TupleScope: A Graphical Monitor and Debugger for Linda-Based Parallel Programs

Paul Bercovitz and Nicholas Carriero

Research Report YALEU/DCS/RR-782
April 1990

This work was supported in part by National Science Foundation grant CCR-8657615 and in part by Office of Naval Research Grant N00014-89-J-1906.

TupleScope: A Graphical Monitor and Debugger for Linda-Based Parallel Programs

Paul Bercovitz and Nicholas Carriero*

Department of Computer Science
Yale University
New Haven, Connecticut

April 1990

Abstract

The Linda model of parallel programming lends itself to the development of graphical, interactive monitoring and debugging tools. Several of the characteristics of Linda which facilitate visual monitoring and debugging are discussed. TupleScope, an X Window System-based monitoring and debugging tool for C-Linda, is presented. TupleScope effectively provides a window onto tuple space. We also present a special-purpose language for expressing debugging actions. The software architectures of the run-time version of TupleScope, for shared-memory machines, and the postmortem version of TupleScope, for both shared-memory and distributed-memory machines, are described.

TupleScope currently runs on the Encore Multimax and Sequent Symmetry shared-memory multiprocessors, and on Sun workstations. Of the distributed-memory systems supporting C-Linda, TupleScope has been implemented on the Intel iPSC/2 hypercube.

1 Introduction

Parallel programming systems need adequate monitoring and debugging tools, but such tools are often difficult to implement and difficult to use. The more distributed the architecture of a system, the less one expects in the way of

*This research was supported in part by National Science Foundation grant CCR-8657615, and in part by Office of Naval Research grant N00014-89-J-1906. Linda is a trademark of Scientific Computing Associates, Inc. The X Window System is a trademark of MIT. UNIX is a trademark of AT&T. Symmetry and Pdbx are trademarks of Sequent Computer Systems, Inc. Multimax is a trademark of Encore Computer Corporation. iPSC is a trademark of Intel Corporation. Sun is a trademark of Sun Microsystems, Inc.

debugging tools. One of the virtues of Linda, a model for parallel programming in which a sequential language is extended by a set of operations on a logical shared memory, is that it lends itself to the construction of powerful monitoring and debugging tools across a range of machine architectures. TupleScope, an X Window System-based C-Linda monitor/debugger that we have developed, provides many sequential-style debugging functions for both shared-memory and distributed-memory Linda programming environments. While the distributed-memory version of TupleScope performs only postmortem monitoring/debugging, it shares with the run-time version many useful interactive features. TupleScope currently runs on the Encore Multimax and Sequent Symmetry shared-memory multiprocessors, and on Sun workstations. Of the distributed-memory systems supporting C-Linda, TupleScope has been implemented on the Intel iPSC/2 hypercube.

We will describe Linda in the next section, and in section 3 we will discuss some of the requirements for and possibilities of a graphical Linda monitoring and debugging tool. TupleScope is presented in section 4. In the final section, some directions for future development of TupleScope are outlined.

2 Linda

Linda is a model of process creation and coordination which relies on a logical shared memory as the medium in which all interprocess communication and synchronization occur [Gel82, Gel85, CG89b]. This memory is known as *tuple space*, since it contains *tuples*, which are ordered sequences of typed fields, each field consisting of either actual data (an *actual*) or a typed slot for data (a *formal*). Tuple space is content-addressable: tuples are retrievable on the basis of the values they contain. Linda defines a set of *tuple space operations* on tuple space. The **out** and **eval** operations generate tuples, while the **in** and **rd** (read) operations retrieve data from tuple space.

When an **out** is performed on a tuple, it is put into tuple space and becomes accessible to any process by means of an **in** or a **rd** operation. An **in** or a **rd** operation has as its operand not a tuple but a *template*. This template, like a tuple, consists of an ordered set of typed fields. As with a tuple, the fields of a template can be either actual or formal. For a template to match a tuple, it is necessary that each field of the template have the same data type as the corresponding field of the tuple. And, if an actual field of a template is to match the actual field of a tuple, these fields must have the same value. A formal in a matching template is filled with the value of the actual in the corresponding tuple field. When an **in** or a **rd** is performed, it either completes immediately, by successfully matching a tuple in tuple space, or it blocks, until a tuple that it can match against is present in tuple space. An **in** removes the matched tuple from tuple space, while a **rd** copies the matched tuple without removing it.

