Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-88-34

1988-10-01

Implementing a Shared Dataspace Language on a Message-Based Multiprocessor

Authors: Gruia-Catalin Roman and Kenneth C. Cox

The term shared dataspace refers to the general class of models and languages in which the principal means of communication is a common, content-addressable data structure called a dataspace. This paper reports on progress we have made toward the development of prototype implementation of a shared dataspace language, Swarm, on a hypercube multiprocessor. The paper includes an informal overview of the Swarm language, describes the design organization of a transaction processing system which forms the kernels of a Swam implementation, and explains the algorithms implementing a subset of Swarm embedded in the language C.

Follow this and additional works at: http://openscholarship.wustl.edu/cse_research

Recommended Citation

Roman, Gruia-Catalin and Cox, Kenneth C., "Implementing a Shared Dataspace Language on a Message-Based Multiprocessor" Report Number: WUCS-88-34 (1988). *All Computer Science and Engineering Research.*
http://openscholarship.wustl.edu/cse_research/791

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

IMPLEMENTING A SHARED DATASPACE LANGUAGE ON A MESSAGE-BASED MULTIPROCESSOR

Gruia-Catalin Roman and Kenneth C. Cox

WUCS-88-34

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

To appear in the Proceedings of the 5th International Workshop on Software Specification and Design, May 1989.

IMPLEMENTING A SHARED DATASPACE LANGUAGE ON A MESSAGE-BASED MULTIPROCESSOR

Gruia-Catalin Roman and Kenneth C. Cox

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130

ABSTRACT

The term *shared dataspace* refers to the general class of models and languages in which the principal means of communication is a content-addressable data structure called a dataspace. This paper reports on progress we have made toward the development of a prototype implementation of a shared dataspace language, *Swarm*, on a hypercube multiprocessor. The paper includes an informal overview of the Swarm language, describes the design organization of a transaction processing system which forms the kernel of a Swarm implementation, and explains the algorithms implementing a subset of Swarm embedded in the language C.

1. INTRODUCTION

Concurrent programming languages may be divided into four broad categories depending on the nature of the communication mechanisms they employ. *Shared variables* as used in Concurrent Pascal[5], for example, allow processes to communicate by reading and writing (directly or indirectly via monitors) the values of a fixed set of variables whose names are known to the communicating processes. *Message-based communication* requires all information sharing to take place by sending and receiving messages in accordance with some predefined protocol. CSP[8] and Actor languages[3] are representative of this category. *Remote operations*, such as the remote procedure call used in Ada[1], permit a process to invoke operations associated with some other process. The parameters and returned values represent the information shared by the processes involved in the exchange. The last category we choose to call *shared dataspace*. It is comprised of languages in which processes have access to a common, content-addressable data structure (typically a set of tuples) whose components may be examined, inserted, and deleted. Associons[10], Linda[4], SDL[11], Transaction Networks[9], Swarm[12], and some artificial intelligence languages such as OPS5[6] belong here.

During the past year our research group has embarked on a systematic study of the shared dataspace paradigm. The main vehicle for this investigation is the language *Swarm*. The design of Swarm takes a minimalist approach and provides a relatively small number of constructs which are believed to be at the core of a large class of shared dataspace languages and are helpful in the development and analysis of concurrent computations. In Swarm, a variety of computing styles have been brought together under a single unified model. Programming convenience has been achieved by the availability of powerful queries over the dataspace and by the ease with which unstructured and unbounded problems may be approached. The opportunities for temporal and spatial decoupling of computations characteristic of shared dataspace languages have been strengthened and enhanced. On-going work on a proof system for shared dataspace languages and on declarative visualization techniques promise to contribute to increased analyzability of shared dataspace programs.

This paper reports on progress we have made toward the development of a prototype implementation of Swarm on a hypercube multiprocessor. The implementation consists of a Swarm kernel designed to support dataspace transactions on a message-passing architecture and of a small subset of Swarm embedded in the language C, henceforth called Swarm/C. The Swarm kernel has additional capabilities beyond those required for Swarm/C; these additional capabilities will be used to extend the scope of our implementation.

The paper consists of four parts. Section 2 provides an informal overview of the Swarm language. Section 3 describes the Swarm kernel and its implementation on the NCUBE[2] hypercube multiprocessor. Section 4 defines the current Swarm/C language subset. Section 5 discusses the implementation of the Swarm/C subset using the kernel.

2. THE SWARM LANGUAGE

The purpose of this section is to give an introduction to the shared dataspace concept and some basic familiarity with the Swarm language and its computational model.‡ In Swarm the state of a computation is represented by the contents of the *dataspace*, a set of content-addressable entities. The dataspace is partitioned into three subsets: the *tuple space*, a finite set of data *tuples*; the *transaction space*, a finite set of *transactions*; and the *synchrony relation*, which is not discussed in this paper. An element of the dataspace is a pairing of a *type* name with a sequence of *values*. In addition, a transaction has an associated behavior specification.

The execution of a transaction is modeled as a transition between dataspaces. An executing transaction examines the dataspace, then deletes itself from the transaction space and modifies the dataspace by inserting and deleting tuples and by inserting (but not deleting!) other transactions. A Swarm program begins executing from a valid initial dataspace and continues until the transaction space is empty. On each execution step a transaction is chosen *nondeterministically* from the transaction space and executed. The transaction selection is *fair* in the sense that each transaction in the space will eventually be chosen.

