Safer Tuple Spaces

Roel van der Goot[1], Jonathan Schaeffer[2], and Gregory V. Wilson[3]

[1] Erasmus University Rotterdam, The Netherlands, vandergoot@few.eur.nl
[2] University of Alberta, Canada, jonathan@cs.ualberta.ca
[3] Visible Decisions, Canada, greg@vizbiz.com

Abstract. The simplicity and elegance of the Linda programming model is based on its single, global, typeless tuple space. However, these virtues come at a cost. First, the tuple space can be an impediment to scalable high performance. Second, the "black box" nature of the tuple space makes it an inherently dangerous data structure, prone to many types of programming errors.
Blossom is a C++ version of Linda with extensions. This paper introduces some of the novelties in Blossom, as they pertain to creating "safe" tuple spaces. These new features include multiple strongly typed tuple spaces, field access patterns, tuple space access patterns, and assertions.

1 Introduction

The computing literature is replete with programming models for writing parallel programs. Many of them are research prototypes that are good enough for a few academic papers and then quietly disappear. Few models attract sustained interest in the literature, or establish a large user community. Linda [7] is one of the rare parallel programming models that has managed to survive for over a decade, in spite of the discriminating (and critical) tastes of the parallel computing community.

Linda's strength is the simplicity of its programming model. In effect, Linda offers a blackboard (the *tuple space*) that processes can read from, write to, and erase portions of. To support Linda, languages (such as C, C++, and Fortran) need only have a library implementing a few calls to access the tuple space. Thus, a simple implementation is easily built. More sophisticated versions of Linda exist, relying on compiler techniques to understand the parallelism in the application and optimize accordingly [8].

Although the uniqueness of the Linda approach excited the parallel computing community in the late 1980's, with time the novelty waned. In part this was because Linda became a commercial product (with the resulting licensing fees), and free versions suffered in performance. Nevertheless there continues to be an active Linda user community.

Although there are many advantages to the Linda model (simplicity, ease of integration in a language, ease of expressing communication and synchronization), there are two major criticisms. First and foremost, the tuple space is often seen to be a bottleneck and an impediment to high scalable performance. For many applications, these performance concerns effectively rule out Linda as a viable tool. Second, an often overlooked point is that the tuple space has some dangerous design flaws that can give rise to serious programming errors. The tuple space is a generic shared data structure that can be viewed as a black box: one can read and write to it, but an application cannot see inside it. It is legal to put *anything* into the tuple space, including semantically invalid data. Furthermore, there is no way to peek into this black box to see its internal structure and check for errors. Thus, to make Linda a more attractive parallel programming model, two things must be addressed: improved program performance, and making the model safer, in the sense that the likelihood of programming errors is reduced.

This paper introduces Blossom, an extended version of Linda for the C++ programming language, designed to address the major concerns of Linda. This paper describes some of the innovations in Blossom that influence the structure of the tuple space. By appropriately specifying tuple space(s), the probability of introducing a programming error into a parallel Blossom program is greatly reduced. Parallel programs are notoriously difficult to design, implement, test, and debug. Anything that can be done to shorten the program development cycle is welcome. Although the enhancements described in this paper address the software engineering concerns of generating correct programs, most of them also can be used to improve program performance.

Since C-Linda and Fortran-Linda made their debut, there has been a major shift in programming language trends. Strong typing and class libraries (as in C++ and Java) have increased the expressive power available to the programmer. In particular, these features and C++ templates allow us to achieve many of the capabilities of a Linda compiler using just a standard C++ compiler, without introducing new syntax or changing the semantics of the programming language.

The Linda model can easily be integrated into an object oriented programming language [15]. Linda implementations for Eiffel [13] and Java [9] require the user to explicitly create objects of class `Tuple`. In C++ Linda [5] a precompiler is required to translate new syntax into C++. Our Blossom system neither needs explicit creation of objects of class `Tuple`, nor does it use a precompiler.

