Customization of First-Class Tuple-Spaces in a Higher-Order Language

Suresh Jagannathan
NEC Research Institute
Princeton, NJ 08540
suresh@research.nec.com

Abstract

A distributed data structure is an object which permits many producers to augment or modify its contents, and many consumers simultaneously to access its component elements. Synchronization is implicit in data structure access: a process that requests an element which has not yet been generated blocks until a producer creates it.

In this paper, we describe a parallel programming language (called $\mathcal{TS}$) whose fundamental communication device is a significant generalization of the tuple-space distributed data structure found in the Linda coordination language[6]. Our sequential base language is a dialect of Scheme[19].

Beyond the fact that $\mathcal{TS}$ is derived by incorporating a tuple-space coordination language into a higher-order computation language (*i.e.,* Scheme), $\mathcal{TS}$ differs from other tuple-space languages in two important ways:

- Tuple-spaces are first-class objects. They may be dynamically created, bound to names, passed as arguments to (or returned as results from) functions, and built into other data structures or tuples.
- The behavior of tuple-spaces may be customized. A tuple-space is manipulated via a *policy closure* that specifies its operational characteristics. The representation of policy closures take significant advantage of Scheme's support for higher-order functions; there is no fundamental extension to Scheme needed in order to support them.

We argue that first-class customizable tuple-spaces provide an expressive and flexible medium for building parallel programs in higher-order languages.

*Appeared in the Proceedings of the Conference on Parallel Architectures and Languages, Europe. Published as Springer-Verlag LNCS 506.

1 Introduction

Distributed data structures are widely recognized to be an important device in structuring explicitly parallel programs. A distributed data structure is valuable because it abstracts low-level details about process synchronization and communication to high-level algorithmic design issues involving data structure access and generation. Generally speaking, the semantics of such structures permit many producers to augment or modify its contents, and many consumers simultaneously to access its component elements. Synchronization is implicit in data structure access: a process that requests an element which has not yet been generated blocks until a producer creates it. Some notable examples of distributed data structures are the blackboard object in Shared Prolog[2], stream abstractions in Flat Concurrent Prolog[20], Concurrent Smalltalk's distributed objects[13] and its closely related variant Concurrent Aggregates[7], the I-structure in Id[4] and C.Linda's flat tuple-space[6].

In this paper, we describe a parallel programming language (called $\mathcal{TS}$) whose fundamental communication device is a significant generalization of the tuple-space distributed data structure found in the Linda coordination language. Our sequential base language is a dialect of Scheme[19].

By way of introduction, a $\mathcal{TS}$ tuple-space defines a shared associative memory whose elements are ordered sets of values or processes known as *tuples.* Fields in passive tuples, *i.e.,* tuples containing only values, can be retrieved using a simple pattern matching procedure. In the default case, failure to produce a substitution causes the process that initiated the match operation to block. A blocked process may resume only after a matching tuple is deposited into the specified tuple-space. An active tuple, *i.e.,* a tuple containing concurrently executing processes, turns into a passive one when each process completes and returns a value. Active tuples are not involved in the matching procedure. [5] discusses how distributed data structures found in other programming models may be implemented using tuple-spaces. [15] discusses compile-time analysis techniques for first-class tuple-spaces in higher-order languages.

Beyond the fact that $\mathcal{TS}$ is derived by incorporating a tuple-space coordination language into a higher-order computation language (Scheme), $\mathcal{TS}$ differs from other tuple-space languages in two important ways:

1. Tuple-spaces are first-class objects. They may be dynamically created, bound to names, passed as arguments to (or returned as results from) functions, and built into other data structures or tuples.

2. The behavior of tuple-spaces may be customized. Tuple-spaces are accessed via a set of policy procedures that define its various attributes.

> These attributes include (a) access to the tuple-space that is to be manipulated, (b) the matching protocol that governs tuple retrieval, (c) the blocking policy that specifies the conditions under which processes accessing this tuple-space block, and (d) the failure policy that specifies conditions under which a tuple-space operation fails.

There are two classes of tuple-space operations – those that store into a tuple-space, and those that read from it. In either case, tuple-spaces are manipulated via a set of policy definitions encapsulated within a closure that is a mandatory argument to the operation. Thus, an expression of the form, `(put` P *tuple*`)`, evaluates P to get a policy closure and uses the definitions found in this closure to determine if and where *tuple* is to be deposited.

Policy definitions are defined in terms of ordinary Scheme procedures. Their utility derives from the fact that they can be used to transform a tuple-space into a distributed data structure analogue of a sequential data abstraction – the representation of a tuple-space structure can be manipulated and its behavior may be tailored to conform to (or take advantage of) particular operational requirements. Tuple-spaces are now properly regarded as objects whose behavior is determined by the definition of its policy procedures.

The paper is organized as follows. The next section gives motivation for the design of the language; Section 3 gives an overview of the language, and describes the specification of tuple-space operations. Section 4 describes a number of examples whose formulation is significantly simplified by the presence of first-class customizable tuple-spaces.