Before explaining the syntax and semantics of programs, we need to introduce a few frequently used basic constructs. A constructor notation is used to form a set of entities and

‡ The language overview presented in this section is an abridged version of a Swarm language description which appeared first in Roman, G.-C. and Cunningham, H. C., "A Shared Dataspace Model of Concurrency — Language and Programming Implications," Technical Report WUCS-88-33, Department of Computer Science, Washington University, Saint Louis, MO 63130.

apply an operator to the elements of the set. It has the form:

[*operator variables* : *domain* : *operands*]

The *operator* field is a commutative and associative operator such as ∃, ∀, Σ, and Π. The *operands* field is a list of operands (separated by semicolons) compatible with the operator. The *variables* field is a (possibly null) list of bound variables whose scopes are delimited by the brackets. An instance of the constructor corresponds to a set of values for the bound variables such that the *domain* predicate is satisfied. (If the domain is blank, then the constant **true** is taken for the predicate.) The operator is applied to the set of operands corresponding to all instances of the constructor; if there are no instances, then the result of the constructor is the identity element of the operator.

Swarm predicates are first-order logical expressions constructed in the usual manner. Predicates can examine the dataspace. For example, the predicate *is_labeled*(17,λ), where λ is a variable, examines the dataspace for an element of type *is_labeled* having two components, the first being the constant 17 and the second being an arbitrary value. If such an element exists, the predicate succeeds and the value of the second component of the matched element is bound to the variable λ.

A special form of the constructor, called a *generator*, is used to form a set of entities, e.g., a set of tuples to be inserted into the tuple space. The operator fields of generators are blank; the domain predicates are not allowed to examine the dataspace.

A Swarm program consists of five sections: a program header, optional constant and macro definitions, tuple type declarations, transaction type declarations, and a dataspace initialization section. The syntax and semantics of these program sections are given below. Figure 1 shows a simple Swarm program to label each pixel in equal-intensity regions of a digital image with the highest coordinate in the region.

Program header. The program header associates a name with the program and optionally defines a set of program parameters. An invocation of the program with specific argument values causes the substitution of the values for the parameter names throughout the program's body. The argument values must satisfy the constraint predicates given in the program header. The program in Figure 1 is named *RegionLabel1*; it has parameters *Rows, Cols, Lo, Hi,* and *Intensity* constrained as indicated. *Intensity* is an array of input values indexed by the pixel coordinates.

Definitions. The optional Swarm definitions section allows the programmer to introduce named constants and "macros" into a program. For example, the *RegionLabel1* program in Figure 1 defines predicates *Pixel(P)* and *P* **neighbors** *Q*. These predicates allow the other sections to be expressed in a more concise and readable fashion.

Tuple types. The tuple types section declares the types of tuples that can exist in the tuple space. Each tuple type declaration defines a set of tuple *instances* that can be examined, inserted, and deleted by the program. In Figure 1 tuple type *has_label* pairs a pixel with a label; *has_intensity* pairs a pixel with its intensity value.

Transaction types. The transaction types section declares the types of transactions that can exist in the transaction space. Each transaction type declaration defines a set of transaction *instances* that can be executed, inserted, or examined by a program. The program in Figure 1 declares a

```
program RegionLabel1 ( Rows, Cols, Lo, Hi, Intensity :
        1 ≤ Rows, 1 ≤ Cols, Lo ≤ Hi,
        Intensity(ρ: Pixel(ρ)),
        [∀ ρ : Pixel(ρ) : Lo≤Intensity(ρ)≤Hi] )

definitions
    [ P, Q, L ::
    Pixel(P) ≡ [∃ x,y : P = (x,y) :
        1 ≤ x ≤ Cols, 1 ≤ y ≤ Rows ] ;
    P neighbors Q ≡ Pixel(P), Pixel(Q), P ≠ Q,
        [∃ x,y,a,b : P=(x,y), Q=(a,b) :
                a−1 ≤ x ≤ a+1, b−1 ≤ y ≤ b+1],
        [∃ ι :: has_intensity(P,ι), has_intensity(Q,ι)] ;
    P is_labeled L ≡ has_label(P,L)
    ]

tuple types
    [ P, L, I : Pixel(P), Pixel(L), Lo ≤ I ≤ Hi :
        has_label(P,L) ;
        has_intensity(P,I)
    ]

transaction types
    [ P : Pixel(P) :
    Label(P) ≡
            ρ,λ1,λ2 : P is_labeled λ1†, ρ is_labeled λ2,
                ρ neighbors P, λ1 < λ2
                    → P is_labeled λ2
        ‖   true    → Label(P)
    ]

initialization
    [ P : Pixel(P) :
        has_label(P,P), has_intensity(P,Intensity(P)),
        Label(P)
    ]

end
```

Figure 1: Nonterminating Region Labeling in Swarm

transaction type *Label(P)*.