Section 2 describes the Linda model and introduces some of the proposed model enhancements in the literature. Section 3 describes four innovations in Blossom. The paper is restricted to discussing the design of the tuple space, since these enhancements are currently implemented and working. Section 4 illustrates programming using Blossom. Finally, Section 5 discusses Blossom work in progress, including the performance enhancements.

2 Linda

Linda was introduced in 1985. Although the Linda model was not in its final form, it contained the idea of a global memory accessible by multiple processes. The global memory contains a collection of data records, called *tuples*, and it is accordingly called the *tuple space*. Tuple space data is organized as an associative memory, meaning that data is retrieved by its value(s), not by an index.

Linda provides six simple operations to access the tuple space. Operation `in` gets a tuple from the tuple space (*in* the program), `out` puts a tuple (*out* of the program) into the tuple space, and `rd` replicates a tuple. The `rd` operation is semantically equivalent to an `in` immediately followed by an `out` (combined in one atomic action).

The following example illustrates how a program might manipulate the tuple space. An `out("todo", 1);` puts a tuple with two fields into the tuple space; the first field is a string with value `"todo"` and the second field is an integer with value `1`. A variable with value `1` as the second field would have the same result.

An `in("todo", 1);` will get the same tuple out of the tuple space, if it is present. If there is no tuple with values equivalent to all the arguments, the operation will block until another process puts a matching tuple in the tuple space. Wild cards can be used to enhance the capabilities of the operations that read tuples. For example,

```
int i;
in("todo", ?i);
```

matches any tuple with two fields, the first being a string with value `"todo"` and the second an integer (with any value). Such a wild card field (`?i`) is called a *formal* parameter as opposed to an *actual* parameter (a value field). Tuples that contain formal fields are sometimes referred to as *anti-tuples*.

Linda provides an `eval` operation for spawning new processes. For example, `eval("todo", sqr(5));` will spawn two processes, each evaluating one of the arguments in parallel with the spawning process. As soon as all the arguments of the `eval` are evaluated, the resulting tuple is put into tuple space (the tuple `("todo", 25)` in this case).

The operations `inp` and `rdp` are nonblocking versions of `in` and `rd`, respectively. They query the tuple space for a match and return a boolean indicating whether they succeeded in that. If no match is found the operations will not wait (i.e., block) for another process to insert a matching tuple.

Because the model is so simple (and elegant), it is easy to find fault with it. Consequently, numerous extensions to the model have been proposed in the literature. Some of the more interesting ones include: having multiple tuple spaces [11], more powerful tuple space operations such as `collect` [4] and `copy-collect` [16], specifying access patterns on tuples [6], persistent tuple spaces [3], fault-tolerant tuple spaces [1], and open Linda [14]. In this paper we will use the extension of multiple tuple spaces.

This paper discusses Blossom, a C++-based implementation of Linda with extensions. Since we are using only the C++ compiler and not a Linda compiler, the syntax of Blossom differs slightly from that of Linda. In Blossom tuple spaces become objects, and the operations on tuple spaces become member functions. The original Linda operations are relative to the program; the Blossom member functions are relative to the tuple space. Hence new names for the operations had to be chosen: `out` becomes `put`, `in` becomes `get`, `rd` becomes `copy`, `inp` becomes `get_nb`, and `rdp` becomes `copy_nb` (where `nb` indicates the call is nonblocking). The `eval` operation is handled as a special case of `put`:

```
eval(5, sqr(5));
```

becomes

```
// "ts" is the name of a user-defined tuple space
ts.put(5, eval(sqr)(5));
```

indicating that only one process is created to evaluate the square of 5.

Formal parameters (`?`) are handled differently as well:

```
int i;
in(5, ?i);
```

becomes

```
Arg<int> i;
ts.get(5, i.var());
```

where `Arg<`*`type`*`>` is a Blossom class that enhances the *`type`* to include additional methods. The `var()` method allows the variable to be used as a wild card.