Eval is similar to **out**, the difference being that **eval** places an unevaluated (a so-called *active*) tuple into tuple space, and its fields are evaluated by means of separate processes. With an **out**, in contrast, the fields of the tuple are evaluated by the same process that executes the **out**, before the tuple is put into tuple space.

The **inp** and **rdp** operations are non-blocking versions of **in** and **rd**. They return the value 1 if a matching tuple is found and the value 0 if no matching tuple is found.

The approach to parallel programming in which process coordination and computation are treated as orthogonal components has been contrasted with the approach in which a single, unifying programming model is used [CG89a]. Linda is an example of the former approach. In C-Linda, the Linda implementation developed at Yale and SCA, C is the computation component and Linda is the coordination component. An approach to parallel programming which exploits this orthogonality leads to the adoption of a similarly orthogonal set of debugging tools. Some of the advantages of providing the user with two axes along which to attack the debugging of a parallel program are discussed below.

3 Monitoring and Debugging a Linda Program

A software tool for monitoring a Linda program will naturally center on tuple space and the operations pertaining to it. More precisely, the essential elements to be monitored are:

1) *the agents*: the processes.

2) *the events*: the tuple space operations.

3) *the field of activity*: tuple space, divided into *partitions*, i.e. regions where tuples of the same type are stored and where templates match tuples of the same type.[1]

A Linda dialect such as C-Linda is a high-level language, implementable on a variety of parallel or distributed machines. Linda programmers deal not at the level of individual physical processors or specific machine architectures but at the higher level of individual threads of control or processes. A Linda monitor/debugger needs therefore to deal with *processes* and their tuple space operations rather than with *processor* configurations. There is no need to track the flow of data between individual processing elements, since Linda processes do not communicate with other processes directly, but only indirectly, through tuple space.

A graphical Linda monitor/debugger should provide a representation of each tuple space operation, including the tuple, the tuple space partition to which it belongs, and the process which is performing the operation. Displaying a tuple textually is non-trivial, since a tuple is itself a complex object, consisting of an arbitrarily long sequence of data fields, each of which can itself be an aggregate such as an array, structure, or union. By making good use of the graphical

[1] See [Car87] for a detailed treatment of tuple space partitions.

resources available in today's workstations, the designer of a Linda monitor can synoptically represent the data associated with a tuple space operation.

We will consider in detail four aspects of the Linda model which greatly facilitate the complex task of constructing a usable parallel program monitor/debugger:

1) *A Linda program can be given a graphical representation*: To represent succinctly the great quantity of information of potential interest, the monitor/debugger should use graphics wherever possible. Linda suggests mental models or images which can form the basis of a graphical rendering of the program.

2) *Tuple space operations can be logically serialized*: Monitoring and debugging tasks are greatly facilitated if the events to be monitored can be serially ordered. It is possible for a Linda monitor/debugger to focus on the activity of one process at a time without altering the logic or the meaning of a program.

3) *A Linda monitor can take advantage of compile-time processing*: C-Linda uses a "pre-compiler" to analyze all of the tuple space operations, grouping them into tuple space partitions according to the type of tuple they operate on. These partitions are analyzed and an efficient data structure for implementing the tuple space operations of each partition is selected. The pre-compiler can also provide a Linda monitor/debugger with a variety of information about a program.

4) *The tuple data structure provides a uniform means for expressing debugging queries*: All tuple space operations operate on the tuple data structure. Hence there is a ready-made way to express monitoring and debugging queries, viz. by specifying the values of one or more fields of a tuple.

These properties of the Linda model permit the construction of tools which greatly facilitate the task of developing, understanding and maintaining parallel programs.

3.1 *The Graphical Representability of a Linda Program*

Among the images with which the Linda programmer is familiar are those having to do with the partitioning of tuple space into disjoint regions where processes exchange tuples of the same type; with the production and consumption of tuples, which occupy tuple space; and with processes which are spawned to evaluate tuples and then disappear as the evaluation completes. A graphical Linda monitor/debugger can make use of these images, without having to invent and import images which are extrinsic to the programming paradigm.