2 Motivation

The motivation for the design of $\mathcal{TS}$ is two-fold. First, we are interested in building highly-modular parallel systems for symbolic computation. First-class tuple-spaces offer themselves as an expressive modularity device in this regard.

Second, we wish to apply abstraction, customization and parameterization techniques commonly used in the specification and implementation of sequential data structures to their distributed data structure counterparts. Our contention is that these techniques can reduce complexity, and enhance expressivity. We elaborate on these two points below.

2.1 Modularity

There has been much recent interest on incorporating explicit concurrency into expression-based programming languages. Mul-T[17] and MultiLisp[12] are languages that augment Scheme with a *future* construct; Concurrent Prolog[21] and Parlog[8] are logic programming languages that are based on an asynchronous process interpretation of clause evaluation. Concurrent Smalltalk[13] and Actors[1] are representative examples of programming languages that extend object-based programming with parallel facilities.

$\mathcal{TS}$ is distinguished from these other efforts in several important respects. By way of comparison, synchronization in concurrent logic programming and other parallel Lisp dialects takes place through low-level primitives (*e.g.,* futures and semaphores in Mul-T) and shared variables (*e.g.,* read-only variables in Concurrent Prolog). Process communication in $\mathcal{TS}$, on the other hand, is decoupled from process instantiation. This attribute makes it possible for processes to initiate requests for the value of objects even if the object itself has not yet been created, and to collectively contribute to the construction of shared objects. These capabilities are absent in many other distributed data structure-based systems (*e.g.,* parallel Lisps[17] or object-based concurrent systems[13]), and have no trivial formulation in a concurrent logic framework[21].

Giving first-class status to tuple-space objects also encourages greater modularity and simplifies implementation[10, 15]. By permitting tuple-spaces to be denoted, we allow the programmer to partition the communication medium as he sees fit. Conventional namespace management techniques available in the base language can be easily applied over tuple-spaces as well. To encapsulate a set of related shared data objects, we deposit them within a single tuple-space; this tuple-space can be made accessible only to those processes that require access to these objects. A set of related processes can also reside within their own tuple-space; data values which they share can be deposited and retrieved within this structure. Other processes need not be aware of these objects, and ad hoc naming conventions need not be applied to ensure that data objects are not mistakenly retrieved by processes which do not need them.

2.2 Customizability

The semantics of other tuple-space languages restrict the ways in which tuple-spaces can be manipulated. In general, the only operations permitted on a tuple-space are those which deposit, read, remove, or test the presence/absence of tuples. Customizing matching protocols, determining where tuples should be deposited based on conditions known only at runtime, specifying the constraints (outside of a match failure) under which processes should block, or specifying constraints under which tuple-space operations may *fail* but not block, are capabilities not available in these languages.

One immediate consequence of preventing user specification and customization of the operational behavior of tuple-spaces is reduced flexibility: while it is easy to implement data structures whose semantics can be naturally expressed using just `get` (remove tuple), `rd` (read tuple), `put` (deposit tuple) and `spawn` (fork process) operations[2], it becomes problematic to implement structures that don't fit neatly into this framework or which require additional constraints not expressible in terms of these operations. Conceptually, a tuple-space is a data abstraction whose representation is manipulated via these operators; the inability of programmers to customize their behavior limits the ease with which different kinds of distributed data structures can be specified.

For example, given a C-Linda version of a distributed object O, one can trivially deposit a method or instance variable x by writing:

`put("O","x",`x`)`

(`"O"` is a label that tags all operations associated with O, and `put` deposits its tuple argument into a global tuple-space.). To send a message M to O one writes:

`rd("O","`M`",?v)`

"?v" introduces an unbound variable or *formal*; the variable is assigned the value of the third field in the tuple to which this tuple template is matched. If v is bound to a method represented as a procedure, one can subsequently apply this procedure to arguments. Tuple-spaces in this simple framework are distributed analogues of a conventional record containing procedure and scalar-valued fields.

Suppose, however, we wish to have these distributed objects adhere to an inheritance protocol[11]: O is to automatically dispatch messages to another object (its parent) if it does not define a field whose value is requested by the message. In other words, a tuple-space is now to be regarded as the representation of a class instance; and inheritance is realized by directing tuple-space operations to particular tuple-spaces. Since O defines a distributed version of a sequential object, we must also ensure that no `get` operations are permitted on its components; all instance variable and method bindings are intended to be immutable, although these bindings may be bound to mutable locations.

Satisfying these constraints is non-trivial within the standard Linda framework. To delegate operations in the manner suggested requires performing an explicit test to determine if a tuple containing the requested operation exists within the specified tuple-space; if it does not, the same test must be performed on its "parent"; such tests proceed along a tuple-space hierarchy until a matching tuple is found. If O has n ancestors, messages sent to O must explicitly include

[2] The names given to these operations in $\mathcal{TS}$ differ from those used in *e.g.,* C-Linda because the semantics of these operators differ from their counterparts in the Linda framework in several significant ways.

n conditional tests for each of these ancestors; operations on O's siblings and children must be structured similarly. The validity of `get` operations must also be explicitly monitored. Changes in the inheritance structure require global changes to the access operations on affected objects. Modularity is compromised and complexity increased.