The body of a transaction instance consists of a sequence of subtransactions connected by the ‖ operator:

variable_list$_1$: *query*$_1$ → *action*$_1$
‖ ...
‖ *variable_list*$_n$: *query*$_n$ → *action*$_n$

Each subtransaction definition consists of three parts: the *variable_list*, a comma-separated list of variable names, the *query*, an existential predicate over the dataspace, and the *action* which defines changes to be made to the dataspace. If the variable list is null, then the colon that separates it from the query may be omitted.

A subtransaction's action specifies sets of tuples to insert and delete and transactions to insert. Syntactically, an action consists of dataspace insertion and deletion operations separated by commas. In the action of a subtransaction, the notation *name* (*values*) specifies that a tuple or transaction of the type *name* is to be inserted into the dataspace; a † appended to a tuple specifies that the tuple is to be deleted if it is present in the dataspace. Generators may be used to

specify groups of insertions or deletions.

As a convenience, tuple deletion may be specified in the query by appending the symbol † to a dataspace predicate. *name*(*pattern*)† indicates that the matching tuple in the tuple space is to be deleted if the entire query succeeds. Any variables appearing in the *pattern* must be defined in the *variable_list* of the subtransaction (not in a constructor nested inside the query).

A subtransaction is executed in three phases: query evaluation, tuple deletions, and tuple and transaction insertions. Evaluation of the query seeks to find values for the subtransaction variables that make the query predicate true with respect to the dataspace. If the query evaluation succeeds, then the dataspace deletions and insertions specified by the subtransaction's action are performed using the values bound by the query. If the query fails, then the action is not executed. The subtransactions of a transaction are executed synchronously: queries are evaluated first, then the indicated tuple deletions are performed, and finally the indicated tuple and transaction insertions are done.

Initialization. By default, both the tuple and transaction spaces are empty. The initialization section establishes the dataspace contents that exist at the beginning of a computation. The section consists of a sequence of initializers separated by semicolons; each initializer is like a subtransaction's action in syntax and semantics. Since the null computation is not very interesting, at least one transaction must be established at initialization.

3. SWARM KERNEL

The model underlying Swarm has properties that make it attractive for expressing large-scale concurrent computations and for investigating the visualization of concurrent computations. These capabilities, however, come at the expense of increased implementation difficulty imposed by the requirement for atomic execution of complex transactions. Although the problems which arise are not new and there is a large body of literature on implementing such database transactions, the low-level granularity of the dataspace and the large number of concurrent transactions demand different kinds of solutions and a much greater reliance on highly parallel machines.

Our investigation into realization strategies for Swarm is following two parallel paths: the design of a shared dataspace machine whose only primitive operation is pattern matching and the development of a multiprocessor-based transaction system implementation. This section provides an overview of the design of the Swarm kernel which supports the transaction system. The first part gives an overview of the kernel design. The second part is a discussion of how this design is realized on the NCUBE, a hypercube-class message-passing multiprocessor developed by the NCUBE Corporation.

3.1. Kernel Overview

We want to be able to implement the kernel on a variety of message-based multiprocessors, including ones which do not support multiprogramming. In such a system, a processor cannot both support user application programs and service dataspace requests from other programs without greatly re-writing the user program. The transaction system therefore distinguishes between *user processes* and *dataspace processes*. The dataspace processes are responsible for storing tuples and handling dataspace requests from user processes. If the system permits, a single processor can run both user and dataspace processes; otherwise, the processors are partitioned into two groups, one used for user processes and one for dataspace processes. We use the term *site* to refer to a processor which is running a dataspace process.

At each site tuples are stored in a list. A position is assigned to each tuple when it is asserted; all positions are unique and increase by order of assertion time. The tuple position and site together form the tuple identifier, which is thus unique over the entire system.

The kernel acts primarily as a manager of *dialogs*. A dialog is the interaction between a single user process and a single site. A typical dialog consists of either two or three messages. The user program first issues a *request*. At some later time the site performs the request and returns a *reply*. The reply may complete the dialog, or the user process may be required to send a *confirmation*. These actions occur asynchronously, so the user process is free to continue with other processing after issuing a request.

A *dialog-id* is a type of object which identifies a dialog. Any object of type dialog-id is unique over the entire transaction system. When a user program issues a request, it receives a dialog-id. Dialogs may request operations on the dataspace, or may request bounds information from the site. The bounds information is simply a pair of integers containing the highest and lowest positions currently assigned to tuples at the site. This capability has been included for implementation of backtracking searches.

The supported dataspace operations are assert (add a tuple), query (examine a tuple), retract (remove a tuple), and modify (change a tuple). In the case of query, retract, and modify requests a *pattern* may be specified to be matched. The pattern includes constraints on the tuple data and the tuple identifier. When such a request is processed by a site the list of tuples at the site is searched in order until a match is found or there are no more tuples. A request can also specify a particular position at which the search is to start.

Matching requests may be issued in either *immediate* or *pending* operation-mode (several other modes are described below). In immediate mode, the site scans the tuples and replies; if no matching tuple is found, the reply contains a failure indication. In pending mode, the tuples are scanned; if no matching tuple is found, the site retains the dialog request until a matching tuple becomes available. When such a tuple becomes available, the request is performed and the reply is made. A pending request may be *released*. This action converts it to the immediate form, causing it to reply even if no matching tuple is available.