The partitioning of tuple space into regions where the matching of tuples with templates of the same type occurs has an obvious graphical interpretation: the presentation of a separate window for each partition. The user's attention can focus on these partitions as separate spheres of activity. For example, processes can be given sequential access to a critical section by having them successively take and release ownership of a tuple associated with the section.

All of the tuple space operations (**ins** and **outs**) involving this tuple can be viewed in a single window. Another example is that of a process which has blocked when doing an **in** because there are no matching tuples. The user can observe at a glance that this partition is indeed devoid of tuples, even while other tuple space partitions may be populated with tuples. Separate windows provide the requisite functionality for monitoring the activity occurring in a tuple space partition.

Tuples are persistent data structures. They are not modifiable in place. Once a tuple is in tuple space it can only be read or removed as a unit. Because of these properties, a tuple can be graphically represented as a thing which, by analogy with physical things, has a position and occupies space. A tuple space partition, implemented as a two-dimensional window, can be filled with tuples, no two of which "occupy the same space." Tuple space and its partitions are bags or multisets. Hence there is no requirement that a particular tuple occupy a particular position in a partition or in the window representing it.

The graphical capabilities of current workstations make it possible to construct levels of abstraction in the representation of tuples. At the top level, they could be given uniform representation in the form of an icon appearing in a tuple space partition. The user would in many instances not need more specific information: useful information is already being conveyed by its occupying this and not a different tuple space partition. Should additional information about the tuple be needed, the user would descend to the next level of representation by clicking on the tuple icon. A window containing the text of the tuple in detail, field by field, would then pop up. The sheer amount of information to be monitored dictates that this kind of encapsulation take place.

A graphical Linda monitor/debugger should show the occurrence of each **eval**, **out**, **in**, or **rd**, always making clear which process is performing the operation. Each time a process is created, by means of **eval**, or terminated, there should be a depiction of that event. The appearance of a new tuple in tuple space should coincide with the execution of an **out**, just as the disappearance of a tuple should coincide with an **in**. The **in** and **rd** operations sometimes block, as discussed above. The monitor should show that a process is blocked and show its transition from a blocked to an unblocked state. This is essential to the adequate monitoring of a Linda program. One error in program logic that the monitor should help detect is deadlock, by showing that each process is blocked, and, further, by showing on what type of tuple, i.e. in which tuple space partition, each process is blocked.

The principle of levels of abstraction can also apply to the representation of process activity. A process could be represented at the top level by an icon. The process icon could change as the process goes from one type of tuple space operation to another, or as it goes from the state of being blocked to that of being unblocked. More detailed information regarding the state of a process, such as the line of source code corresponding to the last tuple space operation performed, could be obtained by clicking on the process icon. An even more detailed level of information would become available if a conventional symbolic debugger were tied in to cover the specifically computational aspect of the program. It would be essential that the user be provided with the capability to examine or even modify the state of a process at the lowest level. The design challenge is to incorporate low-level debugging as one among several levels of abstraction.

3.2 *Tuple Space Events Are Logically Serializable*

One design option for a monitor/debugger for a MIMD machine is to use a single monitoring process to display the states of multiple processes. This approach facilitates the implementation of debugging functions such as single-stepping which are commonly found in sequential debuggers. The issue that must be faced with this approach is the extent to which distortion is introduced when a serial ordering is imposed on events which, resulting from multiple partially asynchronous processes, have in some cases no inherent temporal ordering.

Consider the approach in which a monitoring process presents at every instant the tuples that currently occupy tuple space, the processes that are executing, and some indication of the status of each process with respect to tuple space, such as the most recent tuple space operation performed by each process. At each instant, what is in effect a global state is presented, and a program is represented as the sequence of such global states. But such a global state of the program does not depend on a *total* ordering of the tuple space operations, since Linda does not require that a temporal ordering be maintained for *every* pair of tuple space operations. It requires only that such an ordering be maintained for *some* pairs of tuple space operations. In other words, Linda requires only a *partial*, not a total, ordering of events. A correct ordering is maintained for those events which do require a certain temporal ordering. Functions such as single-stepping which presuppose the serialization of events can consequently be implemented without altering program logic. A utility to reconstruct the tuple space aspect of a Linda program from a number of event streams, such as that used in the postmortem version of TupleScope, can also be implemented.