Blossom's tuple spaces are first class objects, which means that it is possible to have tuple spaces of tuple spaces.

3 Enhancements

The tuple space is a typeless black box. The user can put data of any type in it, and attempt to extract data of any type from it. But the wealth of programming language design experience suggests that this is a bad idea. The trend in computing today is towards strong typing. We have seen the evolution of BCPL to C to C++, motivated in part by the benefits of strong typing. These benefits are twofold: catching more errors at compile-time, and preventing some classes of run-time errors. Given the inherently difficult nature of parallel programming, any language enhancements that reduce the probability of error are welcome.

In this section, four new enhancements to the basic Linda model are presented. All of them have to do with either preventing or detecting parallel program errors at compile- or run-time. This is achieved by changing the definition of the tuple space: it is given structure, behavior, and is made accessible (see Figure 1). The user provides annotations that give the compiler and run-time system information as to the expected behavior of the program, and forces it to meet those constraints.

Blossom is a C++ class library. Unlike C-Linda and Fortran-Linda, it does not add new syntax, alter the semantics, or extend the host language, each of which tends to result in programmer confusion [17]. C++ is a modern programming language with a rich set of features. This enables Blossom to achieve many

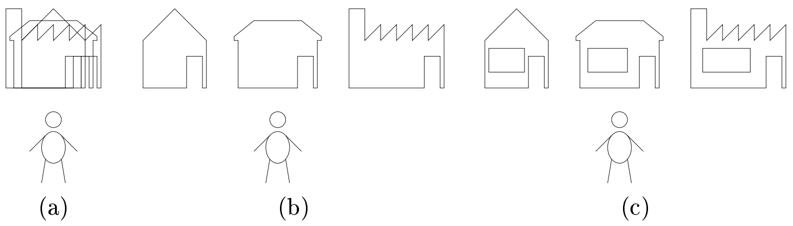

Fig. 1. Differences between tuple spaces; (a) Original Linda tuple space: It is a single collection of tuples of arbitrary structure. This implies that the user can try to insert/remove semantically incorrect tuples. The user cannot look inside the tuple space. (b) Strongly typed tuple space: Each tuple space can contain only one type of tuple. Hence a program cannot inadvertently insert/remove a wrongly typed tuple. (c) Strongly typed tuple space with assertions: It provides the user the additional benefit of being able to peek inside the tuple space.

of its objectives (including preventing programming errors) using only C++ syntax, without the need for a separate precompiler. To achieve the same benefits of a Linda precompiler (including error checking and run-time optimizations) , the user has to specify a small amount of additional detail in their program. This is reflected in the Blossom examples given later. They are (slightly) more verbose than their pure-Linda counterparts.

One issue not addressed in this paper is how to handle run-time errors generated by the new Blossom enhancements. Since we are using C++, a natural solution is to throw an exception whenever a run-time error occurs. The user can (at their discretion) catch the exception and handle it. Although this sounds good, it does have its problems. For example, do child processes throw exceptions in their parents? What if a process' parent is no longer around, but its grandparent is? Right now, all run-time errors result in aborting the application.

3.1 Strongly Typed Tuple Spaces

Linda's single tuple space can contain tuples of any type. Although this sounds conceptually simple, it has the disadvantage that it is easy to inadvertently introduce a bug in a program. Accidentally omitting a tuple field, introducing an additional field, changing a field, or reversing two fields is perfectly legal, yet may result in a semantically incorrect program. A sophisticated precompiler may be able to detect some but not all of these errors.

By having multiple tuple spaces, each strongly typed, these problems cannot occur. Each tuple space is created with a specification that gives the number, order, and type of each field of the tuples legally allowed (much like the parameters of a subroutine call). With this specification, many commonly occurring errors can be caught at compile time without the need for a Linda precompiler.