Any request may be *aborted*. If the site has already performed the request and sent a reply, the operation is not undone and the abort fails. If the abort is successful, the site returns a failure reply. Note in either case a reply is made and can be received by the user program. The *delete* operation behaves in the same manner but does not send a reply when the abort succeeds.

Tuples may be *locked* by requests. Each tuple has an associated *mark* which is modified by the request when it locks or unlocks the tuple. Several requests may hold locks on a tuple simultaneously. A request may be made in either *non-lock* or *lock* mode; lock mode specifies how the request modifies the lock. When a non-lock mode request matches a

tuple, the request is immediately performed and the reply is sent; the dialog is then complete. When a lock mode request matches a tuple, the locking is performed and a reply is sent. The user process must complete the dialog by sending a confirmation. A confirmation may be either a *cancel* or a *commit*. Both unlock the tuple, but the commit performs the request operation while the cancel does *not*.

The effect of the tuple lock is to *block* certain searches. When a request matches a tuple against its pattern, it tests the mark to determine if the tuple is locked. If the tuple is locked, the search is suspended and the tuple mark is modified. When the tuple is unlocked (by a cancel or commit) or altered (by a modify or retract), any requests suspended at the tuple are permitted to continue. The requests resume scanning starting with that tuple; if the tuple was retracted, they proceed to the next.

Requests may be issued in *block*, *non-block*, or *ignore* mode. Block mode causes the request to halt as described above. Non-block mode causes the request to skip the locked tuple; it does *not* re-examine the tuple if it reaches the end of the tuple list. Both block and non-block modes have parameters describing what the request considers to be a lock; block mode has additional parameters which describe how the tuple mark is modified if the request blocks. Ignore mode causes the request to ignore the tuple locking and proceed with the operation and reply.

The functions provided to user processes by the kernel are summarized in Table 1. The functions may be divided into five broad categories: functions which send requests initiate a dialog and return a dialog-id; functions which test for replies allow examination of the replies which have arrived (a search mechanism to examine all available replies is included); functions to receive replies get the data from the replies; functions to send confirmations are used when the dialog requires a cancel or commit; and functions which change the transaction status send abort, delete, and release requests. Space requirements prevent a complete explanation of these functions; a full discussion may be found in a recently released technical report[7].

3.2. Implementation of Kernel on NCUBE

The kernel is written in C under VERTEX, the NCUBE Corporation's proprietary operating system for the NCUBE node processors. This operating system supports message passing and message routing, but not multiprogramming. The nodes of an allocated NCUBE subcube are therefore partitioned into two sets, one set being available for user processes and one for dataspace processes. This partitioning is performed by a program running on the NCUBE host processor, which also loads the processes onto the dataspace nodes. This program is not part of the kernel so will not be discussed further.

The user processes are compiled with a library of functions which provide access to the Swarm kernel. The same library is used in the compilation of the dataspace process. This library provides many common structures to the two types of processes; for example, each process has a table which maps the logical sites to the physical NCUBE dataspace-node addresses. The two processes also have common protocols for formatting and transmitting messages using the VERTEX message-passing capabilities.

Functions to send requests (S-class):

dialog-id **Sassert**(*site, tuple, lock-mode*)
dialog-id **Squery**(*site, position, pattern, operation-mode, lock-mode, block-mode*)
dialog-id **Smodify**(*site, position, pattern, operation-mode, lock-mode, block-mode, modification*)
dialog-id **Sretract**(*site, position, pattern, operation-mode, lock-mode, block-mode*)
dialog-id **Sbounds**(*site*)

Functions to test for replies:

boolean **Tavail**(*dialog-id*)
dialog-id **Tfirstreply**(*trans-type*)
dialog-id **Tnextreply**(*trans-type*)

Functions to receive replies (R-class):

boolean **Rassert**(*dialog-id, site, position, status*)
boolean **Rquery**(*dialog-id, site, position, tuple, status*)
boolean **Rmodify**(*dialog-id, site, position, $tuple_1$, $tuple_2$, status*)
boolean **Rretract**(*dialog-id, site, position, tuple, status*)
boolean **Rbounds**(*dialog-id*, p_1, p_2)

Functions to send confirmations:

boolean **Scancel**(*dialog-id*)
boolean **Scommit**(*dialog-id*)

Functions to change transaction status:

boolean **Sabort**(*dialog-id*)
boolean **Sdelete**(*dialog-id*)
boolean **Srelease**(*dialog-id*)

Table 1: Swarm Kernel Functions

3.2.1. User package

When processing an S-class (send request) function, the user process first performs any necessary error-checking. It then generates a dialog-id and prepares a message containing an integer encoding the request type, the dialog-id, and the function arguments. The message is sent to the dataspace node corresponding to the site. The dialog-id and information about the dialog (including the request modes) are stored on the *outstanding list*. Finally, the dialog-id is returned.

Each user process has an internal counter for each of the dataspace nodes which it uses to generate the dialog-ids. System-wide uniqueness of dialog-ids is guaranteed by concatenating the counter with the address of the user process node and the site to create the dialog-id; the addresses and the counter can be recovered from the dialog-id.