3.3 *Compile-Time Support for Monitoring and Debugging*

A Linda pre-compiler can be an important source of data needed by a monitor/debugger. A Linda monitor/debugger which uses the tuple space operation as its basic unit is necessarily a source-level debugger. If the monitor/debugger can rely on the pre-compiler to provide the source file name and line number of each tuple space operation, then it can give the user immediate access to the text—and context—of any tuple space operation.

As indicated, the pre-compiler does a pre-match over all the tuple space operations, partitioning them into sets where the matching of tuples and templates takes place. The results of this analysis can be made available to a graphical monitor/debugger, enabling it to depict a program's complete tuple space partitioning prior to the start of program execution. In unpacking the tuples for display in textual form, a monitor/debugger can rely on detailed compiler-generated information regarding the structure of a tuple, such as the number of fields it has and the data types of the fields.

3.4 *The Tuple as a Uniform Data Structure for Debugging Queries*

In Linda, interprocess communication and synchronization are accomplished by producing, consuming and reading tuples. Tuples and their fields will naturally figure in many debugging queries. Tuple space could function as a kind of database to which queries could be addressed. Queries could be expressed as conditionals which determine whether the fields of a tuple have certain values. A variety of monitoring and debugging actions could be triggered upon satisfaction of such queries: breakpointing, filtering, highlighting or differentiating tuples by means of color, saving the contents of tuple space to a file, etc. Since the Linda programmer already thinks in terms of tuples and their fields, forming queries in such terms would be easy.

4 TupleScope

In the previous section several of the features that one would expect a graphical Linda monitor/debugger to provide have been discussed. In what follows, one implementation of a graphical monitor/debugger for C-Linda programs, TupleScope, is presented. C-Linda consists of a compile-time component, where the parsing, tuple analysis, optimization and code generation take place, and a run-time component, which is essentially a library of routines to implement the tuple space operations. When the user wants to use TupleScope with his application, he uses a command line option to the compiler which causes it to generate data structures required by TupleScope and to link in an alternate run-time library.

4.1 *User Interface*

A program that uses TupleScope is invoked in the same way as a program lacking TupleScope support. But instead of beginning execution immediately, a group of windows appears on the display—a control window, where a variety of options and functions can be selected, and a window for each of the tuple space partitions. These windows can be moved anywhere on the display, stacked and iconified. Execution of the C-Linda program begins when the "Run" function button is clicked. Processes are numbered according to the order of their creation, and are represented by icons bearing the process number. While executing, a process can be mapped to the tuple space partition of its most recent tuple space operation. In TupleScope, this mapping is expressed by the appearance of each process icon at the top of the window representing the tuple

