# Bonita: A set of tuple space primitives for distributed coordination

A. I. T. Rowstron

Department of Computer Science
University of York
York, YO1 5DD UK
ant@cs.york.ac.uk

A. M. Wood

Department of Computer Science
University of York
York, YO1 5DD UK
wood@cs.york.ac.uk

**Abstract**

*In the last few years the use of distributed structured shared memory paradigms for coordination between parallel processes has become common. One of the most well known implementations of this paradigm is the shared tuple space model (as used in Linda). In this paper we describe a new set of primitives for fully distributed coordination of processes and agents using tuple spaces, called the* Bonita *primitives. The Linda primitives provide synchronous access to tuple spaces, whereas the* Bonita *primitives provide asynchronous access to tuple spaces. The proposed primitives are able to mimic the Linda primitives, therefore providing the ease of use and expressibility of Linda together with a number of advantages for the coordination of agents or processes in distributed environments. The primitives allow user processes to perform computation concurrently with tuple space accesses, and provide new coordination constructs which lead to more efficient programs.*

*In this paper we present the (informal) semantics of the* Bonita *primitives, a description of how the Linda primitives can be modelled using them and a demonstration of the advantages of the* Bonita *primitives over the Linda primitives.*

## 1 Introduction

Over the last ten years the power of the tuple space model for coordinating parallel programs has been demonstrated in such systems as Linda. The first Linda implementations were developed for parallel machines, however the potential for "network based computing" using the Linda primitives and the tuple space model for coordination of processes executing over networks of workstations was soon realised.

As tuple space based systems are created for widely distributed computing resources it is necessary to consider whether the current Linda primitives are sufficient or as effective as they could be given the flexible nature of the underlying tuple space model. In this paper we present an argument, backed with examples, that a new set of primitives for tuple space access should be adopted for use in distributed environments. These primitives are called the Bonita primitives.

The tuple space access operations as embodied in the Linda primitives are synchronous. This enabled the Linda primitives to deliver a clear and simple interface to tuple spaces at the cost of loosing potential parallelism between tuple space access and user computation. The primitives for tuple space access described in this paper provide asynchronous access to tuple spaces. This enables processes accessing the tuple spaces to perform computation concurrently with the tuple space access. The new primitives also allow the creation of new styles of programming, which are more efficient than the equivalent using the Linda primitives. The proposed primitives allow the main Linda primitives to be mimicked, therefore providing the mechanisms to perform synchronous tuple space access (efficiently).

Other relevant work has been completed on the parallelisation of tuple space access costs and user computation by Landry et al.[12]. This work focused on the compile time analysis of Linda primitives and user computation. The idea was that at compile time `in` and `rd` are split into two sections, and these two sections are moved as far apart as possible. The first section is an initialise and the second section a receive. Such compile time analysis can be performed in both open and closed implementations. However, there is a fundamental flaw with their approach, due to the non-deterministic behaviour of the Linda model. Within a Linda program often the length of computation ensures that sometimes competition for specific tuples is avoided, or follows a "natural" course. If the requests are automatically moved then this natural spreading of requests is potentially removed. This leads to a possibility that the "optimised" program will in fact take *longer* than the original. For more details see Rowstron[13].

Initially, a overview of the Linda primitives is presented, followed by a more detailed justification for the new primitives. The informal semantics of the new primitives and how they can be used to produce more efficient styles of programming is then described. Finally a brief description of the current prototype implementation is given.

## 2 The Linda primitives

The Linda model is now well known and a detailed description can be found in [4]. The main primitives are:

**out(***ts, tuple***)** This places the tuple (*tuple*) into a tuple space (*ts*).

**in(***ts, template***)** This removes a tuple from a tuple space. The tuple removed is associatively matched using the template and the tuple is returned to the calling process. If no matching tuple exists then the calling process is blocked until one becomes available.

**rd(***ts, template***)** This primitive is identical to *in* except the matched tuple is not removed from the tuple space, so a copy is returned to the calling process.

**eval(***ts, active-tuple***)** The *active-tuple* contains one or more functions, which are evaluated in parallel with each other and the calling process. When all the functions have terminated a tuple is placed into the tuple space with the results of the functions as its elements.

Some Linda systems support two other primitives, `inp` and `rdp`. These are non-blocking versions of `in` and `rd`. Instead of blocking they return a value to indicate no tuple was found. For a number of (semantic) reasons many systems do not support them.