When any of the package functions are called by the user program, the package first reads and queues any transaction-system messages (replies) which have arrived at the node. This transfers the messages from VERTEX operating system buffers into process buffers and reduces the possibility of a process blocking on a message send. The replies are then

processed. Any replies that do not correspond to outstanding dialogs are discarded (it being assumed that these dialogs have been aborted or deleted). The remaining replies are added to the *available list* and the entry on the outstanding list is changed to show the reply is available.

The **Tfirstreply** and **Tnextreply** functions implement a list-traversal mechanism on the available list. The **Tavail** function simply examines the outstanding list to see if the reply has arrived.

When an **R**-class (receive reply) function is performed, the outstanding list is examined. If the reply is available, the reply contents are copied into the function arguments and the reply is removed from the available list. If the request was issued in non-lock mode, it is removed from the outstanding list; otherwise the outstanding list entry is modified to indicate the reply has been received.

When an **Scancel** or **Scommit** is issued, the outstanding list is checked to ensure that the dialog reply has been received. If it has, the confirmation is sent and outstanding list entry is removed. Otherwise no action is taken.

When an **Sabort** is issued, the outstanding list is again checked. The function returns **false** if the dialog is not on the list. If it is on the list, the reply status is checked. If the reply has not yet arrived, an abort message is sent to the site and the outstanding list entry is modified to indicate it was aborted. If the reply has arrived, it and the dialog entry are discarded. The **Sdelete** function is similar but always removes the dialog's entry on the outstanding list.

When an **Srelease** is issued, the outstanding list is checked. The function returns **false** if the transaction is not on the list or is not pending. Otherwise the reply status is checked, and if the reply has not yet arrived a release message is sent to the site. Finally, the transaction mode is changed to immediate.

3.2.2. Dataspace processes

The dataspace process loops, continually reading messages from user processes and performing the indicated action, perhaps sending a reply as a result. When it receives a message the process first reads all messages queued in the NCUBE system and places them in an internal queue.

Any abort, delete, release, cancel or commit messages are processed first. This is primarily for efficiency, since any of these may permit removal of dialogs from the site and reduce storage requirements. An abort or delete may also remove a message from the queue and save processing time. Request messages from a particular user process are performed strictly in increasing order of dialog-id. If a request arrives out of order, it is held until the intervening requests are obtained and processed. The process keeps the largest dialog-id (as determined by the counter portion) that it has received from each of the user processes.

All requests are stored on a master *outstanding list*; each entry contains all the information associated with the requests. A number of other lists of requests are described below. These are implemented as lists of pointers to entries of the outstanding list, so the data for a request only appears once. These lists have the property that a request appears on at most one such list. A part of the outstanding list entry indicates which other list, if any, the request is on.

The tuples are stored in a *tuple list*. In addition to the tuple data, each list element contains the *position*, a *mark* used for locking, and a *block list* of requests. The process maintains a counter which is used to generate positions. When an *assert* is performed, the tuple is appended to the tuple list with a new position, a zeroed (unlocked) mark, and an empty block list.

To support pending requests, the process maintains a *pending list* of unserviced requests. Any request issued in pending mode which does not produce results when first processed is appended to the list. After performing any action which asserts, alters, or unlocks a tuple, the pending list is scanned for a request which matches the tuple and considers the tuple unlocked. If such a request is found, the request is removed from the list and performed. If the request had no affect on the tuple, the scanning of the pending list continues from the selected request. If the tuple is altered (including locking information) a new tuple results and the scanning of the pending list starts from the beginning. If the tuple is retracted, scanning halts.

When a release request is processed, the largest dialog-id is examined. If the request to be released has not been processed yet, the release is held until the request is received, at which time it is changed to immediate mode. If the request is not on the outstanding list, the release is ignored. If the request is on the pending list, it is removed from the pending and outstanding lists and a reply containing a failure indication is sent. (The pending-list scan algorithm maintains the invariant 'a transaction is on pending list *if and only if* no unlocked matching tuple exists'; a final scan of the tuple list is therefore not necessary.) Finally, if the request is blocked at a tuple, its status is changed to immediate and, when it is unblocked, it is processed as such.

When a request pattern matches a tuple, the results depend on the blocking-mode in which the transaction was issued. If the request was issued in *ignore* mode, or the request was issued in *non-block* or *block* mode and the tuple is not locked, then the appropriate action is taken (either the operation is performed or a lock is applied) and a reply is sent. If the request was issued in *non-block* mode and the tuple is locked, the request proceeds to the next tuple. If the request was issued in *block* mode and the tuple is locked, the request becomes blocked. Requests which are blocked at a tuple are appended to the tuple's *block list*. When the tuple is modified the requests on the block list are re-examined and may continue searching. Processing of the blocked requests occurs before examination of the pending list.

Tuple locking and request blocking are carried out using the tuple *mark*. The mark consists of two elements, a bitfield B and integer I. The request may examine both elements to determine if the tuple is locked. The effect of a block or lock is to set some bits of the appropriate mark's bitfield and increment the mark's integer by some amount, while undoing a block or lock clears the same bits of the bitfield and decrements the integer by the same amount. The test for blocking examines the mark and blocks if particular bits are set or if the integer is greater than some value. In this manner multiple types of locks may be supported.