$\mathcal{TS}$ provides an alternative solution to this problem. The conditions under which a tuple-space operation is to block or fail may be defined explicitly in terms of a policy procedure. Similarly, the particular semantics of matching or the tuple-space repository in which tuples are to be deposited or removed may all be specified in terms of ordinary Scheme procedures that operate over tuple-spaces.

Since tuple-space operations manipulate policy closures directly, processes that manipulate tuple-spaces need not be aware of their implementation, and no explicit bookkeeping information need be maintained to enforce correctness of its specification. The conceptual and implementation overhead of encapsulating tuple-space operations within user-defined procedures and applying these procedures as needed is absent; all tuple-space operations have a uniform syntax and self-consistent behavior.

We emphasize that policy definitions are *not* orthogonal to the semantics or implementation of tuple-spaces in any significant sense. Any specification of a tuple-space abstraction must contend with issues dealing with tuple access, matching, failure, and blocking. The semantics of tuple-spaces discussed in this paper makes these issues manifest to the $\mathcal{TS}$ programmer.

3 The Language

$\mathcal{TS}$ is a set!- and call/cc-free dialect of Scheme[19] with first-class tuple-spaces. We concentrate in this section on the coordination subset of $\mathcal{TS}$; this subset defines operations for creating and manipulating tuple-spaces. The interaction of tuple-spaces on other Scheme constructs is considered in Section 4.1.

3.1 Creating Tuple-Spaces

The coordination language upon which $\mathcal{TS}$ is based treats tuple-spaces as denotable objects with a distinguished type. To create a tuple-space, we evaluate `(make-ts)`. The value returned is a reference to a newly created tuple-space object. More precisely, this object returns a *representation* structure of a tuple-space as determined by suitable compile-time analysis[15]. In the abstract, a tuple-space, like any other Scheme structure, is accessed via pointer, not value; like any other Scheme object, a tuple-space may be garbage-collected if there remain no extant references to it.

Operations on a tuple-space T access it indirectly via a policy closure. This closure determines T's operational attributes; in particular, the policy closure encapsulates operations that specify the constraints under which operations block or fail, the protocol used to retrieve tuples are retrieved, and the mechanism by which tuples are to be retrieved.

A policy closure is a procedure of one argument. This argument determines the policy definition to be evaluated. Thus, a policy closure takes the form:

```
(lambda (op)
   (cond ((eq? op policy definition)
          (lambda (args) body)
              ⋮
         (else op))))
```

Tuple-space operations in $\mathcal{TS}$ make use of four such definitions:

1. The *access* policy specifies the tuple-space to be operated upon. Tuple-space operations use this policy to determine the tuple-space of interest. Since tuple-spaces are first-class, the same access policy can yield different tuple-spaces on different calls.
2. The *matching* policy specifies the set of tuples a tuple-template can match against. Match operations are initiated only by processes wishing to read or remove a tuple from a tuple-space.
3. The *blocking* policy specifies the conditions under which a process may not have access to this tuple-space.
4. The *failure* policy specifies the conditions under which a tuple-space operation returns `fail`. Failure does not imply blocking; the failure policy is used to realize non-blocking exceptional conditions on tuple-space access.

Each of these policy definitions have a default implementation defined by the system; these definitions are bound to the names: `:access`, `:match`, `:block` and `:fail` respectively. These default definitions are presumed global.

If T's policy closure is structured thus:

```
(lambda (op)
   (cond ((eq? op :match)
          (lambda (args) match policy))
         (else op)))
```

and is bound to T-`policy`, then evaluating: ((T-`policy :match`) *arguments*) applies (`lambda` (*args*) *match policy*) to *arguments*. In this example, T's policy closure does not define definitions for `:access`, `:fail` or `:block`; thus, the expression (T-`policy :fail`) returns the default failure policy.

Policy definitions all take three arguments:

1. `PC` – the policy closure of the current tuple-space.
2. `tuple` – the tuple argument being manipulated. We describe the policy operations allowable over tuples below.
3. `kind` – a symbol drawn from the set,

   { `read, store, remove, fork` }

   that indicates the type of operation being performed. Thus, `put` operations invoke a policy closure with kind `store`; `spawn` operations invoke a policy closure with kind `fork`; `get` operations have kind `remove`; `rd` operations have kind `read`.

