Using the BONITA primitives: A case study

Antony Rowstron*
Computer Laboratory, Cambridge University,
New Museums Site,
Pembroke Street,
Cambridge CB2 3QG, UK

Abstract

The co-ordination language Linda has been used for parallel processing for many years. Linda uses a shared tuple space and a number of primitives to provide access to the tuple space and thereby enabling communication between processes executing concurrently. Linda provides asynchronous communication between processes, but *synchronous* access between the processes and the tuple spaces. The BONITA primitives are a different set of primitives that provide asynchronous access to the tuple spaces. The BONITA primitives can emulate the primary Linda primitives and therefore provides both asynchronous access and synchronous access to tuple spaces. It has been previously claimed[15] that asynchronous tuple space access primitives are required to provide new co-ordination constructs and to improve performance for *geographically distributed* processes which are required to co-ordinate distribute processes (or agents).

In this paper a talk program is used as an example to demonstrate that the concept of tuple spaces are well suited for process co-ordination for distributed processes (or agents), and to provide a comparison between the use of Linda primitives and the BONITA primitives. It is shown that asynchronous tuple space access is essential for such process co-ordination.

1 Introduction

The concept of shared tuple spaces for parallel process co-ordination has been being used successfully for many years, and is the foundation of the Linda[4] co-ordination language. A tuple space is an unordered collection of tuples, and the Linda primitives provide the access mechanisms for the tuple space. Although Linda provides asynchronous process communication it provides primarily[1] syn-

*E-Mail: Antony.Rowstron@cl.cam.ac.uk

[1] The `out` primitive may be seen as either a synchronous or asynchronous primitive depending on whether the implementation supports `out` ordering (see Section 7.2).

chronous tuple space access. The BONITA[15] primitives are a set of asynchronous access primitives for shared tuple spaces.

In distributed environments the need for asynchronous primitives is driven by both functionality and performance concerns. The BONITA primitives can be used to emulate the Linda primitives[2] and therefore provide both asynchronous and synchronous tuple space access.

A detailed description of the BONITA primitives can be found in Rowstron et al.[15], and the only difference is that another primitive has been added to the BONITA primitives, called `cancel`. This new primitive described in detail in Section 3.

A justification in terms of performance of the BONITA primitives is presented in detail in Rowstron et al.[15] and will not be reiterated in this paper. In this paper a comparison of the use of C-Linda and C-BONITA for an interactive talk tool for use over a WAN is presented, rather than a LAN. The implementation of the C-BONITA uses the run-time system used in the York Linda Kernel II[13, 12].

The original Linda primitives are described in Section 2. The BONITA are described (informally) in Section 3. The general structure and functionality of the talk program is described in Section 4, and the C-Linda version is described in detail in Section 5, and the C-BONITA version is described in detail in Section 6. A comparison (partly based on experimental results) is presented in Section 7 and finally a review of other relevant work is presented in Section 8.

2 The Linda primitives

The Linda model is now well known and a detailed description can be found in Carriero et al.[5]. 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 primitive (and subsequently 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.

- **eval(ts, active-tuple)**

[2] Except `inp` and `rdp`.

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 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 there relationship with each other.

There are two new primitives which have been proposed that are of particular interest; `collect`[3] and `copy-collect`[14]. The `collect` primitive (collect(*ts1, ts2, template*)) *moves* all available tuples from `ts1` that match `template` into `ts2`, returning a count of the number of tuples moved. The `copy-collect` primitive (copy-collect(*ts1, ts2, template*)) 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.

3 The BONITA primitives

There are many properties of the tuple space model which makes it a good model for distributed systems. It allows asynchronous inter-process communication and allows communicating processes to be both spatially and temporally separated, which is important for distributed systems. This is achieved because the tuple spaces are persistent; a tuple space exists even when the process which creates it has terminated. This in fact provides what is known as orthogonal persistence[2]. We are proposing a new set of primitives which use the tuple space model (with multiple tuple spaces). 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 for using tuple spaces in *geographically 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. Further more the same tuple and template

matching is used. The matching criteria are summarised here; a tuple is matched by a template, if the tuple has the same cardinality as the template, and 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[3].