Issuing a request in either block or non-block mode requires the specification of a test to determine if the tuple is locked. This test consists of two values, a bitfield b_t and an integer i_t. Block mode also requires the specification of an alteration to be applied to the mark if the request is blocked. This alteration consists of a bitfield b_b and integer i_b.

Issuing a request in lock mode requires the specification of an alteration to be applied to the mark if the request succeeds in locking the tuple. This alteration consists of a bitfield b_l and integer i_l.

Let $\otimes$ be bitwise AND, $\oplus$ be bitwise OR, and $\overline{b}$ be logical NOT (of b). Then the lock/block actions are:

A request considers a tuple locked if

$$((B \otimes b_l) \neq 0) \vee (I > i_l)$$

When a request blocks on a tuple, it performs

$$B \leftarrow B \oplus b_b \qquad I \leftarrow I + i_b$$

When a request is unblocked, it performs

$$B \leftarrow B \otimes \overline{b_b} \qquad I \leftarrow I - i_b$$

When a request locks on a tuple, it performs

$$B \leftarrow B \oplus b_l \qquad I \leftarrow I + i_l$$

When a request unlocks a tuple, it performs

$$B \leftarrow B \otimes \overline{b_l} \qquad I \leftarrow I - i_l$$

When a request applies a lock to a tuple, the request is placed on a *confirmation list*. When a confirmation message (cancel or commit) is received, this list is searched for the request; if it is not found the confirmation is ignored. If it is found, the request operation is applied to the tuple (for a commit), the tuple is unlocked, and the request is removed from the confirmation and outstanding lists. The block list for the tuple and the pending list are then processed.

When an abort is received, the largest dialog-id is used to determine if the request has been processed. If it has not, the abort is held until the request is received, at which time the failure reply is made. If the request has been processed, the outstanding list is examined. If the request is found, the failure reply is made; otherwise no action is taken.

4. SWARM/C DEFINITION

Many of the Swarm kernel features described above are designed for later implementation of powerful transaction-processing capabilities. To test the kernal and provide a working transaction system, the kernel is being used to implement a subset of Swarm embedded in the language C, which we call Swarm/C.

Swarm/C provides a limited number of atomic transaction types implemented as function calls in the C language. Most of the complexities of the Swarm kernel are hidden from the user—for example, sites are completely hidden and all necessary site manipulations are performed by the Swarm/C runtime library. In addition, Swarm/C provides some capabilities not used for Swarm; for example, transactions may specify a timeout period and are aborted if they exceed this period. Obviously this is not needed for Swarm, which does not include information about time. In addition, there is a large suite of functions designed to make the creation and manipulation of tuples and patterns easier.

Swarm/C transactions are subject to the following restrictions:

(1) Each transaction consists of only one subtransaction—synchronous execution of multiple subtransactions is eliminated.

(2) Transactions can refer to only one tuple in the tuple space and may involve only simple comparisons among variables and constants (in the query part). The need for implementing backtracking is thus eliminated and transaction atomicity is easily ensured.

(3) The single tuple referenced by the transaction may be examined, inserted, deleted, and modified. A modification involves a deletion followed by the insertion of a tuple of the same type in a single atomic operation.

(4) Transactions cannot refer in any way to the transaction space. The transaction space is essentially external to this implementation, because the transactions are issued from processes written in C.

Applying this set of restrictions to the region labeling program shown in Figure 1 leads to the version presented in Figure 2. The transaction type *Label(P)* had to be separated into two transactions each referring to a single tuple in the tuple space. Although the resulting solution is presented in Swarm, the reader is reminded that in Swarm/C one needs to create C programs which issue the transactions against the tuple space. To highlight this fact we placed a box around any actions which are actually performed in C without any support from the Swarm kernel.

```
program RegionLabel2 (Rows, Cols, Lo, Hi, Intensity :
        1 ≤ Rows, 1 ≤ Cols, Lo ≤ Hi,
        Intensity(ρ: Pixel(ρ)),
        [∀ ρ : Pixel(ρ) : Lo ≤ Intensity(ρ) ≤ Hi] )

definitions
    [ P, Q, L ::
    Pixel(P) ≡ [∃ x,y : P = (x,y) :
        1 ≤ x ≤ Cols, 1 ≤ y ≤ Rows ] ;
    P is_next_to Q ≡ Pixel(P), Pixel(Q), P ≠ Q,
        [∃ x,y,a,b : P=(x,y), Q=(a,b) :
            a−1 ≤ x ≤ a+1, b−1 ≤ y ≤ b+1];
    (P,I) is_labeled L ≡ has_intensity_and_label(P,I,L)
    ]

tuple types
    [ P, L, I : Pixel(P), Pixel(L), Lo ≤ I ≤ Hi :
        has_intensity_and_label(P,I,L)
    ]

transaction types
    [ P,Q,L,I : Pixel(P), Pixel(Q), Pixel(L), Lo≤I≤Hi :
    Check(P,Q) ≡
            ι,λ : (Q,ι) is_labeled λ
                    → skip
        ‖  [true    → Label(P,ι,λ), Check(P,Q) ;]
    Label(P,I,L) ≡
            λ : (P,I) is_labeled λ†, λ < L
                    → (P,I) is_labeled L
    ]

initialization
    [ P,Q : Pixel(P), Pixel(Q), Q is_next_to P :
        has_intensity_and_label(P,Intensity(P),P),
        [Check(P,Q)]
    ]
end
```

Figure 2: Nonterminating Region Labeling in Swarm/C

5. IMPLEMENTATION OF SWARM/C

The four Swarm/C functions which user processes may invoke in a C program are *assert*, *query*, *retract*, and *modify*. An invocation of any of these functions is called a *transaction*. Assert inserts a tuple in the dataspace, returning its identification; query examines a tuple and returns the contents; retract deletes a tuple, returning the contents; and modify is equivalent to a retract and assert, returning both the previous and new tuple values.