Over the last ten years there have been a number of proposed extensions to both the Linda primitives and the underlying tuple space model. Of these the most important is the extension of the tuple space model with multiple tuple spaces. Schemes based on hierarchies of tuple spaces have been suggested[7, 8] as well as one with a mixture of flat and hierarchical tuple spaces[9]. The work described here is not affected by the exact relationship of the multiple tuple spaces. In a distributed environment it is important to have multiple tuple spaces, however no assumptions are made about their relationship with each other.

There are two new primitives which have been proposed that are of particular interest; `collect` and `copy-collect`. The semantics of these primitives are described below:

**collect(***ts1, ts2, template***)** The `collect`[3] primitive *moves* all available tuples from `ts1` that match `template` into `ts2`, returning a count of the number of tuples moved.

**copy-collect(***ts1, ts2, template***)** The `copy-collect`[17] primitive is similar to `collect` except it *copies* all available tuples that match the given `template` in the source tuple space (`ts1`) to the destination tuple space (`ts2`). As with `collect` it returns a *count* of the number of tuples copied.

The motivation for the addition of the `collect` primitive is outlined in Butcher et al.[3]. The motivation for the `copy-collect` primitive is a particular operation that is difficult to perform using the standard Linda model. This operations is referred to as the multiple `rd` problem and a detailed discussion of it is presented in Rowstron et al.[17].

For this paper we wish to make a clear distinction between the Linda primitives and the underlying tuple space model. When we refer to the tuple space model we are referring to the basic concept of shared tuple spaces of tuples (multi sets). The access to tuple spaces is purely by a process of associative matching. When we refer to the Linda primitives we are referring to a particular set of primitives that provide the interface to the tuple spaces.

Most Linda implementations fall into one of two categories, *closed* and *open*. Closed implementations use compile time analysis to create very efficient implementations[1]. However, closed implementations normally do not allow processes to join and leave at will. All processes that wish to communicate via a shared tuple space must be available at compile time. Open implementations allow processes to join and leave quite freely and provide persistent tuple spaces, rendering most of the compile time analysis useless. For distributed environments open implementations are normally the most useful.

## 3 The justification for a new set of primitives

The motivation and justification for the new primitives comes from extensive experience in the implementation of open Linda systems[14, 6, 16] and their use in distributed environments. The other motivating factor is the current trend within the Linda research community to alter the current Linda primitives to fulfill the requirements of distributed coordination[2, 11] in a way which appears still *too* restrictive for distributed computing.

The observations made during the profiling of current "open" Linda implementations led to the conclusion that time costs associated with Linda primitives were not due to bottlenecks in the implementations but quite simply the time taken to send and receive messages over networks. The second observation is the problem of controlling how long a process performing *any* Linda operation should block before the operation is aborted. In order to overcome this it has been suggested that *timeouts* should be added to the primitives. The problems of such an approach are shown in Section 6.

### 3.1 Time costs involved in performing a Linda primitive

Let us consider the primitives `in` and `rd`. In the informal semantics of Linda these primitives are described as blocking only if a tuple is not available. This implies that a process which uses either of these primitives should block only if the required tuple is not available when the process requests it. However in reality, within any practical Linda system a process will always block even if the required tuples are available because of the overheads associated with finding the matching tuple. Regardless of how an open distributed Linda implementation works it is possible to provide a set of time cost categorisations which will represent the time that the primitive is blocked from the *user process's* point of view. These time cost categorisations are:

$$T_{Pack} + T_{SendRequest} + T_{Queue} + T_{Process} + \\ T_{Block} + T_{SendReply} + T_{Unpack}$$

where

$T_{Pack}$ is the time taken for the information required for a tuple to be retrieved to be packed into a message by the user process for passing to the control system[1] and the initiation of the message transmission;

$T_{SendRequest}$ is the time taken for the message to pass through the communication channels and reach the control system;

$T_{Queue}$ is the time that the message has to wait before it is serviced, when it has reached the control system;

$T_{Process}$ is the time taken for the control system to process the message it receives, find a suitable tuple, pack the message for return to the user process, and perform the initiation of the message transmission back to the user process;

$T_{Block}$ is the time that the primitive blocks because no matching tuples are available;

$T_{SendReply}$ is the time taken for the message to pass through the communication channels to reach the user process which requires it; and

$T_{Unpack}$ is the time that the user process takes to unpack and interpret the message being returned from the control system.