Here are the informal semantics of the BONITA primitives:

- **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. The primitive is non-blocking.

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

  This initiates the movement of tuples between tuple spaces. Tuple space `ts1` is the source tuple space and the destination tuple space is `ts2` and the tuples are either moved (`destructive`) or copied (`nondestructive`). A count of the number of tuples copied or moved is returned. This is achieved by again the primitive returning a *request identifier* (**rqid**) which is subsequently used with other primitives to get a count of the number of tuples moved or copied. The number of tuples moved or copied depends on the stability of the tuple space. If the tuple space is stable (there are no operations which are destructive being performed in parallel with this primitive) then all the available tuples are copied or moved. The semantics of the primitive, when other primitives are being performed concurrently can be found in the description of the `copy-collect` primitive in Rowstron[12]. The primitive is non-blocking.

- **arrived(rqid)**

  This primitive checks to see if the result associated with rqid is available. If the result is available then the primitive returns the result. The primitive is *non-blocking* and returns false if the result associated with rqid is not yet available. The result will either be a tuple (the result of a `dispatch`) or an integer (the result of a `dispatch_bulk`).

[3]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.

- **obtain(rqid)**

  This primitive is similar to the `arrived` primitive except it blocks waiting for the result associated with rqid to become available if it is unavailable.

- **cancel()**

  This primitive is used to inform the system that *all* pending dispatches are to be cancelled. The primitive is non-blocking.

- **ts = tsc()**

  This primitive creates a new tuple space and returns a handle which uniqly identifies the created tuple space.

The exact syntax of each of the primitives depends upon the host language being used. For example, the syntax of the `obtain` and `arrived` primitives may include variables to be used to store the information returned. Also the primitives may return error values, for example 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` *or* `arrived` is performed for a rqid. In the ISETL-BONITA version tuples are first class objects so the `obtain` primitive returns a tuple, and the `arrived` primitive returns either a tuple or false.

The current primitives use a *request identifier* to enable the returned tuple to be found by the other primitives. Such an approach has the advantage that it is implementationally cheap, but has the disadvantage that the management of *request identifier* can be more complex. Another approach, which is currently under evaluation is to allow the `obtain` and `arrived` primitives to use templates (and tuple space names) to retrieve the desired tuples. However, the problem is that it is possible to use different templates in the `dispatch` and `arrived` or `obtain` primitives, which can lead to unintentional deadlock in the programs. This is because a tuple which matches a template given in an `obtain` primitive need not have been requested using a `dispatch` and therefore can never arrive. Therefore, currently, *request identifiers* are still being used.

It should be noted that the non-deterministic nature of the tuple space usage 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 that the BONITA primitives provide, and that is one of *tuple ordering*. If a single process performs several `dispatch`s 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. This is important to ensure that the behaviour of the `dispatch_bulk` primitive is correct. For more information refer to the *out ordering* details in Section 7.2.

4 The talk program

In the following sections the implementation of a simple talk tool using both C-Linda and C-Bonita is considered. The talk program is very simple and the requirements are that an arbitrary number of people should be able to communicate concurrently (interactively) using the talk program and the text that makes the conversation should be stored for future reference. The people involved in the conversation can dynamically alter.