The advantages of strong typing can be summarized as follows:

1. Eliminate Errors. A strongly typed tuple space can eliminate many common errors.
2. Performance. Although this paper emphasizes the software engineering aspects of Blossom, it is important to realize that many of these enhancements can also be used to improve program performance. In effect, the typing partitions the tuple space into multiple disjoint address spaces. Hence, given a tuple of a particular type, the system knows exactly in what address space to look for the tuple.

The following example illustrates having multiple, strongly-typed tuple spaces in Blossom. The `TupleSpace` declarations each create new named tuple spaces, and specify the type of the tuples that are allowed inside them.

```
TupleSpace<Tuple<int, int> > square;
TupleSpace<Tuple<int, float> > sqrroot;

// Put values in the tuple spaces
for (int i = 1; i < MAX; i++) {
   square.put(i, i*i);
   sqrroot.put(i, sqrt(i));
}

// Take values out of the tuple spaces
Arg<float> answer;
sqrroot.get(4, answer.var()); // Succeeds
square.get(1, 1);             // Succeeds
square.get(2, 2);             // Blocks: no such tuple available
square.get("one", 1);         // Compile-time type error
square.get(1, 1.0);           // Compile-time type error
square.get(1.0, 1);           // Compile-time type error
```

3.2 Field Access Patterns

The philosophy behind typing a tuple space can be carried even further. A common programming scenario in Linda is to extract tuples using one of the fields as a discriminator. For example, one might have a producer-consumer computation where the tuples produced have to be processed in order. Every tuple contains a unique number which is more or less a time stamp. The smaller the number the older the tuple. The consumer `get`s the next tuple by increasing the time stamp field. In this case, this field is not only an integer, but it is always used as a *value*, not a wild card variable.

Every field in a tuple space declaration contains a declaration as to whether it is legal to use a wild card variable to retrieve a value or not. In other words, a field is designated as either `Actual` or `Any`. `Actual` indicates that a tuple field can only be retrieved by specifying a value, while `Any` extends this to include wild card variables.

This additional piece of information offers the user several advantages:

1. Eliminate Errors. By providing additional information about the intended behavior of the tuple space, the compiler can check whether the programmer always satisfies the specified requirements.
2. Performance. Specific fields can be used to determine (by hashing) on which machine to store or look for a tuple. Another method to distribute the tuple space is by broadcasting every change of the tuple space to all Linda processes. Hashing has the advantage that broadcasts are not necessary to distribute the tuple space. Instead a single communication is enough to put a tuple in a remote tuple space and only two communications are needed to get or to copy a tuple from a remote tuple space (the first sends the `get` or `copy` request to the remote tuple space; the second one when a matching tuple is found and how the formals are instantiated). An evenly distributed tuple space results in less tuples to be searched and removes possible bottlenecks of a big tuple space on one machine.

The following example illustrates field access patterns:

```
TupleSpace<Tuple<Actual<int>, Any<int> > > square;
Arg<int> sq;
for (int i = 1; i < MAX; i++)
   square.put(i, i*i);
square.get(1, sq.var());  // Succeeds
square.get(2, 2);         // Legal, but blocks
square.get(sq.var(), 4);  // Compile-time error
```

3.3 Tuple Space Access Patterns

The object-oriented community has recently embraced the concept of *design patterns* [10]: generic, frequently occurring programming paradigms. Similar ideas have played a prominent role in parallel computing, with common parallel structures such as master-slave and pipelines. Recent studies have shown that there are commonly occurring access patterns[1] on objects [2]. As part of program design, the user knows the intended access patterns on their objects. The first two columns of Table 1 gives these access patterns and their description, as defined in the Munin system [2].

The access patterns on objects are easily translated to access patterns on tuples [6]. In Blossom the access patterns are applied to tuple spaces instead of tuples. There are several reasons for doing this. First, in a good design, every tuple space has a unique purpose. All members of that tuple space are there to fulfill the design goals of the tuple space and should be handled similarly. Tuples with different access patterns in one tuple space are an indication of a poor design. Second, access patterns can be used to influence the (hidden) implementation of the tuple space, affecting program performance. For example, the choice of design pattern can effect the placement of the tuple space, the distribution of the tuple space, and whether data should be replicated.