When considering the predicate versions of the primitives (`inp` and `rdp`), and the `collect` and `copy-collect` primitives the time $T_{Block}$ will always be zero, because these primitives never block waiting for tuples. For an `out` the time cost will just be $T_{Pack}$. Depending on specific implementations some of the categories may overlap, however, in general these categories represent the time costs of performing a Linda operation.

## 4 A new set of primitives

There are many properties of the tuple space model which makes it a good model for distributed systems. It is asynchronous and allows communicating processes to be both spatially and temporally separated. This is very important for distributed systems. We are proposing a new set of primitives (called the Bonita primitives) which use the tuple space model (with multiple tuple spaces) to allow a programmer to parallelise the access of a tuple space with computation, thus minimising many of the time costs associated with tuple space access outlined in the previous

[1]This stores and manages the tuples – in many implementations the control system is distributed.

section. The primitives provide a mechanism for placing tuples in a tuple space, retrieving them from tuple spaces and the bulk movement of tuples between tuple spaces. These primitives provide a better interface to the tuple spaces in *distributed environments* than the Linda primitives. Because these primitives are only an interface with the tuple space model they use the same concepts of tuple spaces, tuples and templates as the Linda primitives. Furthermore the same tuple and template matching is used: a tuple is matched by a template, if the tuple has the same cardinality as the template, if each of the fields in the tuple has the same type as the same field in the template, and if an actual is specified in the template it exactly matches with the same field in the tuple[2].

The BONITA primitives may be informally described as follows:

### rqid = dispatch(ts, tuple | [template, destructive | nondestructive])

This is an overloaded primitive which controls all of the accesses to a tuple space which require a tuple to be either placed in a tuple space or removed from a tuple space. The tuple space to be used is `ts`. If a `tuple` is specified then this tuple is placed in the tuple space. If a `template` is specified then this indicates that a tuple is to be retrieved from the specified tuple space. If this is the case then an extra field is used to indicate if the tuple retrieved should be removed (`destructive`) or not removed(`nondestructive`) from the tuple space `ts`. This primitive is non-blocking and returns a *request identifier* (**rqid**) which is subsequently used with other primitives to retrieve the matched tuple.

### rqid = dispatch_bulk(ts1, ts2, template, destructive | nondestructive)

This requests the movement of tuples between tuple spaces. The source tuple space is `ts1` and the destination tuple space is `ts2` and the tuples are either moved (`destructive`) or copied (`nondestructive`). The number of tuples moved or copied depends on the stability of the tuple space. If the tuple space is stable (there are no destructive operations being performed in parallel with this primitive) then all the available tuples are copied or moved. This primitive is non-blocking and returns a *request identifier* (**rqid**) which is subsequently used with other primitives to get a count of the number of tuples moved or copied.

[2]We recognise that there are many proposals for the extension of the matching, some of which may be more suited to a distributed domain. However, the matching algorithm used *does not* effect the proposed primitives, just the tuples they retrieve.

### arrived(rqid)

This detects if a tuple or result associated with an **rqid** is available. The primitive is *non-blocking* and either returns true or false to indicate whether the tuple has arrived. If an invalid **rqid** is used then the primitive returns false.

### obtain(rqid)

This is a *blocking* primitive which waits for the tuple or result associated with **rqid** to arrive. When the result becomes available then it is returned.

The exact syntax of each of the primitives depends upon the host language being used. For example, the syntax of the `obtain` primitive may include variables to be used to store the information returned. Also the primitives may return error values if an invalid **rqid** is used. In the C-BONITA version we assume that the template provides a number of variables into which the returned tuple fields are placed *when an* `obtain` *is performed* for that tuple. When BONITA is embedded into other languages different approaches may be taken, for example if ISETL was used as the host language[5] which supports tuples as first class objects the `obtain` primitive may return a tuple or an integer.

It should be noted that the non-deterministic nature of the tuple space model is preserved. If there are many tuples in a tuple space that match a template then the choice is non-deterministic, and if there are many processes competing for the same tuple which process gets the tuple is non-deterministic.

There is an extra property of tuple spaces that the BONITA primitives require, and that is one of *tuple insertion ordering.* When a *single* process performs several `dispatch` primitives then the `dispatch` primitive guarantees that tuples appear in the tuple spaces in the same order as the process produces them. The guarantee is enforced across tuple spaces. It should be noted that the order in which the tuples are *removed* from the tuple space is *not* dependent on the order in which they are inserted. This does not produce an insertion dependency across multiple processes. Each process must ensure that the tuples it produces are inserted into all tuple spaces in the order they are produced. The process inserting the tuples is *not* affected by the order in which other processes insert tuples.