The query, modify, and retract transactions specify a *pattern* which is used to determine which tuple is affected by the operation. A transaction can either succeed or fail. An assert transaction can only fail if the tuple cannot be added (due to lack of storage); the other types can fail if no appropriate tuple to access or alter can be found.

Each transaction acts on a single tuple, and any transaction may access any tuple. A transaction appears to the user as a single atomic action; that is, each transaction modifies the dataspace without interference from other transactions. In particular, the *modify* transaction is not identical to a *retract* followed by an *assert* because the modify is an atomic action while other transactions can be interleaved between the retract and the assert.

Implementation of Swarm/C on top of the kernel functions involves three major additions: automatic distribution of tuples to sites, a method of representing and controlling the multiple dialogs involved in a Swarm/C transaction, and protocols for the dialogs involved in each of the transactions.

5.1. Tuple Distribution Method

The tuple site is determined automatically when the tuple is asserted. In order to add efficiency when a search of the dataspace is carried out, this assignment is done so the sites where a tuple may be found can be determined from a pattern matching the tuple. To do this, we require two functions; $F_t(tuple)$ produces the site at which a tuple is to be stored, and $F_p(pattern)$ produces a set of sites where a tuple matching the pattern may be stored. To be able to determine the possible sites from a pattern, we require $F_t(tuple) \in F_p(pattern)$ for *all* patterns which match the tuple.

Although this property of F_t and F_p appears difficult to realize, it is actually quite simple to develop appropriate functions. Assume that there are 2^n dataspace nodes (adaptations when there are not exactly 2^n dataspace nodes are simple). Each node is assigned a base-2 number consisting of n bits. Given a tuple to assert, an n-place vector of values is constructed from the tuple's structure and tuple data elements. A hashing function which produces 0 or 1 is then applied to each vector element, producing an n-digit binary number which is used as the node address.

A similar method is used for tuple matching. The pattern includes the structure of the tuple and some information about the values in the various fields. The same method as above is used to construct an n-digit number in which the structure and specified data element positions are 0 or 1 (as produced by the hashing function) and the unspecified positions are left empty. The set of possible tuple sites is generated by filling in the empty positions in all possible ways with 0 or 1.

The use of this method speeds up the process of searching for a tuple. However, it also requires that at all times $F_t(tuple)$ be the site where the tuple is stored. This means a tuple can only be modified if the new tuple has the same F_t as the old tuple, since tuples do not move once asserted.

The method used to guarantee this behavior uses the fact that tuples are vectors of data elements. A user program selects a number called the *cut*. Once a user process has asserted its first tuple, it cannot change the value of the cut. Any data elements whose position in the tuple is at or before the cut can be modified; data elements whose position is after the cut cannot be modified. The functions F_t and F_p use only information about positions after the cut.

5.2. Representation of Swarm/C Transactions

A Swarm/C transaction consists of a number of simultaneous dialogs with different sites. A transaction begins with the invocation of one of the Swarm/C functions and ends when the function returns; at this time all the dialogs have completed. No information needs to be kept from one transaction to the next.

With this information, design of the transaction representation is simple. Each transaction determines the sites with which it must interact, with *assert* using F_t to find the site where the tuple is to be stored and the others using F_p to find the sites that must be searched. Dialogs with each of these sites are initiated using the kernel functions.

The transaction then monitors the replies using the protocols described below. When the transaction selects a reply to be returned, it sends delete messages for any uncompleted dialogs and possibly a commit to complete the selected dialog. The transaction then returns. Any subsequent replies that arrive from the sites (due to transmission delays or two messages passing) are discarded by the user package as not belonging to active dialogs.

5.3. Protocols for Swarm/C Transactions

The assert transaction is the simplest, since a tuple is stored at exactly one site. The site for the tuple is selected using F_t and a request to assert the tuple is sent to that site. The assert request specifies no locking. The site either adds the tuple or fails (due to lack of storage) and replies accordingly. The site reply is passed on to the user.

The other transactions each permit pattern-searching. As such, they must interact with multiple sites. In the case of the query, the request is broadcast to all sites where tuples matching the pattern might be stored. The request specifies immediate mode, no locking, and block if some bit b is set. Each site eventually responds with a failure or with the contents of a matching tuple. If some site replies affirmatively, the transaction sends deletes to the sites with which it has active dialogs and returns the reply to the user. Otherwise, once all sites respond negatively the transaction returns a failure.