3.1.1 Concrete Representations

Compile-time analysis of $\mathcal{TS}$ programs is used to determine efficient representations for tuples and tuple-space objects; this representation is derived by examining the structure and type of tuple components[15]. For example, a tuple-space T closed under a policy closure P_T that contains tuples of length two and whose `rd` and `in` operations are all of the form: (*op* P_T `(i,?v)`) for some integer-yielding expression i can be implemented in terms of a concurrent vector data structure. A tuple denotes an $<$ *index,value* $>$ pair in this vector; attempts to read an "empty" vector elements results in the process blocking; and processes that wish to update a vector element must gain exclusive access to it first. $\mathcal{TS}$ programmers can determine the representation type associated with any tuple-space using operators described below. Insofar as the representation structures generated by the implementation are visible to the program, $\mathcal{TS}$ bears resemblance to "reflective" languages[22, 25]. The underlying behavior of tuple-space operations, and the structures they manipulate can be examined and customized by the programmer.

Let R_T be a representation of a tuple-space T, and let R_t be a representation of a tuple within T. (The representation type of a tuple t is derived by evaluating `:rep_t` t). All concrete representations of tuples provide a set of operations for accessing various components; we introduce some of these operators during the course of the paper.

The representation object of a tuple-space must provide the following operations; given a tuple-space T, its representation is accessed by evaluating `(:rep` T`)`:

- `(:type` R_T`)` returns R_T's type; policy definitions dispatch on this type to determine the index and selection operations they should perform.
- `(:size` R_T`)` returns the number of elements in R_T.
- `(:empty?` R_T`)` returns true iff R_T contains no tuples.
- `(:write` R_T R_t`)` stores R_t into R_T.
- `(:with-lock` R_T `kind` *body*`)` grabs a suitable read or write lock on R_T (depending upon the value of `kind`) and evaluates *body*; the lock is released after *body* completes. Policy definitions manipulate read and write locks to ensure atomicity and serialization constraints.

In general, users needing only standard Linda functionality need not be concerned about manipulating tuple-space representations; the default policy definitions for tuple-spaces implement the necessary constraints. (See figure 1.)

The policies obeyed by a tuple-space created by evaluating `(default-ts)` are the same as, *e.g.,* C-Linda's global tuple-space. The behavior of the default `:match` policy is determined from the representation type of the argument tuple-space. The value returned is a list; the elements in this list are extracted from the fields of a resident tuple such that the i^{th} element in this element is to be bound to the i^{th} formal in the argument tuple. If there are several candidate tuples that satisfy the match requirements, one is chosen non-deterministically. The representation of a tuple-space is responsible for ensuring proper serialization; thus, two `get` operations that occur simultaneously on the same tuple-space must not result in the same tuple being matched for both operations.

The constraint imposed by the default block policy is dictated by the success of `:match`. If a `rd` or `get` operation is performed on a tuple-space such that the `:match` policy is unable to produce a matching tuple, `:block` returns `#block`. In all other instances, the policy procedure succeeds. The default `:access` policy returns a global tuple-space object. The `:fail` policy always succeeds in the default implementation – tuple-space operations either succeed or result in a process becoming blocked; there is no "failure without blocking" semantics in this implementation.

When `default-ts` is applied, it creates a new tuple-space using `make-ts`; the access policy associated with the policy closure returned yields this tuple-space.

```
(define (:match PC tuple kind)
   (let ((TS_rep (:rep ((PC :access) PC tuple kind)))
         (tuple_rep (:rep_t tuple)))
     (case (:type TS_rep)
        ((semaphore)   (read_sem TS_rep kind))
        ((queue)       (head TS_rep kind))
            ⋮)))

(define (:block PC tuple kind)
   (if (memq kind '(read remove))
       (let ((matches ((PC :match) PC tuple kind)))
          (if (null matches) #block matches))
       #succeed))

(define (:fail PC tuple kind)
    #succeed)

(define (:access PC tuple kind)
    (global-TS))

(define (default-ts)
   (let ((newTS (make-TS)))
     (lambda (op)
        (cond ((eq? op :access)
               (lambda (PC tuple kind) newTS))
              (else op)))))
```

Figure 1: One possible implementation of default policy definitions that conform to Linda-style semantics.

3.2 Depositing Tuples

A tuple is deposited into a tuple-space using one of two operators. Let `PC` be an expression that yields a policy closure; then, the expression:

(put PC (e_1, e_2, ..., e_n))

is evaluated as follows:

1. Evaluate `PC` to yield a policy closure P.
2. Evaluate each of the e_i in the current evaluation environment to get a value v_i. Let the tuple constructed by replacing each e_i by v_i be called t. Our notion of "value" is consistent with Scheme's[19] – constants and symbols are values as are references to closures, lists, tuple-spaces, and vectors.
3. Evaluate P's blocking policy definition:

   ((P :block) P t 'store)
4. If the result of this application is `#block`, block the current process. Otherwise, evaluate P's failure policy:

   ((P :fail) P t 'store))

   If the result of this application is `#fail`, return `#fail`. Otherwise, evaluate

   (let ((R_T ((:rep ((P :access) P t 'store)))))
      (:with-lock R_T 'store (:write R_T (:rep_t R_t)))

   and resume all processes blocked on this tuple. (`:rep_t` returns the tuple representation of its argument.)