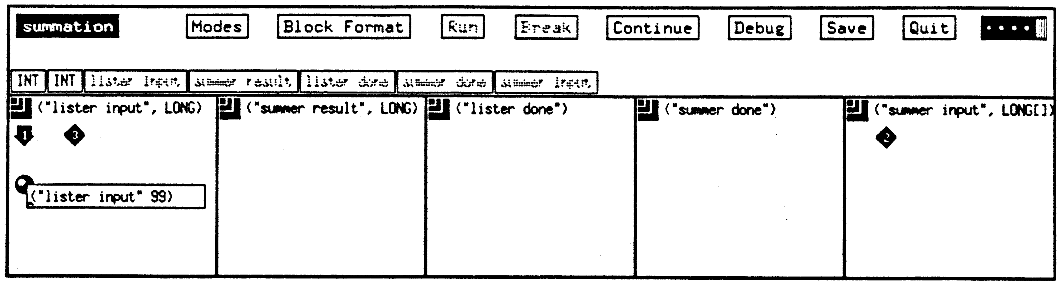


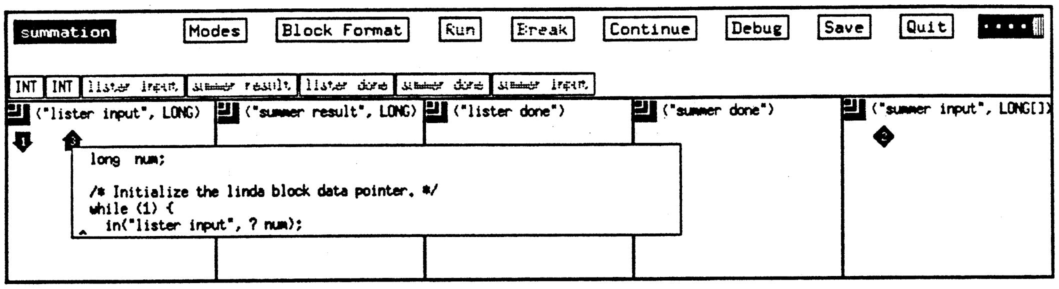
Figure 1: Process 1 **outs** a tuple containing user input.

space partition which it most recently operated on. The process icon changes to reflect its most recent tuple space operation: a black down arrow for the **out** and **eval** operations, both of which produce a tuple; a black up arrow for the **in** operation; and a white up arrow for the **rd** operation. When the **in** and **rd** operations are in a blocked state they are represented by black and white diamonds, respectively. Suspension of execution at any moment provides a view of the most recent tuple space operation performed by each process on some tuple space partition. Source code is integrated into the TupleScope display: clicking on a process icon causes a scrollable window containing the text of the source file, with the line containing the tuple space operation pinpointed, to pop up. This is an example of a debugging function that relies on pre-compiler support.

Most of the tuple space partition window is reserved for the display of tuples, represented by sphere-shaped icons. The user can select from three sizes of tuple space partition windows. If a window becomes filled with tuples, the tuples are redisplayed on a smaller scale. When a tuple icon is clicked, a scrollable window containing the text of the tuple, field by field, pops up. In C-Linda, these fields can consist of scalars, having any of the scalar data types of the C language, or aggregates, i.e. arrays, structures, or unions.[2] The iconification of tuples helps in managing the potentially large quantity of tuple data.

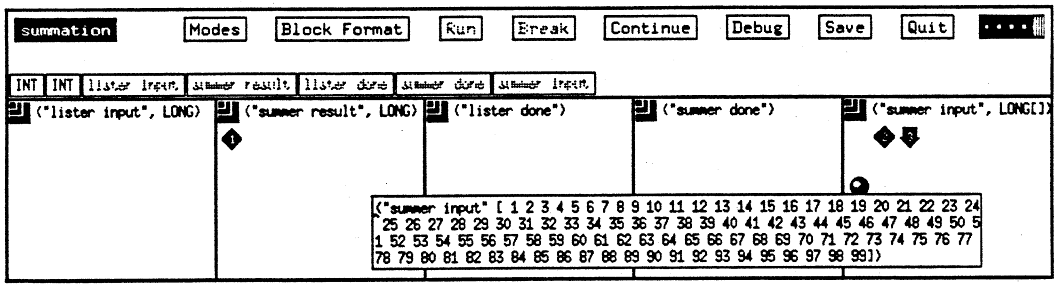
Figures 1 through 5 depict TupleScope in five stages of the execution of a simple program in which the user is prompted for an integer, and the sum of the integers from 1 to the given integer is calculated. The Linda processes are given specific tasks: process 1 is responsible for handling the terminal I/O; process 3 is responsible for creating the list of integers to be summed; and process 2 is responsible for calculating the sum. In Figure 1 process 1 has just received the user input and is **out**ing a tuple containing this data. Process 3 is blocked on an **in**, waiting for this tuple. Process 2 is also blocked, waiting for the tuple to be produced by process 3. In Figure 2 process 3 is **in**ing the tuple containing the user input. A window has been opened indicating the line of code which is being executed by process 3. Figure 3 shows process 3 **out**ing a tuple consisting

[2]In the current implementation, structure and union fields are displayed as arrays.