There is no synchronisation between the processes inserting the tuples unless the programmer writing the user processes causes the processes to synchronise using tuples. This is important in order to ensure that the behaviour of the `bulk_dispatch` primitive is correct. If a process produces a number of tuples and then produces a marker tuple which indicates that these tuples have been produced, another process checking the marker tuple must be able to guarantee getting all the tuples using a bulk primitive. For more information refer to the *out ordering* section in Douglas et al.[6], the description of `copy-collect` in Rowstron et al.[17, 15] and the description of the "predicate operation forms: `inp` and `rdp`" in the SCA C-Linda user manual[1].

The Bonita primitives make no reference to process creation. Currently, the creation of processes in a distributed environment is viewed as something that the underlying operating systems or the system in which the Bonita primitives are embedded should manage. However, in the future the addition of a primitive (or primitives) to manage the spawning of processes may be added.

### 4.1 Time costs in performing a Bonita primitive

The time costs of performing the Bonita primitives are now considered. The same time cost definitions as used in Section 3.1 with an extra time cost, $T_{Check}$ which represents the time it takes to check locally if a required message has arrived. This is required because there is the need to perform a certain amount of arbitration of messages from the control system, since it is possible to have several tuples requested, and hence several tuples waiting locally. With the Linda primitives once a tuple has been requested, by any primitive, the next message received *must* be the requested tuple. Each of the Bonita primitives have different time costs, relative to the initiating process, associated with them:

**dispatch and dispatch_bulk** The time cost associated with these primitives is:

$$T_{Pack}$$

Both these primitives create a message and then dispatch it to the control system.

**arrived** The time cost associated with this primitive is simply:

$$T_{Check}$$

**obtain** The time cost associated with this primitive is:

$$T_{Check} + T_{Block} + T_{Unpack}$$

If the primitives are being used to mimic the Linda primitives (see Section 5) then $T_{Block}$ will represent not only the "blocked" time but also the time costs $T_{SendRequest} + T_{Queue} + T_{Process} + T_{SendReply}$ as specified for the Linda primitives. However, the ability to split the request for a tuple and the process of decoding it allows the time costs $T_{SendRequest} + T_{Queue} + T_{Process} + T_{SendReply}$ to be performed concurrently with user computation. (See Section 6).

The time costs for the `arrived` and `obtain` primitives may alter slightly in some circumstances. For example, `arrived` may need to unpack messages in order to check if it is the required one. If this is the case, the time cost $T_{Unpack}$ would not be associated with `obtain` but rather with `arrived` *if* `arrived` was used to check to see if the result was available.

## 5 The Linda primitives using Bonita primitives

The Bonita primitives have been designed to allow the basic Linda primitives of `out`, `rd` and `in` to be emulated. This means that the functionality of the Linda primitives is preserved within the Bonita primitives. Figure 1 shows how the Bonita primitives can be used to create an `in`. If the `destructive` tag was changed to `nondestructive` then the section of code would mimic a `rd`.

| C-Bonita |
|---|
| int id, x; |
| id = dispatch(ts, ?x, destructive); |
| obtain(id); |

Figure 1: A Linda style: `in(ts, ?x)`.

Figure 2 demonstrates the overloading of the `dispatch` primitive so that it mimics an `out`.

| C-Bonita |
|---|
| dispatch(ts, 10); |

Figure 2: A Linda style: `out(ts, 10)`.

The Bonita primitives do *not* provide a mechanism for creating a construct identical to an `inp` or `rdp`.

The use of the `arrived` primitive provides a mechanism for checking whether a requested tuple or result is available. It *does not* abort the `dispatch` if the result is not available, therefore it can not be used to mimic an `inp`.

The Bonita primitives can be used to mimic the `collect` and `copy-collect` primitives, as demonstrated in Figure 3. If the `destructive` was changed to `nondestructive` then this example would be a `copy-collect`.

| C-Bonita |
|---|
| int rqid, count; |
| rqid = dispatch_bulk(ts1, ts2, ?int, destructive); |
| count = obtain(rqid); |

Figure 3: A Linda style: `count = collect(ts1, ts2, ?int)`.

Having shown how the Bonita primitives can impersonate the Linda primitives the next section shows how they can be used in their own right.