[1] The original name is type-specific coherence mechanisms.

If a tuple space were annotated with an access pattern specification, then that might impose a constraint on how the tuple space is accessed at run-time (as shown in the third column of Table 1). Note that not all access patterns result in constraints on the tuple space. Read-mostly, write-many, and general read-write all allow an arbitrary combination of reads/writes to the tuple space. The only difference is the relative frequency with which the reads/writes occur. Hence these are too general to allow constraints to be imposed.

Table 1. Access patterns for tuple spaces

Access Pattern	Description	Restrictions
write-once	written during initialization, afterwards only read	no `put`s after a `get` or `copy`
private	local accesses only	only one process is allowed access
write-many	frequently modified	—
result	not needed until all data is collected	—
synchronization	semaphores	no `copy` or `copy_nb` allowed
migratory	accessed by a single process at a time	—
producer-consumer	written by one process, consumed by other processes	only one process is allowed to `put`
read-mostly	read far more often read than written	—
general read-write	general access pattern	—

There are a number of important advantages for having the user declare the access pattern of a tuple space:

1. Eliminate Errors. If the access pattern of a tuple space does not match the access pattern in the user's code, then warnings or errors can be given at compile- and run-time. For example, if the access pattern is producer-consumer and there are two or more processes `put`ting tuples in the tuple space, the program can detect this at run-time. (Not at compile-time because we do not use a precompiler.) If the program has to run efficiently this run-time check can be turned off.
2. Performance. A programmer usually knows the intended use of a tuple space. Depending on these intentions, a distribution strategy could be selected by the user. For example a producer-consumer tuple space is best located in the data space of the producer process. It does not make sense that the producer first distributes the tuples over several processes, and next the consumer gathers the tuples from all these processes. The former approach

needs less communications to ship the data to the place it is used (consumed) and is thus faster.

3. Awareness of access patterns results in better designs. A good design never uses an object for more than one reason. This general rule applies to tuple spaces too. Access patterns can help the programmer distinguish different functionalities more easily.

It is important to realize that the access pattern attribute only changes the semantics of a tuple space if a restriction is encountered; if this is not the case access pattern attributes only change the implementation and not the semantics.

The following example illustrates the use of access patterns:

```
TupleSpace<Tuple<Any<int> >, ProducerConsumer> buffer;

// PRODUCER...
while(1) {
   int number = ...;
   buffer.put(number);       // Succeeds
}

// CONSUMER...
while(1) {
   Arg<int> number;
   buffer.get(number.var()); // Succeeds
   // ...
   buffer.put(16);           // Run-time error
}
```

3.4 Tuple Space Assertions

The simplicity of the Linda programming model comes, in part, from the capabilities of the tuple space; it is a black box with all the structure and implementation details hidden from the user. However, when looking for a bug, you want to be able to look inside the black box. This cannot be done in Linda, unless you extract each of the tuples one at a time and then put them back.

Tuple space assertions allow the user to peek into the tuple space. A problem with assertions in parallel programming is that several processes have access to the same tuple space. This means that if you put a tuple in the tuple space and then, for example, immediately assert that the number of tuples in the tuple space is greater than zero, the assertion can fail because another process removed the tuple. So either you have to make the assertion general (i.e., it takes other processes into account) or the assertion should form an atomic action with the other Linda operations. We chose the latter kind of assertion, because it better meets the needs of concurrent debugging.

Assertions act on an entire tuple space, which may be very inefficient if the tuple space is distributed. This is mainly because the whole tuple space should be locked during the evaluation of an assertion; no other processes are allowed to change its state. Assertions on tuple spaces should only be used to debug a program. However after fixing the bug you want to be able to recompile the program without removing the assertions again. Inclusion of another library (a compile-time option) can achieve this.