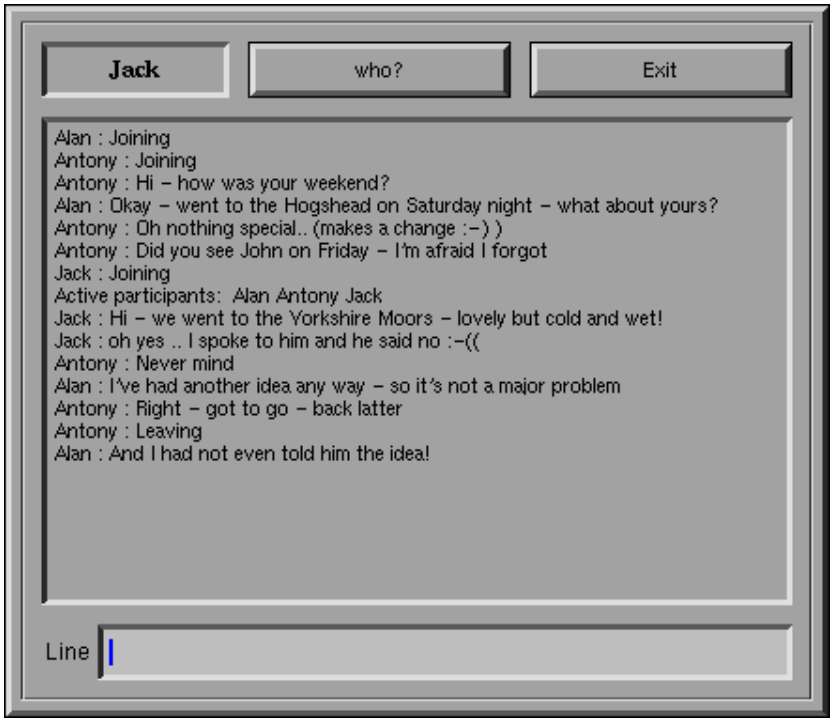

Figure 1: Screen shot of the C-Bonita talk program.

For the both the versions the basic architecture of the talk program is the same. The talk program consists of an *initialisation section* and a *main section*. The initialisation stage displays the conversation to date, and the main section allows the user to participate in the conversation.

A simple X front end[4] has been developed. A screen shot of the C-Bonita version is shown in Figure 1. The main window contains the conversation, with messages being typed in by the user in the *Line* window. Whenever a new talk tool is started, or a talk tool exists, a comment is inserted within the current

[4]Using the XForms package.

conversation. It is also possible for a user to request to know who is currently active within the conversation, using the *Who?* button.

A talk program was chosen primarily because one of the main uses of geographically distributed computing is to support human collaboration. A talk tool embodies two of the requirements: multiple users (many users concurrently using many instantiations of the talk program) and the talk program is interactive. The talk tool also has the advantage that it is simple enough to be easily understood and demonstrate the advantages of the BONITA primitives.

5 C-Linda version

Initially the C-Linda implementation is considered. The *initialisation section* is shown in Figure 2 and the *main section* is shown in Figure 3.

5.1 C-Linda initialisation section

```
1       out(con, user_name);
2
3
in(con, ?num_lines);                /* Get the line counter   */
4       out(con, num_lines, user_name, "Joining");
5       out(con, ++num_lines);
6
7
init_window();                      /* Set up the window      */
8
9       for (pos = 0; pos < num_lines; pos++)
10      {
11        rd(con, pos, ?name, ?text);
12        print_screen(name,text);
13      }
```

Figure 2: The initialisation section of the talk program written in C-Linda.

This initialisation code shown in Figure 2 assumes that each conversation uses a unique tuple space (in this case represented by a tuple space handle called `con`), and this has been initialised and a tuple representing a counter of the number of messages in the conversation is present. Each line of the conversation is a simple tuple of the form $[index_{integer}, name_{string}, textline_{string}]$, and the counter tuple contains the value of `index` for the next line to be inserted. The first thing the initialisation section does is to insert a tuple with the users name

into the conversation tuple space (line 1, Figure 2). This is so other users can ask who is currently active in a conversation. When a user exits the tuple containing the name is removed (line 16, Figure 3). The initialisation section then inserts a line in the conversation to indicate that the a new person has joined the conversation (lines 3-5, Figure 2). This is achieved by removing the counter tuple (line 3), and then inserting a new conversation line (line 4), and then replacing the incremented counter tuple (line 5). Next each line of the conversation is read[5] and printed on the screen (lines 9-13, Figure 2).