## 6 Using the Bonita primitives independently

In this section the use of the Bonita primitives to improve both performance and provide extra functionality is demonstrated. The examples used represent constructs that have been used to date. There are probably other useful constructs that can be produced using the Bonita primitives.

### 6.1 Increasing performance

The most obvious use of the new primitives is to parallelise tuple space access with computation within a user process. Figure 4 demonstrates this. The Linda version "performs the calculation", then retrieves a tuple and then calls a function to use it. The Bonita version performs exactly the same computations, but dispatches a request for a tuple ahead of needing it, thereby parallelising the tuple retrieval with computation, so potentially $T_{SendRequest}$, $T_{Queue}$, $T_{Process}$, $T_{Block}$, $T_{Process}$ and $T_{SendReply}$ are performed in parallel with the function `perform_calculation()` and therefore the total time taken for the Bonita version is the time taken for the Linda version minus these time costs.

As well as providing the potential to parallelise computation and tuple space access the Bonita primitives also provide the potential to pipeline access to

| C-Linda |
|---|
| int x; |
| |
| perform_calculation(); |
| in(ts, "RESULT", ?x); |
| use_result(x); |
| **C-Bonita** |
| int rqid; |
| int x; |
| |
| rqid = dispatch(ts, "RESULT", ?x, destructive); |
| perform_calculation(); |
| obtain(rqid); |
| use_result(x); |

Figure 4: Parallelising tuple space access and computation.

tuple spaces. Figure 5 demonstrates the parallelisation of tuple space access. The Linda version uses `in` to retrieve three tuples[3] and the Bonita version does the same, except the three requests are dispatched and then retrieved. In the worst case the execution times of the two examples will be the same. In the best case the Bonita versions execution times will be the time taken to do a single Linda `in` + $2 \times T_{Unpack}$.

| C-Linda |
|---|
| |
| in(ts1, "ONE"); |
| in(ts2, "TWO"); |
| in(ts3, "THREE"); |
| **C-Bonita** |
| int rqid1, rqid2, rqid3; |
| |
| rqid1 = dispatch(ts1, "ONE", destructive); |
| rqid2 = dispatch(ts2, "TWO", destructive); |
| rqid2 = dispatch(ts3, "THREE", destructive); |
| obtain(rqid1); |
| obtain(rqid2); |
| obtain(rqid3); |

Figure 5: Pipelining multiple tuple space access.

### 6.2 Coordination constructs

The previous examples have shown how the new primitives can be used to provide performance increases. The next example demonstrates how new

[3]In this case from three different tuple spaces but this need not be the case.

more efficient coordination constructs can be produced using the primitives. The first of these constructs is similar to an ALT construct in the occam language, a non-deterministic choice operator. This construct allows a number of different tuples to be requested and then perform actions as the results arrive. The need for such a construct is discussed in Kaashoek et al.[10]. Figure 6 demonstrates this construct. Initially, a number of tuples are requested, in this case from different tuple spaces but they could be from the same tuple space. The while loop then keeps checking *locally* using the `arrived` primitive to see if either of the requested tuples have arrived. When either of them arrives the appropriate function is called. Currently there is no concept of a cancel, once a request has been made it will persist forever (or at least until the system (*not process*) terminates)[4].

C-BONITA
```
int rqid1, rqid2;

rqid1 = dispatch(ts1, "FIRST", destructive);
rqid2 = dispatch(ts2, "SECOND", destructive);

while (1)
{
  if (arrived(rqid1)) { do_first(rqid1);
    rqid1 =dispatch(ts1, "FIRST", destructive);
    break; }
  if (arrived(rqid2)) { do_second(rqid2);
    rqid2 =dispatch(ts2, "SECOND", destructive);
    break; }
}
```

Figure 6: A non-deterministic choice construct.

In Figure 6 a simple example is shown. It is quite possible to insert other checks (for keyboard presses perhaps) within the while loop. Although this example uses polling to check if the tuples have arrived this is *local* and does *not* require communication to the remote control system. When a tuple is found it is processed, then the same tuple is requested again. Currently in Linda an `inp` or `rdp` would be used for checking whether a tuple has appeared. This is demonstrated in Figure 7 which performs a similar function to the example in Figure 6. However, there are three problems with the Linda version:

- The first is that this uses polling which is *not* necessarily local, so it keeps sending messages to the underlying control system, causing it to have to keep searching for the same tuple and potentially not finding it, an expensive operation both in terms of computational time wasted in the control system and the number of messages being sent through the communication channels. If we assume that both the required tuples for the examples appear in the tuple space once, then the C-BONITA version will require exactly four messages to be passed from the user process to the control system. The C-Linda version will require between 4 and an unbounded number messages (it will be a multiple of 2) to be passed between the user process and the control system.
- The second problem with the Linda version is the time taken to perform an `inp` has the costs explained in Section 3.1, meaning that each of the primitives could block for some time, acceptable in this situation but perhaps if a program interacts with a human not acceptable.
- The third is that not all systems support an `inp`.