`Spawn` is the *non-strict* counterpart of `put`. If `PC` is a policy closure, the expression

(spawn PC (e_1 e_2 ... e_n))

concurrently instantiates n *processes*; the i^{th} process in this ensemble is responsible for computing the value of e_i. Since processes cannot communicate via shared variables they may freely access common binding environments. When all processes complete, a passive tuple containing the values they have generated is deposited into the appropriate tuple-space by evaluating a `put` operation using the newly constructed passive tuple as the tuple argument.

3.3 Retrieving Tuples

`Put` and `spawn` generate tuples into tuple-space. `Rd` and `get` read tuples and binding-values from tuple-space. A `rd` expression takes the form:

(rd PC (e_1, e_2, ..., e_n) *body*)

Unlike tuple-generator expressions, each of the e_i may be either an ordinary $\mathcal{TS}$ expression *or* a formal. The evaluation of the above expression takes place as follows:

1. Evaluate `PC` to yield a policy closure P.
2. Evaluate each e_i that is *not* a formal in the current evaluation environment to get a value v_i. Let the tuple constructed by replacing each e_i by v_i be called t.
3. Bind each formal to `#undefined`.
4. Evaluate P's blocking policy definition:

   `((`P `:block)` P t `'read)`
5. If the result of this application is `#block`, block the current process; the representation type of any tuple-space must provide a queue to hold blocked processes. Otherwise, bind the object returned by the application to V and evaluate P's failure policy:

   `((`P `:fail)` P t `'read)`

   If the result of this application is `#fail`, return `#fail`. Otherwise, bind the i^{th} element in V to the i^{th} formal in t, and return the value yielded by evaluating *body* in this augmented environment.

The `get` operation is semantically identical to `rd` except that the tuple chosen for the match is removed from the given tuple-space.

4 Examples

We consider several simple examples to illustrate the utility of the language. The first is the implementation of a cell abstraction; the second is a specification of a dynamic load balancing system; the third is a sketch of an inheritance system using $\mathcal{TS}$ tuple-spaces as concurrent objects.

While each structure is useful in different domains, they are related to one another insofar as they impose conditions on their use not readily expressible using ordinary tuple-space operations; we argue that first-class tuple-spaces and customization improves the clarity of their definition.

4.1 Cells

$\mathcal{TS}$ contains no assignment operations on variables. Given that the constituent elements of a tuple-space are shared $\mathcal{TS}$ objects, arbitrary assignment operations on $\mathcal{TS}$ objects would permit "backdoor" inter-process communication through shared variables. Such a capability would violate the immutability constraint on tuples, and make it impossible to enforce mutual exclusion or atomicity constraints.

Nonetheless, despite the absence of explicit `set!` operations, the fact that data objects can be added to and removed from a tuple-space makes it straightforward to implement applications that require object mutation. Consider the implementation of a memory cell shown in figure 2.

A cell will always contain at most one element. A write operation on a cell uses the failure policy to remove an old element before writing a new one. Two processes attempting to write into the cell at the same time are serialized by the definition of the failure policy and `put`. The policy closure returned by `make-cell` is closed over a newly created tuple-space, `newTS`; the closure's access policy returns this tuple-space when applied. Compile-time analysis on the cell abstraction would reveal that the tuple-space holding the current state does not escape its lexical contour and only holds tuples of length one of string type. This tuple-space can be thus represented as an ordinary protected shared variable.

All side-effecting operations in Scheme are reimplemented in $\mathcal{TS}$ to check that the object to be mutated is indeed a cell. For example, to create a mutable vector of n arguments, we write:

```
(make-vector 10 (make-cell))
```

The operation:

```
(vector-set! vector k v)
```

is equivalent to:

```
(let ((obj (vector-ref v k)))
   (if (cell? obj)
       (write-cell obj v)
       error))
```

Cells elevate $\mathcal{TS}$ to a middle ground between purely functional and completely constructive languages. Pure functional languages (*e.g.,* Haskell[14]) eliminate any notion of global state; completely constructive languages (*e.g.,* Ada[24]) use state variables pervasively. Both language designs have significant implications in the context of concurrency. Functional languages prevent users from expressing non-deterministic or explicitly parallel programs; statement-based parallel languages often require bulky modularity devices (*e.g.,* monitors or rendezvous points) to ensure atomicity constraints. $\mathcal{TS}$, on the other hand, purports to serve as a bridge between these two proposals. The functional core of $\mathcal{TS}$ makes it straightforward to program in a mostly side-effect free style; nonetheless, the modularity and serialization properties of tuple-space make it easy to build to local mutable objects when required.

```
(define (make-cell)
  (let* ((newTS (make-ts))
         (state (default-ts)))
    (put state ("empty"))
    (cons 'cell
          (lambda (op)
              (cond ((eq? op :fail)
                     (lambda (PC tuple kind)
                        (let ((ts (:rep ((PC :access)
                                          PC tuple kind))))
                           (get state (?condition)
                                (cond ((and (equal? condition "empty")
                                            (memq kind '(store fork))))
                                       (put state ("full"))
                                       #succeed)
                                      ((memq kind '(read remove))
                                       #succeed)
                                      (else #fail))))))
                     ((eq? op :access)
                      (lambda (PC tuple kind) newTS))
                     (else op))))))


(define (read-cell cell)
   (rd (second cell) (?v) v))


(define (write-cell cell v)
   (let ((contents (second cell)))
     (if (fail? (put contents (v)))
         (get contents (?old-v)
              (write-cell cell v)))))


(define (cell? cell)
  (equal? (first cell) 'cell))
```

Figure 2: A $\mathcal{TS}$ implementation of a memory cell.