If the user wants to define an assertion on a tuple space, he has to derive a new class from class `Assertion`. In this class he can define two methods, `foreach()` and `exit()`. The constructor call initializes the object, the `foreach()` call is performed for every tuple in the tuple space, and the `exit()` call should return a boolean, indicating whether the assertion succeeded (`true`) or failed (`false`). If the `exit()` call returns false the program terminates. Blossom provides the (hidden) code to ensure that `foreach()` and `exit()` are called correctly.

Assertions are illustrated in the next section.

4 Example

The following example illustrates the usage of the concepts introduced in this paper. A (distributed) sieve of Eratosthenes calculates all primes smaller than a certain upper bound `LIMIT`. The function `is_prime` decides whether a number (the first argument) is a prime or not by trying to divide the number by all smaller primes (passed in the tuple space in the second argument). The strongly typed tuple space checks at compile-time that all accesses (here `put` and `copy`) have a tuple of the correct type (`put` only allows actual fields and `copy` only allows the specified fields). The `SizeAssertion` checks that the number of tuples in the tuple space is correct after all the primes are calculated.

```
#include <iostream.h>
#include "ts.h"

typedef TupleSpace<Actual<int>, Any<bool> > PrimesTS;

const int LIMIT = 5000;

class SizeAssertion: public Assertion {
   int size;
   int expected_size;
public:
   SizeAssertion(int s): expected_size(s), size(0) {}
   ~SizeAssertion() {}
   void foreach(int, bool) {
      size++;
   }
   bool exit() {
      return size == expected_size;
   }
};

bool is_prime(int number, PrimesTS primes) {
   int limit = sqrt(number);
   for (int i = 2; i < limit; i++) {
      Arg<bool> prime;
      primes.copy(i, prime.var());
      if (prime && (number%i == 0))
         return false;
   }
   return true;
}

void main() {
   PrimesTS primes;
   Arg<bool> prime;
   for (int i = 2; i < LIMIT; i++) {
      primes.copy(i, prime.var());
      if (prime)
         cout << i << endl;
   }
}
```

5 Future Work

The four enhancements discussed in this paper have been implemented and are working. All four help the user eliminate and/or detect common parallel programming errors. By making the tuple space safer, the user will spend less time debugging and they can invest more time in the design of their program.

Regrettably, it is impossible for us to quantify the benefits of the safe tuple spaces of Blossom. There is no easy metric, such as speedup, to compare against. A fair evaluation would require users to implement solutions using C-Linda and Blossom, and then compare the time it takes to correctly implement their solutions. Although we have experience performing these type of experiments [17], it is too early in the life-cycle of Blossom to go to this effort. Once we have completed the full Blossom implementation, we will be very interested in quantifying the influence of safer tuple spaces on the program design and implementation time.

Blossom is an evolving system. The following performance enhancements are in various stages of completion:

Futures: Arguments to `get` and `copy` can be futures [12]. Futures allow asynchronous versions of the aforementioned operations.

Get-update-put operation: The equivalent of a get-update-put operation in Linda results in three communications: send `get`, receive result, (update value,) and put update. By combining the operations, a reduction of one communication can be achieved: send get, (update value and put,) and receive result.

Reduction operations: Without a reduction operation, reducing a tuple space requires the user to get each tuple and perform the reduction, resulting in a potentially large amount of communication. A reduction operator performs the reduction according to the *owner computes* paradigm. There is a strong similarity between tuple space assertions and reduction operations: both require access to the entire tuple space in an efficient manner. The user can specify the reduction operator and let Blossom do the rest.

Conditionals: Sending a conditional to the tuple space, indicating what tuples you are interested in, can reduce communication and improve parallelism. Blossom can apply the conditional to all tuples in a tuple space and return those that match. The alternative in Linda is to get tuples until you find one that meets your conditions, and then put back all unneeded tuples.