5.2 C-Linda main section

```
1       next = num_lines;
2
3       while (!exit_status)
4       {
5
/* If the next line of conversation here then display it    */
6         if (inp(con, next, ?name, ?str))
7         {
8           print_screen(name,text);
9           next++;
10        }
11
12        if (ready_line(text_input))
13        {
14          in(con, ?num_lines);
15          out(con, num_lines+1);
16          out(con, num_lines, user_name, text_input);
17        }
18      }
19      in(con, user_name);
```

Figure 3: The main section of the talk program written in C-Linda.

The main section code shown in Figure 3 of the C-Linda program uses polling for detecting the tuple containing the next line of the conversation in the tuple space and for getting user input text. An `inp` primitive is used to keep checking if a tuple containing a new line of the conversation has been inserted into the tuple space (line 6, Figure 3), and if a tuple has been inserted it is displayed and the counter incremented thus enabling checking for the next tuple (lines

[5]The last n lines could be read and displayed rather than all the lines.

8 and 9, Figure 3). The function `ready_line` manages the input of text from the keyboard and checks if a line of text is ready. If a line of text is ready it is inserted into the conversation by retrieving the counter (line 14, Figure 3), reinserting it incremented (line 15, Figure 3), and then adding the new tuple to the conversation (line 16, Figure 3).

It should be noted that the `inp` primitive is used. In Section 7 the implications of not using `inp` are discussed.

6 C-BONITA version

Having considered the C-Linda version of the talk tool, the C-BONITA version is now considered. The *initialisation section* is shown in Figure 4 and the *main section* is shown in Figure 5.

6.1 C-BONITA initialisation section

```
1
dispatch(con, user_name);          /* Place name in tuple space */
2
3
in(con, ?num_lines);               /* Get the line counter      */
4       dispatch(con, num_lines, user_name, "Joining");
5       dispatch(con, ++num_lines);
6
7
init_window();                     /* Set up the window         */
8
9
/* Get the lines of text - pipelining the tuple space access   */
10
11      for (pos = 0; pos < num_lines; pos++)
12
ref_arr[pos] = dispatch(con, pos, ?name, ?text, NONDEST);
13
14      for (pos = 0; pos < num_lines; pos++)
15      {
16        obtain(ref_arr[pos]);
17        print_screen(name,text);
18      }
```

Figure 4: The initialisation section of the talk program written in C-BONITA.

The initialisation code for the C-BONITA version functionally does exactly the same as the C-Linda version. Indeed the first lines are identical, except for the `out` is replaced with a `dispatch` (line 1, Figure 4). Because the Linda primitives can be emulated using the BONITA primitives where appropriate they can be used, and indeed in the current version of the C-BONITA the Linda primitives are macros containing the pairs of BONITA primitives. However, the second part of the initialisation code uses the BONITA primitives to obtain pipelining of the tuple space accesses. A request for all of the tuples required is made first, then each of the tuples is actually retrieved and dispatched. The requesting of all tuples is achieved within the for loop (line 11, Figure 4) and the `dispatch` primitive (line 12, Figure 4). The *request identifier* for each of the `dispatch` primitives is stored (in an array), to be used when retrieving the results. The for loop (line 14, Figure 4) and the `obtain` primitive (line 15, Figure 4) retrieve the requested tuples, and the retrieved conversation text is displayed on the screen (line 17, Figure 4).

6.2 C-BONITA main section

As with the C-Linda version the C-BONITA version uses polling to check if the next line of the conversation is available or if there is user input ready. The polling of the next line of the conversation *does not* use `inp` but instead requests the tuple using a `dispatch` primitive (line 2 and 11, Figure 5) and checks for the tuples arrival using the `arrived` primitive (line 7, Figure 5). It should be noted that the `dispatch` primitive on line 2, Figure 5 is used to request the first tuple containing a text line that was not displayed during the initialisation section, and subsequent tuples containing text lines are requested by the `dispatch` on line 11, Figure 5. If the tuple is available then the `arrived` primitive retrieves it, and the text within the tuple is displayed (line 9, Figure 5). Again the function `ready_line` is used to manage and check the text input by the user. If a line of text is available, as with the C-Linda version, the counter is retrieved (line 16, Figure 5), and then incremented and reinserted (line 17, Figure 5) and then the tuple with the line of text is inserted (line 18, Figure 5). As with the C-Linda version the users' name is removed from the tuple space (line 21, Figure 5) and finally a `cancel` is performed to terminate all pending `dispatch` primitives.

