication guarantees that both forks are obtained atomically; therefore, the deadlock-free property is guaranteed.

3. EXTENSION OF KERNEL TUPLE COMMUNICATION CLASSES

3.1 Extensions for Communicating with Multiple Tuples/Tuple Spaces

We next define extensions to the primitive Kernel operations so that communications may be performed with multiple Tuples simultaneously.

- <Tuple Space> put: <Tuple> and: <Tuple> {and: <Tuple> ...}
- <Tuple Space> set: <Tuple> and: <Tuple> {and: <Tuple> ...}
- <Tuple Space> set: <Tuple> or: <Tuple> {or: <Tuple> ...}

  These operations are extensions to the message send operations. The sender is not blocked (put:and:), blocked until all its Tuples are instantiated by the matching receiver Tuples (set:and:), or blocked until at least one of its Tuples are instantiated (set:or:).

- <Tuple Space> get: <Tuple> and: <Tuple> {and: <Tuple> ...}
- <Tuple Space> read: <Tuple> and: <Tuple> {and: <Tuple> ...}

  These are extensions to the message receive operations. The receiver is blocked until all its Tuples are instantiated by the matching sender Tuples. An OrderedCollection of Tuples is returned by the operation. Within the collection, the Tuples retain their syntactical ordering specified in the operation. All the matching sender Tuples are extracted from the Tuple Space for get:.

- <Tuple Space> get: <Tuple> or: <Tuple> {or: <Tuple> ...}
- <Tuple Space> read: <Tuple> or: <Tuple> {or: <Tuple> ...}

  These are extension to the message receive operations. The sender is blocked until at least one its Tuples is instantiated by the matching sender Tuples. The Tuple returned is the one of the Tuples matched. The extraction of the Tuples are non-deterministic for get:; at least one, and at most all, of the matching sender Tuples are extracted.

We also extend the operations so that a single communication operation may be performed with multiple Tuples Spaces at the same time. This allows, for example, an object to wait for messages from multiple discrete communication domains. The returned Tuples in the receive operations obey rules similar to the above operations.

- <Tuple Space> set: <Tuple> andInTS: <Tuple Space> set: <Tuple> {andInTS: <Tuple Space> set: <Tuple> ...}
- <Tuple Space> get: <Tuple> andInTS: <Tuple Space> get: <Tuple> {andInTS: <Tuple Space> get: <Tuple> ...}
- <Tuple Space> get: <Tuple> orInTS: <Tuple Space> get: <Tuple> {orInTS: <Tuple Space> get: <Tuple> ...}
- <Tuple Space> read: <Tuple> andInTS: <Tuple Space> read: <Tuple> {andInTS: <Tuple Space> read: <Tuple> ...}
- <Tuple Space> read: <Tuple> orInTS: <Tuple Space> read: <Tuple> {orInTS: <Tuple Space> read: <Tuple> ...}

We can actually conceive every possible combinations of Tuples, Tuple Spaces and wait conditions. In general, for each Tuple Space TS_1, ..., TS_m, there can be n_i Tuples t_1^i, ... , t_n^i involved in the operation. The combination of waiting conditions can be expressed with valid combinations of and:, or: and andInTS:, orInTS:. Tuples that are operands of and: or andInTS: block the object until they are all instantiated. An OrderedCollection of all the Tuples are returned, and all the matching sender Tuples are extracted from the Tuple Spaces. Tuples that are operands of or: or orInTS: block the object at least one of them is instantiated. Only one of the matching Tuples are selected non-deterministically and returned.

3.2 Extension of Kernel Functionality with the Use of Inheritance

It is also possible to employ the inheritance mechanism in order to enhance the functionality of the Kernel Classes. This is possible because we have defined the Tuples to be first-class objects in our model. The inheritance mechanism may be the normal taxonomical inheritance found in most Object-Oriented languages, or the Part-Whole mechanism similar to the one proposed by [36].

Inheritance, in general, assumes that messages are sent to the objects directly. In our prototype system, communication operations defined by the Kernel classes has the same syntax as direct message sends in Smalltalk-80. Hence, it is possible to define that the operations are inherited as normal methods are, i.e., subclasses of Kernel classes can accept requests for communication operation from other objects identically as their superclasses. As noted earlier, however, the semantics of the message sends may be different from direct message sends; for example, it is possible for the sender not to be suspended.