4.2 Dynamic Load Balancing

Besides allowing the implementation of different kinds of distributed data structures, customizable first-class tuple-space also permit the $\mathcal{TS}$ programmer to manipulate the process state of tuple-spaces. For example, consider a collection of tuple-spaces that represent virtual processors; each tuple-space contains a number of active processes and data elements. These processes communicate with one another via standard tuple-operations. Since processes are periodically run within a virtual processor, it is useful to allow process load on any given processor to be monitored and balanced dynamically, based on runtime conditions.

We present in figure 3 the structure of an abstraction that views virtual processors. The processors coordinate their activities by monitoring their load (*i.e.,* the number of extant processes they contain), and by handing incoming processes to less loaded processors when they become saturated.

In the implementation described here, `dynamic_load` returns a list of policy closures; the i^{th} closure corresponds to the i^{th} virtual processor in the processor ensemble. Each policy closure is closed over its index this ensemble. To permit dynamic load balancing, we specify an access policy that examines the current load of its virtual processor. If the policy is invoked by a `spawn` operation (which is the only device for instantiating new processes), and the current load on this processor has already reached its maximum, we apply the policy closure of its neighbor with the current arguments. In the case where the load has not reached its maximum, we simply increase the load factor by the number of processes being instantiated, and return the current tuple-space. When the processes instantiated by a `spawn` operation complete, the resulting passive tuple is recorded in the specified tuple-space via a `put` operation. Since such an operation signals the completion of a related set of processes, the load value of the virtual processor in which the passive tuple is to be deposited is decremented appropriately. (Of course, this implementation assumes that processes do not generate spurious `put` operations on these virtual processors.) In all other cases, the current tuple-space is returned with no other processing initiated.

Dynamic load balancing in the absence of policy procedures is possible by abstracting access policy decisions into user defined procedures that mediate all tuple-space operations into the processor set. Rather than writing expressions of the form:

`(spawn (nth proc_set i)` *tuple*`)`

we now need to invoke a dedicated procedure to determine the suitable tuple-space. Abstraction is compromised – `spawn` operations over ordinary tuple-spaces are distinguished from operations on the virtual processor ensemble;

```
(define (dynamic_load n)
   (let ((load-vector (make-vector n (make-cell))))
      (vector-fill! load-vector default-load)
      (let iterate ((i 0)
                    (proc_set '()))
         (cond ((= i n) proc_set)
               (else (iterate (+ i 1)
                         (cons proc_set
                            (load-policy (make-ts) proc_set
                                         load-vector i))))))))


(define (load-policy newTS proc_set load-vector i)
   (lambda (op)
      (cond ((eq? op :access)
             (lambda (PC tuple kind)
                (cond ((and (equal? kind 'fork)
                            (> (vref load-vector i)
                               *max-load*))
                       (let ((next-PC (nth proc_set
                                           (modulo (1+ i) n))))
                         ((next-PC :access)
                          next-PC tuple kind)))
                       ((equal? kind 'fork)
                        (increase! load-vector i
                                   (:tuple_length (:rep_t tuple)))
                        newTS)
                       ((equal? kind 'put)
                        (decrease! load-vector i
                                  (:tuple_length (:rep_t tuple)))
                        newTS)
                       (else newTS))))
            (else op)))))))))
```

Figure 3: A dynamic load balancing abstraction using a customized access policy definition.

different implementations of a load balancing policy entail different procedure interfaces. Policy closures present a transparent and seamless interface to tuple-spaces despite the fact that the behavior of tuple-space operations may greatly vary across distinct tuple-space objects.

4.3 Inheritance

The technique used to build a dynamic load balancing system using customized policy procedures can be applied in a very different domain with only slight modification. Consider the implementation of a Smalltalk-style[11] inheritance-based system. The basic entities in such system are objects that retain local state, and which communicate via message-passing. Moreover, objects are free to dispatch messages they receive to their parents in the object hierarchy if they determine that they cannot interpret them. A concurrent message-passing system permits objects to process many messages simultaneously; message sends are asynchronous, and objects may respond to many messages concurrently. The approach described here is simple, omitting many important details and disregarding any syntactic sugaring, but it captures the essence of inheritance-based programming relying only on policy definitions and first-class tuple-spaces to do so.