These sections have outlined the code for the talk program in both C-Linda and C-BONITA. In the next section a comparison of the different programs is presented to demonstrate the advantages of using the BONITA primitives.

7 Comparison of the C-Linda and C-BONITA programs

By examining the two programs it is clear to see that the C-Linda version appears more compact and in some ways more elegant. First the two initialisation

```
1       next = num_lines;
2       ref = dispatch(con, next, ?name, ?text, NONDEST);
3
4       while (!exit_status)
5       {
6
/* If the next line of conversation here then display it    */
7         if (arrived(ref))
8         {
9           print_screen(name,text);
10          /* Request the next line */
11          ref = dispatch(con, ++next, ?name, ?text, NONDEST);
12        }
13
14        if (ready_line(text_input))
15        {
16          in(con, ?num_lines);
17          dispatch(con, num_lines+1);
18          dispatch(con, num_lines, user_name, text_input);
19        }
20      }
21      in(con, user_name);
22      cancel();
```

Figure 5: The main section of the talk program written in C-BONITA.

sections are compared, and then the two main sections are compared.

7.1 Comparison of initialisation sections

The fundamental difference between the two initialisation sections, Figures 2 and 4, is the way in which the tuples representing the text of the past conversation is retrieved. In the C-Linda version this is achieved by the repeated use of the `rd` primitive, and in the C-BONITA version by the repeated use of the of `dispatch` and `obtain`. The C-BONITA version provides pipelined access to the tuple space, where as the C-Linda version does not. The pipelining of the tuple space access cannot lead to the C-BONITA version taking longer than the C-Linda version, and in most cases will provide a speed-up, which in a WAN environment could be considerable. This is because when using a `rd` primitive, a message is dispatched to a run-time system, the run-time system processes the message, and returns a reply message. When the reply message is received the `rd` has completed and the program continues to perform the next `rd` primitive.

In the C-BONITA version the communication between the run-time system and the process is pipelined, the request for another tuple is *not* dependent upon one request being completed before the next request can be sent.

The experimental results shown in Table 6 are an example of the speed up that can be achieved. The table shows the time taken for the retrieval of a number of tuples. These results were produced using a network of Silicon Graphics Indy workstations connected by a non-dedicated 10Mbit/s Ethernet LAN network. The tuple space kernel[6] was placed on a single Indy Workstation. The kernel used for the C-Linda test program was the York Kernel II[13, 12], and for the C-BONITA test program a slightly modified York Kernel II. None of the modifications to the kernel were to increase efficiency of the kernel, just simply to support the new primitives. The York Kernel II is a kernel which supports the distribution of tuple spaces over many workstations, therefore allowing parallel access to a single tuple space (because the tuple spaces are distributed over all the nodes of the kernel). However, the kernel was configured to act as a single server running on a single workstation and therefore did not provide parallel access to the tuple spaces. The test programs were run on another Indy Workstation. The C-Linda test program was lines 9-13 of Figure 2, and the C-BONITA test program was line 11-18 of Figure 4.

Number of tuples retrieved	C-Linda (seconds)	C-BONITA (seconds)	C-BONITA speedup over C-Linda	
1	0.004	0.004	0 seconds	(0%)
10	0.033	0.019	0.014 seconds	(42%)
100	0.362	0.197	0.165 seconds	(46%)
1000	4.645	2.683	1.962 seconds	(42%)

Figure 6: Timings for retrieving tuples from a tuple space as in the initialisation section.