Newly defined communication operations of subclasses of TS must 'trap' and preprocess the Tuple before it inserts it into the Tuple Space. This has the drawback that, if the operation is mutually exclusive, degrades the utility of the Tuple Space communication by not allowing concurrent communication operation with the Tuple Space. The advantages overcome the deficiencies, however, for special Tuple Spaces that arbitrate complex synchronized interactions among multiple concurrent objects. For example, [37] proposes a special language construct called the *Compact* which serves as a mediator for multiple process to synchronize with each other. By using inheritance, we were easily able to define CompactTS, which emulates the basic functions of the Compact.

4. REALIZABILITY OF THE TUPLE SPACE COMMUNICATION KERNEL

In our proposal, the efficiency of the entire system depends on the power of the Tuple Spaces to process the pattern matching among the Tuples as efficiently as possible. This requires performing a difficult task of establishing mutual control among the Tuples so that multiple pattern matchings proceed without conflicts.

We have developed an algorithm so that high degree of parallelism can be achieved within a Tuple Space. The pattern matchings of Tuples are done concurrently, and the activities of sender and receiver Tuples are carefully controlled to avoid mutual conflicts. The conceptual structure of the Tuple Space employed in the algorithm is illustrated in Figure 4.1.

The algorithm must satisfy the following criteria: 1) correctness, 2) freeness of deadlock, 3) freeness of starvation, 4) fairness, and 5) efficiency. Here, we will avoid formal discussions for brevity. In [32], proofs of algorithm correctness and criteria satisfaction are given.

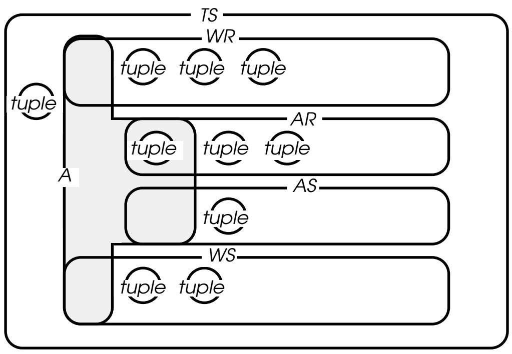

Figure 4.1 — Structure of the Tuple Space: The Tuple Space *TS* consists of four collections, *WS*, *WR*, *AS*, and *AR*. A Tuple may be a member of at most one of them at the same time. There is another atomic object, *A*, whose membership is mutually excluded.

5. IMPLEMENTATION OF THE PROTOTYPE TUPLE SPACE SMALLTALK

We are developing the prototype *Tuple Space Smalltalk System* in order to evaluate the effectiveness of Tuple Space communication in distributed Object-Oriented languages. The prototype language is being implemented atop the *Concurrent Smalltalk* [15, 16]. As stated earlier, Tuples and Tuple Spaces are objects, i.e., instances of actual Smalltalk classes. The basic functionality of the Kernel classes are implemented as Smalltalk methods. Concurrency in the system is achieved with concurrent objects of the Concurrent Smalltalk. At present, the basic Kernel is finished, and we are working to implement the extensions.

The Tuple Space communication is intended for implementation on actual distributed systems. But because Concurrent Smalltalk is not actually distributed (yet), our prototype, as a result, does not run on a true distributed system; however, we expect to obtain considerable programming experiences with the prototype. [32] discusses in detail the various implementation issues associated with Kernel communication classes, as well as the anticipated problems and their possible solutions in distributed systems.

We are also investigating the effectiveness of the distributed Tuple Spaces in distributed coordination of objects [32]. In one example, Japanese game of "Janken" is played distributively among a group of autonomous players selecting their 'hands' and determining the result of each game. Another example outlines how the Tuple Space communication might be used for implementing high-level protocols such as the contract net protocol. In the example, it is shown how environmental factors that only affect close proximities can be modeled with distributed Tuple Spaces, and how the time and space uncoupling characteristics play essential roles.

6. FUTURE WORKS

6.1. Further Extensions to the Kernel.

It is possible to add a non-blocking receive primitive that allows 'peeking' into the Tuple Space, and returns with a nil result if there are no matches. Alternatively, the Tuple object can become a *future* object [38], which is substituted with an instantiated receiver Tuple when a match occurs; the thread of control blocks only when a message is sent to an uninstantiated Formal object. The former is simpler, but requires active checking on the part of the programmer. The latter offers greater transparency of the operation, but requires some changes in the Tuple Space communication model, because a Tuple outside the Tuple Space is affected after the communication has taken place.