The ability to customize the behavior of tuple-spaces in $\mathcal{TS}$ allows a simple formulation of inheritance within a parallel system. The basic idea is to implement objects as tuple-spaces, and message passing as tuple reads. Methods and instance variables are stored as tuples. If a tuple-read operation is performed on a tuple-space T that contains no matching tuple, the operation is redirected using the `T`'s access policy definition to T parent in the object hierarchy. In other words, a tuple-template for whom no match exists in one tuple-space is *implicitly* sent to another tuple-space to be resolved. Implicit redirection of messages is the operational intuition underlying inheritance. Customization of tuple-spaces gives $\mathcal{TS}$ this capability.

Object methods are implemented as abstractions whose first argument plays the role of Smalltalk's `self` pseudo-variable – `self` yields the object containing the instance variables to be used by a method. Thus, a method definition takes the form:

```
(lambda (self . args)
    definition)
```

Objects are represented as policy closures. To invoke method M in object O with self argument `self` and arguments *args*, we write:

```
(rd O ('M ?v)
    (v self args))
```

Method definitions and instance variables are recorded within a tuple-space as two-tuples: $<$ *symbol*, *definition* $>$.

Thus, to augment object O with a new method definition M, we write:

```
(put O ('M definition))
```

An object O is created by applying `make-object`. The policy closure which it returns defines customized access and failure policies. O is closed over a tuple-space T that contains methods and instance variables. These definitions are recorded using `put` operations. When a put operation on O is evaluated, the method or instance variable name is recorded in a list (`objectList`); this list is referenced whenever a process attempts to send a message to this object. When the definition of a method or instance variable is required (via a `rd` operation), the access policy tests whether the symbolic reference to this method is defined within `objectList`; if it is, the tuple-space is returned and the default matching protocol is guaranteed to find a tuple containing a definition for the requested operation. If no such component exists in the current tuple-space, the access policy evaluates the object's parent; the tuple-space yielded by evaluation of the parent is the tuple-space on which the message is to be sent (*i.e.,* the tuple-space on which the `rd` operation is to be performed).

The failure policy prevents object components from being removed or initiating processes within objects. Mutable components must be realized by binding the component name to a cell and performing the necessary changes on the cell.

To illustrate how we might use customized tuple-spaces to build inheritance systems, we consider an example discussed in [9, 16]. A `circle` is a sub-class of a `point`. `Point` defines instance variables `x` and `y` to specify its location. It also defines two methods: `DistfromOrig` computes the distance of a point from its origin and `ClosertoOrig` is a predicate which given another point object as its argument returns true if its point is closer to the origin than its argument, and false otherwise. We use `self` to denote the object whose tuple contents define the environment within which an expression should be evaluated. The code for a `point` generator is given in figure 5.

A circle is defined in terms of points. Because circles have a radius, they have a different meaning of distance from the origin. The notion of distance from origin for circles is given in terms of the definition of `DistfromOrig` found in point objects: if l is the distance from the origin to the circle's center and r is the circle's radius, then $l - r$ gives the distance from the origin of the circle object. If this difference is negative, the distance is assumed to be 0. Thus, the meaning of `DistfromOrig` in a `circle` instance should refer to its meaning as

```
(define (make-object parent)
   (let ((newTS (make-ts))
         (objectList (make-cell))
     (write-cell objectList '())
     (lambda (op)
        (cond ((eq? op :access)
               (lambda (PC tuple kind)
                  (let* ((tuple_rep (:rep_t tuple))
                         (component (:tuple_index tuple_rep 1)))
                    (cond ((equal? kind 'store)
                           (write-cell objectList
                                       (cons component (read-cell
objectList))
                           newTS)
                          ((equal? kind 'read)
                           (if (memq component (read-cell objectList))
                                newTS
                                ((parent :access) parent tuple kind)))
                          (else newTS)))))
               ((eq? op :fail)
                (lambda (PC tuple kind)
                   (if (memq kind '(remove spawn))
                       #fail
                       #succeed)))))))


(define (send obj name)
  (rd obj (name ?definition)
      definition))
```

Figure 4: Implementation of a simple object in $\mathcal{TS}$, and a `send` operation that returns the definition of its argument, `name` as defined in object `obj`'s hierarchy.

```
(define (make-point a b)
   (let ((obj (make-object (global-ts))))
      (put obj ('x a))
      (put obj ('y b))
      (put obj
         ('DistfromOrig (lambda (self)
                           (sqrt (+ (sqr (send self 'x))
                                    (sqr (send self 'y)))))))
      (put obj
         ('ClosertoOrig (lambda (self p)
                           (< ((send self 'DistfromOrig) self)
                              ((send p 'DistfromOrig) self)))))
      obj))
```

Figure 5: A `point` generator.

```
(define (make-circle a b r)
   (let ((super (make-point a b))
         (obj (make-object super)))
      (put obj ('radius r))
      (put obj
         ('DistfromOrig (lambda (self)
                          (max (- ((send super 'DistfromOrig) self)
                                   (send self 'radius)) 0))))
       obj))
```

Figure 6: A `circle` generator.

specified by the `circle` (*not* the `point`) generator. The code for the `circle` generator is given in figure 6.