The results for the C-BONITA version demonstrate the speed advantage of pipelining the tuple space accesses. There are a number of factors that will further influence the performance of the BONITA primitives:

- Using a distributed kernel. If the kernel is distributed then it is possible to have multiple requests serviced concurrently by the kernel. Therefore, when using the BONITA primitives there is the potential to have different `dispatch` primitives being serviced concurrently. This is assuming that the time taken to pack the message is *less* than the time taken for the run-time system to decode the message and check if a suitable matching tuple exists. If this is true, then the `dispatch` messages sent to a centralised kernel will be queued at the kernel. If the kernel is parallel then it will potentially be able to service them concurrently.

[6] The run-time system which stores the tuples.

Process one	Process two

`{`
`  out(ts1, 10);`
`  out(ts1, "DONE"));`
`}`
	`{ int x;`
	`  in(ts1, "DONE");`
	`  inp(ts1, ?x);`
	`}`

Figure 7: `out` ordering/`inp` example.

- The experimental results presented here are obtained using a LAN. If the primitives were to be used over a WAN (eg. the Internet) then the *communication* times could increase dramatically. However, the computation times to create the message and computation time at the kernel should not increase (these are independent of communication times). If the communication time increases the effect of pipelining the tuple space accesses will be to increase the speedup of the pipelined access (Bonita primitives) over the non-pipelined access (Linda primitives).

  It should be noted that if the underlying communication system does *not* support asynchronous message passing then the C-Bonita version will not perform significantly better than the C-Linda version.

7.2 Comparison of main sections

The fundamental difference between the two main sections, Figures 3 and 5, is the way in which the tuples representing the next line of text in a conversation are retrieved. In the C-Linda version this is achieved using the `inp` primitive to check for the next tuple, and in the C-Bonita version this is achieved by requesting the tuple using a `dispatch` and then using an `arrived` primitive to check if it is available.

The `inp` primitive is not widely supported in Linda implementations. There are a number of perceived "semantic problems" associated with these primitives which are used as the primary reason for their removal[11], replacement[3] or, when implemented, behaviour which can potentially lead to unintentional program behaviour[1]. The problem is based on the semantics of the `out` primitive. Given the example code fragments in Figure 7, can the `inp` in *process two* fail?

Some implementations[1] of Linda state that it is possible for the `inp` to fail, because `out` is an asynchronous primitive and therefore completion of the primitive does not indicate that the tuple is present within the tuple space. On the other hand, if it is assumed that the `out` primitive is synchronous then the `inp` can not fail[7] because the two processes synchronise on the tuple ["DONE"$_{string}$]. Therefore, the previous `out` primitive must have completed

[7] Assuming there are no other process using the tuple space which `ts1` refers to.

before the ["DONE"$_{string}$] is inserted, so the tuple [$10_{integer}$] will be present. If this is not true then this allows implementations to create `inp` as a function that always returns false. A detailed description of the problem can be found in Leichter[11] and Rowstron[12]. Here, it is assumed that `out` primitives are ordered.

The C-Linda approach of using `inp` has a disadvantage over the C-BONITA approach. The disadvantage is the level of communication between the user process and the run-time system. The C-BONITA approach requires two messages to pass between the user process and the run-time system. One message associated with the `dispatch` primitive, and one associated with the reply for that `dispatch` primitive. The use of the `arrived` primitive is simply a local check within the user process to see if the reply message has arrived. If it has not there is no need to pass information to the run-time system. Whereas, in the C-Linda version whenever and `inp` is performed a message is passed to the run-time system, and a message is required back before the primitive completes. The reply message will either contain a tuple or an indication that the tuple does not exist. Each time an `inp` is performed this will require the run-time system to check all potential tuples for a match, so not only increasing the communication load within the system, but also increasing the computational load within the kernel. Even if the kernel is distributed this extra load could be very significant, particularly if there are many instantiations of the talk program running concurrently.