C-LINDA
```
while (1)
{
  if (inp(ts1, "FIRST"))
     { do_first(rqid1); break; }
  if (inp(ts2, "SECOND"))
     { do_second(rqid2); break; }
}
```

Figure 7: A non-deterministic choice construct using Linda.

In response to the second problem other researchers considering the use of Linda in distributed environments have suggested that the addition of timeouts to the Linda primitives[2, 11], whereby an `in` can "block" only for a specified number of seconds. Such an approach at first seems attractive, providing an easy way of controlling a primitive. However, this does not solve the problem, it merely hides it and introduces further problems.

The `in` primitive still takes time (as detailed in Section 3.1), which means that the user process is blocked. Also, some care has to be taken into the exact semantics of adding timeouts. If the timeout is relative to the *user* process then it is possible that a process always "misses" a tuple which exists simply because the overhead times ($T_{Pack}$, $T_{SendRequest}$, $T_{Queue}$, $T_{Process}$, $T_{SendReply}$ and $T_{Unpack}$) are greater than the timeout

[4]We are currently considering the addition of either an explicit cancel or at least an implicit cancel when a process terminates.

specified. In the worst case this can lead to a program live-locking. Alternatively if the timeout refers just to the time $T_{block}$ the primitive can still block the *user* process for far longer than the timeout indicates. Therefore a primitive can still block from a *user's* point of view.

The introduction of timeouts can reduce the communication costs associated with polling for tuples. Instead of using the `inp` primitive to keep checking for a tuple, an `in` primitive can be used. If this is used then the primitive can be forced to block for only a certain time and then return. This *may* reduce the communication associated with the polling but the fundamental approach remains polling involving communication to keep checking if a tuple is present. Of course such an approach introduces the potential for a delay from when a tuple becomes available to when the process requests it, because the process could be blocked on the alternative choice, waiting for that to timeout before checking for the tuple that is now available.

## 7 Current implementation state

Currently a prototype implementation exists. The run-time system is currently a modification of the Linda system described in Rowstron et al.[16] which is built using PVM[18]. A language embedding for C has been developed. The syntax of the Bonita primitives within this embedding is poor, but can easily be improved by the use of a pre-processor. The implementation is currently used upon a network of workstations.

When originally designing the Bonita primitives a larger geographical distribution of processes (workstations) was imagined where the communication latency could potentially be larger than those experienced on a LAN. However, even in the limited LAN environment used to test the prototype there are appreciable speed increases. For example, when pipelining the access of tuple space by a single process as demonstrated in Figure 5 speedups of 40% are achieved when using the Bonita primitives instead of the Linda primitives on an implementation running on a network of workstations. If the system was more geographically distributed the communication times would increase and the speedup provided by the Bonita primitives over the Linda primitives would also increase.

Work is currently being done to implement a number of "real life" distributed systems using the Bonita primitives.

## 8 Future work

The primitives presented here represent a first step towards a complete system for distributed coordination, based on tuple spaces.

Future work will involve examining and integrating into the tuple space model extensions for more advanced tuple matching, control mechanisms to control access to tuple spaces and individual tuples, and a mechanism to efficiently pass different data types between user processes. Also better implementation strategies are needed for the run-time systems used to control tuple spaces to enable them to cope better with large numbers of workstations.

Within the Bonita primitives the role of the *request identifier* is to be considered. Allowing the user to specify different templates within the `dispatch` primitive, the `arrived` primitive, and `obtain` primitive may be more expressive. However, it introduces the problem that a process may block on an `obtain` which can *never* be satisfied because no matching tuple has been requested. The use of compile time analysis may enable the detection of this, and therefore, the ability to inform the programmer that such a situation exists.