Another possible extensions to the model is to allow hierarchical nesting of Tuples, so that Tuples can become elements of Tuples. This allows passing of uninstantiated or arbitrary size Tuples via Tuple Space communication.

6.2 Inheritance

As noted earlier, inheritance of Tuples as messages is not provided in Tuple Space communication because it is difficult to specify where in the superclass script the message should be accepted. Explicit delegation of messages [39] is an alternative to inheritance for sharing knowledges among objects. We feel that delegation is favorable when distribution is taken into account; when object migration occurs, inconsistency in the distributed class definition can be resolved with dynamic changes in the target of delegation. We also feel, however, that better linguistic support is necessary to organize the delegation relation among the objects, and are currently investigating this matter.

6.3 Formal Investigation

Further investigation into formal aspects of our model is also necessary, especially temporal characteristics of Tuples and Tuple Spaces. We would like to be able to formally express the temporal behavior of the Tuples in more precise manner than we have done. More powerful mathematical tools, would probably be required for this purpose.

6.4 Application to Actual Distributed Systems and Problem Solving

As noted earlier, our current prototype does not run on a true distributed environment. For practical purposes, our next system must be implemented as a distributed one. Underlying problems in inter-object communication will probably surface, along with the problems caused by distribution of class definitions. Some of the limitations of our prototype is caused by the restrictions imposed by Smalltalk-80. Extensions to adapt Smalltalk-80 to distributed environment, or creation of a language better suited for distribution, is necessary. Applications to actual distributed problem solving is also required; we are currently looking for a field where the advantages of Tuple Space communication are best utilized.

7. ACKNOWLEDGEMENTS

We would like to thank Sony Inc. for the NEWS workstation; Yasuhiko Yokote and Mario Tokoro for the VM of their Concurrent Smalltalk; members of the Maekawa Lab. for the use of their SUN-160; and members of the Kawai Lab., who gave numerous helpful comments and discussions during the course of this research.

8. REFERENCES

[1] Pyle, I. C., The Ada Programming Language, Prentice-Hall International, Eaglewood Cliffs, NJ, 1981.

[2] Silberschatz, A., "Cell: A Distributed Computing Modularization Concept," IEEE Trans. Soft. Eng., no. 2, pp. 178-185, Mar., 1984.

[3] Hansen, B. P., "The Programming Language Concurrent Pascal," IEEE Trans. Software Eng., no. 2, pp. 199-207, Jun., 1975.

[4] Hoare, C.A.R., "Communicating Sequential Processes," Comm. ACM, no. 8, pp. 666-677, Aug., 1978.

[5] Hansen, B. P., "Distributed Processes: A Concurrent Programming Concept," Comm. ACM, no. 11, pp. 934-841, Nov., 1978.

[6] Holt, R. C., Concurrent Euclid, The Unix System, and Tunis, Addison-Wesley, Reading, MA, 1983.

[7] Feldman, J. A., "High Level Programming for Distributed Computing," Comm. ACM, no. 6, pp. 353-368, Jun., 1979.

[8] Andrews, G. R., "Synchronizing Resources," ACM Trans. Prog. Lang. and Systems, no. 4, pp. 405-430, Oct., 1981.

[9] Andrews, G. R. and Schneider, F. B., "Concepts and Notations for Concurrent Programming," Computing Surveys, no. 1, pp. 3-43, Mar., 1983.

[10] Stemple, D. W. et al, "Functional Addressing in Gutenberg: Interprocess Communication without Process Identefiers," IEEE Trans. Soft. Eng., no. 11, pp. 1056-1066, Nov., 1986.

[11] Lieberman, H., Concurrent Object-Oriented Programming in Act 1, in Object-Oriented Concurrent Programming, MIT Press, pp. 9-36, Cambridge, MA, 1987.

[12] Agha, G. and Hewitt, C. E., Concurrent Programming Using Actors, in Object-Oriented Concurrent Programming, MIT Press, pp. 37-54, Cambridge, MA, 1987.

[13] Yonezawa, A. et. al., Modelling and Programming in an Object-Oriented Language ABCL/1, in Object-Oriented Concurrent Programming, MIT Press, pp. 55-90, Cambridge, MA, 1987.

[14] Hur, J.H. and Chon K., "Overview of a Parallel Object-Oriented Language CLIX," Proceedings of ECOOP '87, pp. 265-273, Springer-Verlag, Jun., 1987.

[15] Yokote, Y. and Tokoro, M., "Design and Implementation of Concurrent Smalltalk," OOPSLA '86 Conference Proceedings, Meyrowitz, N., pp. 331-340, ACM SIGPLAN, 1986.