Tuple space attributes: Additional information can be associated with a tuple space. *Ordered vs. Unordered*: Tuple spaces in which the tuples are ordered (sorted) are better for database applications (faster searching), but unordered tuple spaces generally have better performance. *Fair vs. Unfair*: In a fair tuple space every matching tuple has an equal opportunity of being selected. Unfair tuple spaces have biases but may have more efficient implementations. *Persistence*: Persistent tuple spaces are kept in files, which means that they are still present after a program terminates. Databases are in general persistent. *Size*: Allowing the user to specify the maximum size of a tuple space allows Blossom to more efficiently organize the data.

We expect to have complete working system by the end of the year.

Acknowledgments

We would like to thank *Nederlandse Organisatie voor Wetenschappelijk Onderzoek* (NWO), National Sciences & Engineering Research Council of Canada (NSERC), and IBM Toronto Labs for their financial assistance. Furthermore, we thank Josée Lajoie (IBM) for her patience answering numerous ANSI C++ questions, Bill O'Farrell (IBM) for sharing some of his knowledge on C++, Steve MacDonald (University of Alberta), and Arie de Bruin (Erasmus University Rotterdam) for their comments on the article.

References

1. Bakken, D. E., and Schlichting, R. D. Supporting Fault-Tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel and Distributed Systems 6*, 3 (1995), 287–302.
2. Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)* (1990), pp. 168–176.

3. Brown, T., Jeong, K., Li, B., Talla, S., Wyckoff, P., and Shasha, D. *PLinda User Manual.* http://merv.cs.nyu.edu:8001/~binli/plinda/manual.ps, 1997.
4. Butcher, P., Wood, A., and Atkins, M. Global Synchronisation in Linda. *Concurrency: Practice and Experience 6*, 6 (Sept. 1994), 505–516.
5. Callsen, C. J., Cheng, I., and Hagen, P. L. The AUC C++Linda System. In *Linda-Like Systems and Their Implementation*, G. Wilson, Ed. Tech. Rep., EPCC TR91–13, Edinburgh Parallel Computing Centre, June 1991, ch. 4, pp. 39–73.
6. Carreira, J., Silva, L., and Silva, J. G. On the design of Eileen: a Linda-like library for MPI. In *Proceedings of the 1994 Scalable Parallel Libraries Conference* (1995), pp. 175–184.
7. Carriero, N., and Gelernter, D. *How To Write Parallel Programs: A First Course.* MIT Press, Cambridge, MA, 1990.
8. Carriero, N., and Gelernter, D. *Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler.* Research Monographs in Parallel and Distributed Computing. Pitman, London, 1990, ch. 7, pp. 114–125.
9. Ciancarini, P., and Rossi, D. Jada: Coordination and Communication for Java agents. In *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds., vol. 1222 of *Lecture Notes in Computer Science.* Springer Verlag, 1997, pp. 213–228.
10. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns.* Addison-Wesley, 1995.
11. Gelernter, D. Multiple Tuple Spaces in Linda. In *Proceedings PARLE'89: Parallel Architectures and Languages Europe* (June 1989), pp. 20–27.
12. Halstead, A. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (1985), 501–538.
13. Jellinghaus, R. Eiffel Linda: An Object-Oriented Linda Dialect. *ACM SIGPLAN Notices 25*, 12 (Dec. 1990), 70–84.
14. Kielmann, T. Designing a Coordination Model for Open Systems. In *Coordination Languages and Models, First International Conference COORDINATION '96* (1996), P. Ciancarini and C. Hankin, Eds., pp. 267–284.
15. Matsuoka, S., and Kawai, S. Using Tuple-Space Communication in Distributed Object-Oriented Architectures. *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 23*, 11 (1988), 276–284.
16. Rowstron, A., and Wood, A. Solving the Linda Multiple `rd` Problem. In *Coordination Languages and Models, First International Conference COORDINATION '96* (1996), P. Ciancarini and C. Hankin, Eds., pp. 357–367.
17. Szafron, D., and Schaeffer, J. An Experiment to Measure the Usability of Parallel Programming Systems. *Concurrency: Practice and Experience 8*, 2 (1996), 147–166.