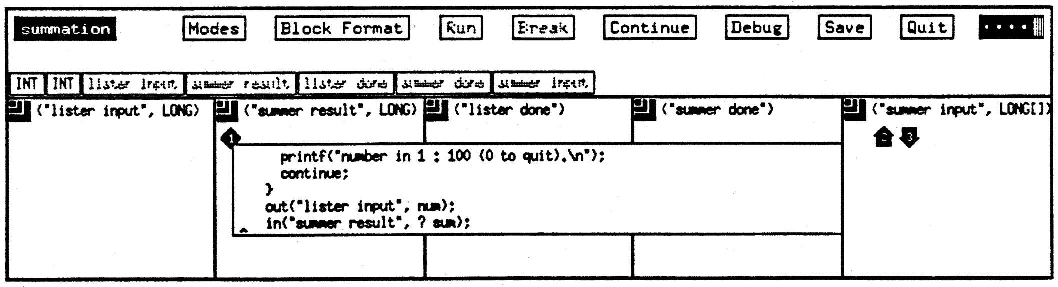
```
long  num;

/* Initialize the linda block data pointer. */
while (1) {
  in("lister input", ? num);
```

Figure 2: Process 3 ins the tuple outed by process 1.



```
("summer input" [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 5
1 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99])
```

Figure 3: Process 3 outs a new tuple containing a sequence of integers to be summed.



```
    printf("number in 1 : 100 (0 to quit).\n");
    continue;
  }
  out("lister input", num);
  in("summer result", ? sum);
```

Figure 4: Process 2 ins the tuple outed by process 3.

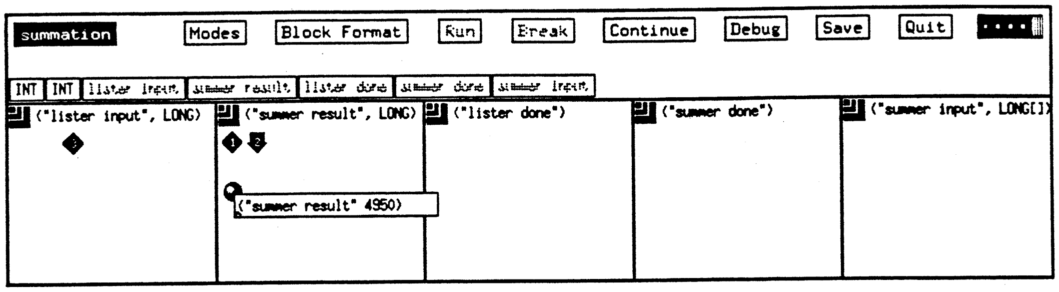


Figure 5: Process 2 **outs** a tuple containing the result of summing the integers.

of the list of integers to be summed. A window displaying the contents of the tuple (a character string followed by an array of long integers) has been opened. Figure 4 shows process 2 **in**ing the tuple **out**ed by process 3 and shows the line of code containing the **in** operation on which process 1 is blocked. Figure 5 shows the result tuple produced by process 2. Process 1 is waiting for the tuple so that it can write the result to the terminal and prompt the user for the next integer.

4.2 *Breakpoints, Single-Stepping, Speed Control*

As pointed out above, a special monitoring process can be used to sequentially focus on the activity of each Linda process without affecting the program logic. Scanning the processes round-robin, TupleScope updates the status of each process with respect to tuple space. Each tuple space operation serves as a potential breakpoint. When a breakpoint has been reached, the monitoring process communicates to the other processes that they are to suspend execution. Thus the entire Linda program halts, albeit with a degree of latency, since a breakpoint is only detected prior to the execution of a tuple space operation. Besides being settable interactively and at arbitrary times, by clicking the "Break" button, a breakpoint can also be set under the control of a program written in the TupleScope Debugging Language, which is described below.

TupleScope's single-stepping function is simply an extension of the breakpoint function: in single-step mode a breakpoint is automatically inserted at each tuple space operation. By clicking the "Continue" button the user can proceed from one tuple space operation to the next.

Another useful TupleScope function, similar in design to the single-step function, is the speed control function. Rather than causing each process to suspend execution when the monitor detects a new tuple space operation, execution is merely delayed, for a duration selectable by the user. By sliding the Speed Control bar to a position along a range of settings, the user can slow down the execution of the program to a speed suitable for the debugging or monitoring task at hand. When a program with TupleScope support is running full speed, the user's grasp of what is happening in tuple space is generally restricted to a perception of general patterns of data movement and process activity.