[16] Yokote, Y. and Tokoro, M., Concurrent Programming in Concurrent Smalltalk, in Object-Oriented Concurrent Programming, MIT Press, pp. 129-158, Cambridge, MA, 1987.

[17] Black, A. et. al., "Distribution and Abstract Yupes in Emerald," IEEE Trans. Soft. Eng., no. 1, pp. 65-75, Jan., 1987.

[18] Ishikawa, Y. and Tokoro, M., Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation, in Object-Oriented Concurrent Programming, MIT Press, pp. 159-198, Cambridge, MA, 1987.

[19] America, P., POOL-T: A Parallel Object-Oriented Language, in Object-Oriented Concurrent Programming, MIT Press, pp. 199-220, Cambridge, MA, 1987.

[20] Kahn, K. et al., "Objects in Concurrent Logic Programming Languages," SIGPLAN Notices, no. 10, pp. 29-38, Oct., 1986.

[21] Hewitt, C. E., "Viewing Control Structures as Patterns of Passing Messages," Journal of Artificial Intelligence, no. 3, pp. 323-364, Jun., 1977.

[22] Strom, R., "A Comparison of the Object-Oriented and Process Paradigmes," SIGPLAN Notices, no. 10, pp. 88-97, Oct., 1986.

[23] Briot, J.-P. and Yonezawa, A., "Inheritance and Synchronization in Concurrent OOP," Proceedings of ECOOP '87, pp. 32-40, Springer-Verlag, Jun., 1987.

[24] America, P., "Inheritance and Subtyping in a Parallel Object-Oriented Language," Proceedings of ECOOP '87, pp. 234-242, Springer-Verlag, Jun., 1987.

[25] Bennett, J. K., "The Design and Implementation of Distributed Smalltalk," OOPSLA '87 Conference Proceedings, pp. 318-330, 1987.

[26] Goldberg, A. and Robson, D., Smalltalk-80: Language and its Implementation, Addison-Wesley, Reading, MA, 1983.

[27] Smith, R. G. and Daivs, R., "Frameworks for Cooperation in Distributed Problem Solving," IEEE Trans. SMC, no. 1, pp. 61-70, Jan., 1981.

[28] Decouchant, D., "Design of a Distributed Object Manager for the Smalltalk-80 System," OOPSLA '86 Conference Proceedings, Meyrowitz, N., pp. 444-452, ACM SIGPLAN, 1986.

[29] Gelernter, D., "Generative Communication in Linda," ACM Trans. on Prog. Lang. and Systems, no. 1, pp. 80-112, Jan., 1985.

[30] Hendler, J., "Enhancement for Multiple Inheritence," SIGPLAN Notices, no. 10, pp. 98-106, Oct., 1986.

[31] Wesson, R. et. al., "Network Structures for Distributed Situation Assessment," IEEE Trans. SMC, no. 1, pp. 5-23, Jan., 1981.

[32] Satoshi Matsuoka, “Tuple Space Communication in Distributed Object-Oriented Computing,” Thesis, University of Tokyo, 1988.

[33] Cardelli, L. and Wegner, P., “On Understanding Types, Data Abstraction, and Polymorphism,” Computing Surveys, no. 4, Dec., 1985.

[34] Filman, R. F. and Friedman, D. P., Coordinated Computing: Tools and Techniques for Distributed Software, McGraw-Hill, 1984.

[35] Wegner, P., “Classification in Object-Oriented Systems,” SIGPLAN Notices, no. 10, pp. 173-182, Oct., 1986.

[36] Blake, E. and Cook, S., “On Including Part Hierarchies in Object-Oriented Languages,” Proceedings of ECOOP '87, pp. 41-50, Springer-Verlag, Jun., 1987.

[37] Charlesworth, A., “Multiway Rendezvous,” ACM Trans. on Prog. Lang. and Systems, no. 2, pp. 350-366, Jul., 1987.

[38] Yonezawa, A. and Tokoro, M., Object-Oriented Concurrent Programming: An Introduction, in Object-Oriented Concurrent Programming, MIT Press, pp. 1-7, Cambridge, MA, 1987.

[39] Lieberman, H., “Using Prototypical objects to Implement Shared Behavior in Object-Oriented Systems,” OOPSLA '86 Conference Proceedings, Meyrowitz, N., pp. 214-223, ACM SIGPLAN, 1986.