To create a circle C at coordinates `(3,4)` with radius `10`, we evaluate:

`(define` C `(make-circle 3 4 10))`

To determine C's distance from the origin, we evaluate:

`((send` C `'DistfromOrig)` C`)`

Because message sends are implemented via tuple space, many processes may simultaneously send messages to the same object; conversely, many processes may simultaneously respond to received messages. This property distinguishes $\mathcal{TS}$ from several other concurrent object systems[3, 26] that serialize hierarchical abstractions. In these systems, objects can process only one message at a time;

in contrast, objects in $\mathcal{TS}$ exhibit totally asynchronous behavior since they are implemented in terms of general tuple-space objects.

Furthermore, new method definitions and instance variables can be added to an object dynamically without requiring reevaluation of the inheritance tree – since object components are tuples, and there is no restriction in an object's specification that limits the addition of new tuples, processes are free to augment an object's component set even after the instance is defined. The ability to add new components to instances of objects is tantamount to a delegation-based system[18, 23]. Different instances of the same class can have different behavior based on how new instances are added and old instances are changed.

Single inheritance is realized as a simple application of tuple-space customization. Generalizing this method to handle multiple inheritance is straightforward, involving only slight modification to the matching policy to specify a total ordering over an object's superclasses.

5 Conclusions

$\mathcal{TS}$ is an attempt to increase the flexibility and expressivity of distributed data structures. First-class tuple-spaces contribute to modularity and abstraction. Permitting the programmer to customize the operational behavior of tuple-spaces makes it possible to build distributed data structures whose contents are devoid of bookkeeping clutter. Although the policy definitions given here subsume a wide range of programming paradigms, an interesting avenue of research is to develop linguistic mechanisms that permit users to generate other kinds of policy definitions dynamically that can be suitably interpreted by tuple-space operations.

First-class tuple-spaces and customization have implications on implementation as well. Because tuple-spaces are denotable objects with a concrete specification in the base language, we can apply standard optimization techniques (*e.g.,* flow- and lifetime analysis) to optimize their representation[15]. We anticipate that such analysis can be used to significantly reduce any implementation penalty incurred because of customization.

In summary, we view the design of $\mathcal{TS}$ as one step towards the realization of a highly flexible, efficient parallel programming system for symbolic computation. Work is currently underway on a parallel implementation.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] V. Ambriola, P Ciancarini, and M. Danelutto. Design and Distributed Implementation of the Parallel Logic Language Shared Prolog. In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 40–49, March 1990.

[3] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented System. In *Proceedings of the ECOOP Conf.*, pages 234–242, 1987.

[4] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. *Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[5] Nick Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3), September 1989.

[6] Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[7] Andrew Chien and W.J. Dally. Concurrent Aggreates (CA). In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–197, March 1990.

[8] K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8:1–49, 1986.

[9] William Cook and Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *OOPSLA'89 Conference Proceedings*, pages 433–444, 1989. Published as SIGPLAN Notices 24(10), October, 1989.

[10] David Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of* PARLE '89*, volume 2*, pages 20–27, 1989.

[11] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Press, Reading, Mass., 1983.

[12] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[13] Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, June 1989.

[14] Paul Hudak and Philip Wadler, editors. Report on the Functional Programming Language Haskell. Technical Report YALEU/DCS/RR-666, Yale University, Dept. of Computer Science, December 1989.

[15] Suresh Jagannathan. Optimizing Analysis for First-Class Tuple-Spaces. In *Third Workshop on Parallel Languages and Compilers*, August 1990. Forthcoming from MIT Press.

[16] Samuel Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In 15^{th} *ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[17] David Kranz, Robert Halstead, and Eric Mohr. Mul-T: A High Performance Parallel Lisp. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, pages 81–91, June 1989.

[18] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *OOPSLA'86 Conference Proceedings*, pages 214–223, 1986.

[19] Jonathan Rees and editors William Clinger. The Revised^3 Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), 1986.

[20] Ehud Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19(8):44–60, August 1986.

[21] Ehud Shapiro, editor. *Concurrent Prolog : Collected Papers.* MIT Press, 1987. Volumes 1 and 2.

[22] Brian Smith and J. des Rivières. The Implementation of Procedurally Reflective Languages. In *Proceedings of the 1984 Conference on Lisp and Functional Programming*, pages 331–347, 1984.

[23] David Ungar and Randall Smith. Self: The Power of Simplicity. In *OOPSLA'87 Conference Proceedings*, pages 227–241, 1987.

[24] United States Dept. of Defense. *Reference Manual for the ADA Programming Language*, 1982.

[25] Mitchell Wand and Daniel Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the 1986 Conference on Lisp and Functional Programming*, pages 298–307, 1986.

[26] A. Yonezawa, E Shibayama, T Takada, and Y. Honda. Object-Oriented Concurrent Programming – Modelling and Programming in an Object-Oriented Concurrent Language, ABCL/1. In *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, 1987.