4.3 *The TupleScope Debugging Language*

We have discussed how a Linda monitor/debugger could utilize the tuple as a common data structure for formulating debugging queries. Expressed as conditionals, such queries could control the triggering of a range of debugging-related events. While there are several ways in which such functions could be realized, in TupleScope this is accomplished by having the monitoring process execute an auxiliary program written in a Linda-specific debugging language. The debugging programs can be written, translated, revised and cancelled during execution of the C-Linda program. The TupleScope Debugging Language is a simple, high-level notation for expressing a set of debugging actions, each triggered by the satisfaction of a condition. A Debugging Language program is translated, using the UNIX utilities **lex** and **yacc**, into code interpretable by TupleScope. When a program is in effect, each monitored tuple space operation is examined to determine whether one or more of the conditions exists, and if so, the associated action is taken.

A TupleScope Debugging Language program consists of one or more *statements*. Each *statement* has the form:

if *condition* **then** *action*

A *condition* consists of one or more *tests*. The *tests* are separated by one of the logical operators, **and** or **or**. A *condition* is enclosed by parentheses and a *test* is enclosed by square brackets.

There are three kinds of *tests*: tuple field comparison tests, tuple space operation comparison tests, and process number comparison tests. Each *test* evaluates to true or false, and contributes to the truth value of the *condition* in conjunction with any logical operators.

A tuple field comparison test detects tuples, based on the values of one or more of the tuple fields.[3] A tuple field comparison test has the form:

field *N* [*equality-operator* | *relational-operator*] *constant*

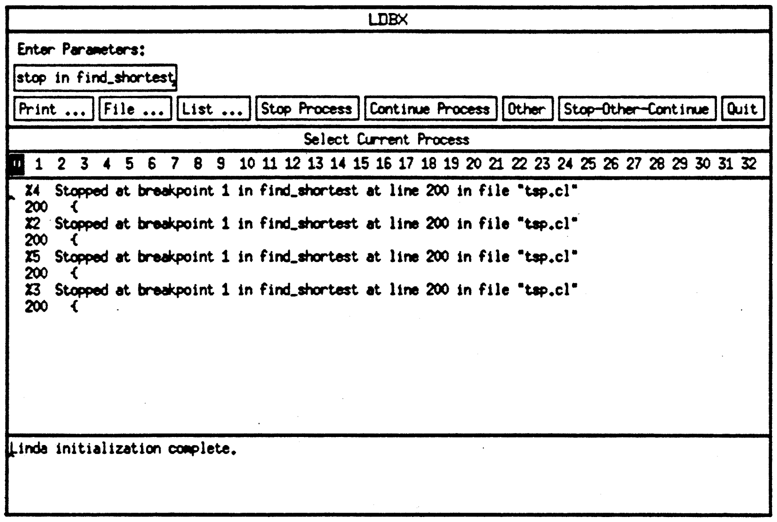
N is an integer greater than or equal to 1. An *equality-operator* is either

== (equals) or != (does not equal).

A *relational-operator* is either

< (less than) or > (greater than).

[3]In this preliminary version of the TupleScope Debugging Language elements of aggregates such as arrays or structures are not specifiable.



```
%4  Stopped at breakpoint 1 in find_shortest at line 200 in file "tsp.cl"
200   {
%2  Stopped at breakpoint 1 in find_shortest at line 200 in file "tsp.cl"
200   {
%5  Stopped at breakpoint 1 in find_shortest at line 200 in file "tsp.cl"
200   {
%3  Stopped at breakpoint 1 in find_shortest at line 200 in file "tsp.cl"
200   {
```

```
Linda initialization complete.
```

Figure 7: The ldbx window. Processes 2, 3, 4 and 5 have reached a breakpoint.

ging approaches in the future. Instead of a single, ever-present window for the low-level debugging of all processes, there could be an individual window for each process which pops up only when there is a specific low-level debugging task (e.g. to display the value of a program variable). While clicking the process icon with the left mouse button would open a window for displaying the text of the last tuple space operation, clicking it with the right button would open a window for the low-level debugging of that process. With the low-level debugger integrated in this way, TupleScope would come closer to implementing the levels-of-abstraction approach to debugging.