The addition of a `cancel` primitive also appears attractive for use with the `dispatch` primitive. However, the addition of such a primitive may lead to an impossible semantics for `dispatch_bulk`, because tuples could potentially be matched and have left the tuple space when the `cancel` is performed, thereby the tuple arrives at the user process and has to be returned to the source tuple space. One approach may be to relax the `dispatch_bulk` primitives semantics in conjunction with the addition of access controls to tuples and tuple spaces. The addition of a `cancel` primitive for the `dispatch_bulk` primitive is more complex, because the cancelled primitive may have already either copied or moved the tuples and these may have been consumed by other processes. The protocols needed to enable the addition of the `cancel` primitive may be very costly. And finally a more formal semantics of the primitives would be useful.

## 9 Conclusion

The work presented here was motivated by previous work completed on the development of Linda run-time systems for networks of distributed workstations. Detailed analysis of run time performance of these systems indicated that the primitives that involved the retrieval of tuples spent most time being blocked because of the time taken for messages to pass through the network and be serviced, rather than matching tuples not being available.

The BONITA primitives have been presented as the solution to this problem because they provide asynchronous access to tuple spaces. Therefore, user computation and tuple space access can be performed concurrently. These primitives have the useful property of being able to express the main primitives of Linda (`out`, `in`, `rd`). The use of the primitives to provide a non-deterministic choice construct and increased efficiency has been shown. The non-deterministic choice construct is of particular interest because of the few implementations that support the full `inp` and `rdp` primitives, and the reduction in communication that is achieved when it is used.

The Linda model is a very good model for *parallel processing*. It provides a simple and clean interface for parallel processing on dedicated parallel machines and small networks of workstations. Much of the performance of the best implementations is due to the compile time analysis which can be performed in such closed environments. However, it is perhaps not so appropriate for more *distributed* systems and we believe the BONITA primitives are a good first attempt to overcome the problems.

**Acknowledgements**

During this work Antony Rowstron was supported by a CASE grant from the EPSRC of the UK and British Aerospace Military Aircraft Division. The authors would like to thank Andrew Douglas, Ronaldo Menezes and the referees for their useful comments and discussions on the contents of this paper.

## References

[1] Scientific Computing Associates. *Linda: User's guide and reference manual.* Scientific Computing Associates, 1995.

[2] M. Banville. Sonia: an adaption of Linda for coordination of activities in organisations. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 57–74. Springer-Velag, 1996.

[3] P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[4] N. Carriero and D. Gelernter. *How to write parallel programs: A first course.* MIT Press, 1990.

[5] A. Douglas, A. Rowstron, and A. Wood. ISETL-LINDA: Parallel programming with bags. Technical Report YCS 257, University of York, 1995.

[6] A. Douglas, A. Wood, and A. Rowstron. Linda implementation revisited. In P. Nixon, editor, *Transputer and occam developments*, Transputer and occam Engineering Series, pages 125–138. IOS Press, 1995.

[7] D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe. Volume II: Parallel Languages*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer-Verlang, 1989.

[8] S.C. Hupfer. Melinda: Linda with multiple tuple spaces. Technical Report YALEU /DCS/RR-766, Yale University, 1990.

[9] K.K. Jensen. *Towards a Multiple Tuple Space Model.* PhD thesis, Aalbrog University, Department of Mathematics and Computer Science, 1993.

[10] M. Kaashoek, H. Bal, and A. Tanenbaum. Experience with the distributed data structure paradigm in Linda. In *Distributed and Multiprocessor Systems Workshop*, pages 175–191. USENIX Association, 1989.

[11] T. Kielmann. Designing a coordination model for open systems. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 267–284. Springer-Velag, 1996.

[12] K. Landry and J. Arthur. Instructional footprinting and semantic preservation in Linda. *Concurrency: Practice and Experience*, 7(3):191–207, 1995.

[13] A. Rowstron. *Bulk primitives in Linda run-time systems.* PhD thesis, University of York, 1997.

[14] A. Rowstron, A. Douglas, and A. Wood. A distributed Linda-like kernel for PVM. In

J. Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, editors, *EuroPVM'95*, pages 107–112. Hermes, 1995.

[15] A. Rowstron, A. Douglas, and A. Wood. Copy-collect: A new primitive for the linda model. Technical Report YCS 268, University of York, 1996.

[16] A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 510–513. Springer-Verlang, 1996.

[17] A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 357–367. Springer-Velag, 1996.

[18] V. Sunderam, J. Dongarra, A. Geist, and R Manchek. The pvm concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–547, 1994.