There is also the fact that whilst the `inp` is being performed the user process is blocked. In an LAN environment this is not a problem because the time taken for the `inp` is relatively small. However, if a WAN was being used the time taken to perform an `inp` could potentially become noticeable. A human expects to interact with the talk program and therefore continual delays could be a problem if, for example, this caused the input and display of text the user was writing to be keep pausing. This can be overcome, by effectively running the routines which manage the user input as a separate process. This has two disadvantages however, requiring the system to support multiple processes on the same computer and increasing the complexity of the program. The two processes comprising the talk program can communicate via a tuple space using the Linda primitives[8]. The question is how do I know that the time taken by the `inp` primitive is going to effect the interaction of the program with the user? It is an arbitrary decision which must be made by the programmer. The style of programming offered by BONITA is often more independent of the cost of communication.

7.3 Summary

Therefore, to summarise the advantages of using the BONITA primitives are:

[8] In my opinion the introduction of another inter-process communication method should be avoided.

- speed improvement, providing by being able to pipeline tuple spaces access (shown in the initialisation section);
- reduction in both communication load and run-time system load by the BONITA primitives allowing a more efficient polling than the Linda `inp` primitive provides. In the C-BONITA two messages are required to retrieve a tuple containing a line of text. In the C-Linda version an unbounded number of messages is potentially required (with a minimum of two); and
- programs written using the BONITA primitives often are more independent of the effects of the communication characteristics of the system being used. The same talk program will work on a LAN as over the Internet, without affecting the interaction with the user.

As an aside, one thing that has not been demonstrated in the C-BONITA talk program is that the BONITA primitives can also allow pipelining of tuple space accesses and computation. It is possible to insert computation between a `dispatch` primitive and a `obtain` or `arrived` primitive.

8 Other work

One of the most interesting recent developments in Linda-like systems is the PageSpace project[6] which has been examining the use of Linda-based systems for co-ordination via the WWW. It is based on Java embedding of Linda, called Jada[9]. This appears to use modifications of the current Linda primitives, but maintains the synchronous nature of tuple space access. It is within this type of system that the asynchronous access primitives will be most valuable.

There has been other work looking at the development of primitives for use in open distributed systems[10, 16]. Objective Linda[10] presents a number of extensions to the Linda primitives, to make them more suitable for a distributed and open environment. The primitives retain the synchronous nature of the Linda primitives, but provide bounds for both time a primitive can take and the number of tuples retrieved. Therefore, an `in` and a `rd` can retrieve many tuples, rather than a single tuple. The bounding concept could easily be incorporated into the BONITA primitives. The primitives are also given timeouts, which are appear to be relative to when the run-time system begins to process the operation. When using asynchronous access primitives it is possible to allow the user process to decide how long to wait for a reply.

9 Future work

The run-time system used for the C-BONITA is based on an extension of the York Kernel II[13, 12]. The York Kernel II is a run-time system which supports the

[9] D. Rossi, Department of Computer Science, University of Bologna.

Linda primitives and `collect` and `copy-collect`. However, the York Kernel II is essentially a LAN based run-time system. An extension of the concepts behind the York Kernel II for WAN based computing is described in Rowstron[12], but have not yet been implemented.

Current work on BONITA is concentrating on trying to extend the underlying tuple space model to make it more flexible for the needs for geographically distributed computing. There are three fundamental concerns:

Tuple space handle passing Both the programs assume that the tuple space handle or name is known to each instance of the talk program. However, these have to be passed to the processes in some manner. Laura[16] presents one way of overcoming this type of problem, by using the concepts of services (which is also used in PageSpace[6]).

Tuple space permissions In a distributed environment it is necessary to add read/write etc. permissions to tuple spaces. Currently a form of tuple space access control is being added to the York Kernel II (which with modifications is now called CamKernel 1.0).

Tuple shadowing It is often the case that a structure is imposed on the tuples stored in a tuple space. In the case of the talk program the tuples containing the text that makes up a conversation are ordered into a stream, with the first field representing a tuples position within the stream. When these structures are used within a tuple space usually other tuples are used to store information about the data structure. In the case of the talk program a tuple containing a counter representing the final element of the stream. In order for the stream to be updated this is *destructively* read from the tuple space and then updated and reinserted. Unfortunately, once it is removed no other talk programs can start up and read the initial conversation because they need to access it in order to know how many tuples are in the stream.