4.5 *The Postmortem Version of TupleScope*

Our discussion has thus far been limited to the ***run-time*** version of TupleScope, which provides monitoring and debugging capabilities for a C-Linda program as it is executing. Our implementation of the run-time version of TupleScope requires that the host machine be a shared-memory machine, or, in the case of uniprocessor machines, provide a simulation of shared memory. Besides being used to implement tuple space, shared memory is used for communicating monitoring information between the individual application program processes and the monitoring process. The ***postmortem*** version of TupleScope was developed to address the need for monitoring and debugging tools for implementations of

C-Linda on distributed-memory machines, such as the Intel iPSC/2 hypercube [Bjo89], or a local area network of workstations [AB89]. The postmortem version of TupleScope provides only an after-the-fact replay of a Linda program, or more precisely of the tuple space operations performed in a Linda program. Yet it has proven to be useful not just for debugging distributed-memory programs but also for providing additional flexibility when monitoring and debugging programs on shared-memory machines.

The postmortem version of TupleScope does not make use of an auxiliary monitoring process. The bottleneck that can result with the run-time version when the various application processes simultaneously request servicing by the monitoring process is eliminated. Instead of writing the information needed for monitoring to shared memory and waiting for a monitoring process to retrieve and process it, the postmortem version of TupleScope has each application process immediately write this information to a per process file. Tuple data, generated by the **out** operation, are written to separate files. Since output to the files is buffered, the file system overhead incurred is not prohibitive.

After the program has executed, the files generated during execution need to be reduced to a single file in which the various output streams of tuple space operations are merged into a single stream of tuple space operations that can be executed sequentially. The utility that accomplishes this, **par_order**, uncovers the partial ordering that is implicit in these per process data streams, and generates a single stream based on this ordering. **par_order** depends on each tuple having a unique number, known to every process dealing with the tuple, so that each **in** or **rd** operation that succeeds in getting a tuple can be associated with the correct tuple.

The method used by **par_order** to create a partial ordering of tuple space operations can be described briefly. Data which encode tuple space operations are read sequentially from the files produced by the Linda processes. They are immediately written to the file of the merged operation streams unless the partial ordering requires that the tuple space operation data of another process be written first. In that case, writing the operation data and reading the next data from the process file are delayed until the logically antecedent operation is found. Tuple space operations are identified in part by the tuples associated with them. **par_order** consequently needs to maintain a table of the tuples associated with the operations that it has processed. It also relies on information provided at run time concerning the blocking or unblocking of **in** and **rd** operations.

Using a single merge file as a source of tuple space event data and multiple files for the associated tuple data, the postmortem version of TupleScope reconstructs an execution sequence of a Linda program in its tuple space dimension. The postmortem replay has no requirement for shared memory or for any other parallel computing resources since it executes as a single process.

References

[AB89] M. Arango and D. Berndt. TSnet: A Linda Implementation for Networks of Unix-based Computers. Research Report 739, Yale University Department of Computer Science, August 1989.

[ACM89] ACM. *SIGPLAN Notices*, January 1989. Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging.

[Ber] P. Bercovitz. TupleScope User's Guide. Available from the Yale University Dept. of Computer Science.

[Bjo89] R. Bjornson. Experience with Linda on the iPSC/2. Research Report 698, Yale University Department of Computer Science, March 1989.

[Car87] N. Carriero. Implementation of Tuple Space Machines. Research Report 567, Yale University Department of Computer Science, December 1987. Also a 1987 Yale University PhD Thesis.

[CG89a] N. Carriero and D. Gelernter. Coordination Languages and their Significance. Research Report 716, Yale University Department of Computer Science, July 1989.

[CG89b] N. Carriero and D. Gelernter. Linda in Context. *Comm. ACM*, 32(4):444–458, April 1989.

[Gel82] D. Gelernter. *An Integrated Microcomputer Network for Experiments in Distributed Processing*. PhD thesis, State University of New York at Stony Brook, Stony Brook, New York, 1982. Department of Computer Science.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Gel89] D. Gelernter. Multiple tuple spaces in Linda. In *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE 89)*, volume II, pages 20–27, 1989.

[MH89] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.