The retract and modify transactions are similar to one another. The request is broadcast to all sites where tuples matching the pattern might be stored. The request specifies immediate mode, lock by setting bit b, and block if some bit b is set. Each site eventually responds. If some site responds affirmatively, the transaction sends deletes to all other sites to halt the dialog and unlock any other tuples it holds and sends a commit to the affirmative site to complete the operation and unlock the tuple. Otherwise, once all sites respond negatively the transaction returns a failure.

We now show that with the above scheme all transactions will complete. Clearly the possibility of deadlock exists. For example, transaction T_1 lock tuple τ_1 and transaction T_2 lock tuple τ_2. If T_1 blocks at τ_2 (i.e., waits for the results of a dialog which is blocked at τ_2) and T_2 blocks at τ_1, a deadlock condition results.

We use the following scheme to avoid this deadlock. The sites are ordered by their logical numbering. A transaction has only one dialog with a particular site, and hence holds at most one lock on a tuple stored at the site. Once a transaction has a lock on a particular tuple it renounces all interest in any sites following that of the tuple and terminates the dialogs with these sites using delete messages. In particular, it releases any locks it has at these sites and does not wait for the results of dialogs with these sites.

With these conventions, deadlock situations cannot develop. Consider the previous example with transaction T_1 locking τ_1 and T_2 locking τ_2. If τ_1 and τ_2 have the same site, T_1 cannot be blocked at τ_2 because it only has one dialog with the site, specifically the one which caused τ_1 to be locked. If τ_1 and τ_2 are at different sites, either (the site of) τ_2 follows τ_1 or τ_1 follows τ_2. In the first case transaction T_1 has no interest in τ_2 so cannot be blocked by transaction T_2; the other case is symmetric and T_2 is not blocked. This property also holds for larger systems of transactions, so we deduce that among the currently active transactions at any time there is always some transaction which is not blocked.

We can also show absence of individual starvation – that is, any particular transaction will eventually complete – given two assumptions. The first is that when a tuple is unlocked, the requests blocked at that tuple are permitted to proceed before any other request can lock the tuple. This guarantees a group of blocked requests will not be continually re-blocked at the same tuple by new requests. The second assumption is that blocked requests are processed strictly in the order in which they became blocked. This guarantees that each request blocked at a particular tuple will eventually progress. Accordingly, every dialog will eventually complete (or be deleted) and hence every transaction will complete.

Since the retract and modify protocols described above obey the protocol for absence of deadlock, and the Swarm kernel follows the two unlocking rules for absence of individual starvation, we can conclude that every Swarm/C function will eventually complete.

6. CONCLUSIONS

The shared dataspace paradigm is a novel approach to specifying, visualizing, and reasoning about concurrent computations. Despite its attractive features, the paradigm's full potential will be realized only if we are able to demonstrate that reasonably efficient implementations of shared dataspace languages are feasible. Multiple implementations of Linda already exist and a Linda machine is currently being built. Swarm, however, is significantly more complex than Linda. (SWARM/C already exceeds the feature set included in C-Linda.)

The implementation study presented in this paper is designed to probe the extent to which today's multiprocessors are capable of supporting the processing requirements of shared dataspace languages in general and Swarm in particular. The multi-phased implementation effort is also intended to be a vehicle for making available the shared dataspace paradigm to developers of multiprocessor-based applications and to researchers interested in program visualization techniques.

Acknowledgments: This work was supported by the Department of Computer Science, Washington University, St. Louis, Missouri. The authors express their gratitude to Jerome R. Cox, department chairman, for his support and encouragement. We also thank Conrad Cunningham for his input regarding this paper and his contribution to the development of the shared dataspace concept. The NCUBE is one of the computational resources maintained by the Computer and Communications Research Center of Washington University.

7. REFERENCES

1. "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A-1983, American National Standards Institute, Inc., Washington, D.C. (January 1983).
2. "NCUBE Users Handbook," Version P2.1, NCUBE Corporation, Beaverton, Oregon (October 1987).
3. Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press, Cambridge, Massachusetts (1986).
4. Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," *Computer* **19**(8), pp. 26-34 (August 1986).
5. BrinchHansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* **1**(2), pp. 199-206 (1975).
6. Brownston, L., Farrell, R., Kant, E., and Martin, N., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming,* Addison-Wesley, Reading, Massachusetts (1985).
7. Cox, K. C. and Roman, G.-C., "A Transaction System for the NCUBE," Technical Report WUCS-88-31, Washington University, Department of Computer Science, St. Louis, Missouri (October 1988).
8. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM* **21**(8), pp. 666-677 (August 1978).
9. Kimura, T. D., "Visual Programming by Transaction Network," pp. 648-654 in *Proceedings of 21st Hawaii International Conference on System Sciences*, IEEE, Kona, Hawaii (January 1988).
10. Rem, M., "Associons: A Program Notation with Tuples Instead of Variables," *ACM Transactions on Programming Languages and Systems* 3(3), pp. 251-262 (July 1981).
11. Roman, G.-C., Cunningham, H. C., and Ehlers, M. E., "A Shared Dataspace Language Supporting Large-Scale Concurrency," *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 265-272. IEEE, June 1988.
12. Roman, G.-C. and Cunningham, H. C., "A Shared Dataspace Model of Concurrency — Language and Programming Implications," Technical Report WUCS-88-33, Washington University, Department of Computer Science, St. Louis, Missouri (October 1988).