Therefore, *tuple shadowing* is being considered for C-BONITA. This is where a new operation descriptor for use with templates and the `dispatch` primitive is added. The operations descriptor can be either DESTRUCTIVE, NON-DESTRUCTIVE or SHADOW. A SHADOW is similar to DESTRUCTIVE except the tuple is not removed from the tuple space but no *destructive* operations (either DESTRUCTIVE or SHADOW) can be performed on it. If a destructive operation is performed and there are no other matching tuples the template will be added to the queue with other unsatisfied templates. When a new tuple is inserted which matches the template used to match the shadowed tuple then the shadowed tuple is removed and the new tuple is inserted. If there are any destructive operations waiting that could be satisfied by the inserted tuple then the inserted tuple will be used.

This is useful because it allows the state of the data structure to be looked at by other process whilst it is being updated by anther process. Therefore, in the C-BONITA talk program the line 16, Figure 5 would be replaced by `obtain(dispatch(con, ?num_lines, SHADOW));`. When the `dispatch` on line 17, Figure 5 is performed and the kernel receives the message the shadowed tuple will be removed. This is currently also being added to the CamKernel 1.0.

10 Conclusions

A simple talk program has been used to compare the differences between the Linda primitives and the BONITA primitives. Through some experimental results it has been shown how some operations are quicker when expressed using the BONITA primitives, rather than the Linda primitives even when using a LAN based implementation.

It has also been shown how the BONITA primitives reduce the amount of communication necessary when using polling to watch tuple spaces. Because of the nature of the communication that tuple spaces provide it is often necessary to use polling to watch state stored in a tuple space.

A true demonstration of the BONITA primitives and their advantages will be only truly possible as kernels supporting thousands of geographically distributed processes are created, which is beginning in the form of kernels like the one used in the PageSpace project, and the one described in Rowstron[12]. As such kernels become available there will be many optimisations that will have to be used, some will be explicit (such as tuple shadowing) and others will be implicit. However, the use of asynchronous primitives in one form or another is, in my opinion, ensured.

11 Acknowledgements

The author would like to thank Andy Hopper and the Olivetti & Oracle Research Laboratory, Cambridge for current financial support, and Stuart Wray at the Computer Lab, Cambridge University. The author would also like to thank Alan Wood and Ronaldo Menezes of the University of York, and Andrew Douglas for their useful comments and discussions on BONITA. Also to the reviewers Rowstron et al.[15] for their comments and suggestions.

References

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

[2] M. Atkinson, L. Daynes, M. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 1996. To appear.

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

[4] N. Carriero. *Implementation of Tuple Space Machines.* PhD thesis, Yale University, 1987. YALEU/DCS/RR-567.

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

[6] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. PageSpace: An architecture to coordinate distributed applications on the web. *Computer Networks and ISDN Systems*, 28:941–952, 1996. Proceedings of the Fifth International World Wide Web Conference.

[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-Verlag, 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] 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.

[11] J. Leichter. *Shared tuple memories, shared memories, buses and LAN's – Linda implementations across the spectrum of connectivity.* PhD thesis, Yale University, 1989. YALEU/DCS/TR-714.

[12] A. Rowstron. *Bulk Primitives in Linda Run-Time Systems.* PhD thesis, Computer Science Department, University of York, 1996.

[13] 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-Verlag, 1996.

[14] 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.

[15] A. Rowstron and A. Wood. BONITA: A set of tuple space primitives for distributed coordination. In Hesham El-Rewini and Yale N. Patt, editors, *Proceedings of the 30th Hawaii International Conference on System Sciences*, volume 1, pages 379–388. IEEE Computer Society Press, January 1997.

[16] R. Tolksdorf. Coordinating services in open distributed systems with LAURA. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 386–402. Springer-Verlag, 1996.