Report Number: WUCS-90-10

1990-03-01

The Synchronic Group: A Concurrent Programming Concept and Its Proof Logic

Gruia-Catalin Roman and H. Conrad Cunningham

Swarm is a computational model which extends UNITY in three important ways: (1) UNITY's fixed set of variables is replaced by an unbounded set of tuples which are addressed by content rather than by name; (2) UNITY's static set of statements is replaced by a dynamic set of transactions; and (3) UNITY's static II-composition is augmented by dynamic coupling of transactions into synchronic groups. This paper overviews the Swarm model, introduced the synchronic group concept, and illustrates their use in the expression of dynamically structured programs. A UNITY-style programming logic is given for SWARM, the first axiomatic proof system... Read complete abstract on page 2.

The Synchronic Group: A Concurrent Programming Concept and Its Proof Logic

Gruia-Catalin Roman and H. Conrad Cunningham

Complete Abstract:

Swarm is a computational model which extends UNITY in three important ways: (1) UNITY's fixed set of variables is replaced by an unbounded set of tuples which are addressed by content rather than by name; (2) UNITY's static set of statements is replaced by a dynamic set of transactions; and (3) UNITY's static ll-composition is augmented by dynamic coupling of transactions into synchronic groups. This paper overviews the Swarm model, introduced the synchronic group concept, and illustrates their use in the expression of dynamically structured programs. A UNITY-style programming logic is given for SWARM, the first axiomatic proof system for a shared database language.

THE SYNCHRONIC GROUP: A CONCURRENT PROGRAMMING CONCEPT AND IT'S PROOF LOGIC
Abridged Version

Gruia-Catalin Roman and H. Conrad Cunningham

WUCS-90-10

March 1990

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

As appeared in the Proceedings of the 10th International Conference on Distributed Computing Systems, May 1990, pp. 142-149.

The Synchronic Group: A Concurrent Programming Concept and Its Proof Logic

Gruia-Catalin Roman† and H. Conrad Cunningham‡

† Dept. of Computer Science, Washington University, St. Louis, Missouri, U.S.A.
‡ Dept. of Computer & Info. Science, University of Mississippi, Oxford, Mississippi, U.S.A.

Abstract

Swarm is a computational model which extends UNITY in three important ways: (1) UNITY's fixed set of variables is replaced by an unbounded set of tuples which are addressed by content rather than by name; (2) UNITY's static set of statements is replaced by a dynamic set of transactions; and (3) UNITY's static ||-composition is augmented by dynamic coupling of transactions into *synchronic groups.* This paper overviews the Swarm model, introduces the synchronic group concept, and illustrates their use in the expression of dynamically structured programs. A UNITY-style programming logic is given for Swarm, the first axiomatic proof system for a *shared dataspace language.*

1 Introduction

Attempts to meet the challenges of concurrent programming have led to the emergence of a variety of models and languages. Chandy and Misra, however, argue that the fragmentation of programming approaches along the lines of architectural structure, application area, and programming language features obscures the basic unity of the programming task [3]. With the UNITY model, their goal is to unify seemingly disparate areas of programming with a simple theory consisting of a model of computation and an associated proof system.

They build the UNITY computational model upon a traditional imperative foundation, a state-transition system with named variables to express the state and conditional multiple-assignment statements to express the state transitions. Above this foundation, however, UNITY follows a more radical design: all flow-of-control and communication constructs have been eliminated from the notation. A UNITY program begins execution in a valid initial state and continues infinitely; at each step an assignment is selected nondeterministically, but fairly, and executed.

To accompany this simple but innovative model, Chandy and Misra have formulated an assertional programming logic which frees the program proof from the necessity of reasoning about execution sequences. Unlike most assertional proof systems, which rely on the annotation of the program text with predicates, the UNITY logic seeks to extricate the proof from the text by relying upon proof of program-wide properties such as invariants and progress conditions.

Swarm [8] is a model which extends UNITY by permitting content-based access to data, a dynamic set of statements, and the ability to prescribe and alter the execution mode (i.e., synchronous or asynchronous) for arbitrary collections of program statements. The Swarm model is our primary vehicle for study of the *shared dataspace paradigm* [7], a class of languages and models in which the primary means for communication among the concurrent components of a program is a common, content-addressable data structure called a *shared dataspace.* Elements of the dataspace may be examined, inserted, or deleted by programs. Linda [1], Associons [6], and production rule languages such as OPS5 [2] all follow the shared dataspace approach.

In designing Swarm, we merged the philosophy of UNITY with the methods of Linda. Swarm has a UNITY-like program structure and computational model and Linda-like communication mechanisms. We partition the Swarm dataspace into three subsets: a tuple space (a finite set of data tuples), a transaction space (a finite set of transactions), and a synchrony relation (a symmetric relation on the set of all possible transactions). We replace UNITY's fixed set of variables with a set of tuples and UNITY's fixed set of assignment statements with a set of transactions.

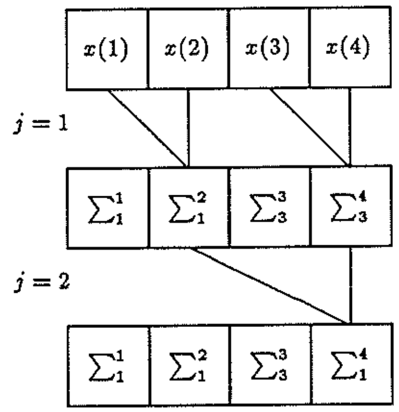
A Swarm transaction denotes an atomic transformation of the dataspace. It is a set of concurrently executed query-action pairs. A query consists of a predicate over the dataspace; an action consists of a group of deletions and insertions of dataspace elements. Instances of transactions may be created dynamically by an executing program.

A Swarm program begins execution from a specified initial dataspace. On each execution step, a transaction is chosen nondeterministically from the transaction space and executed atomically. This selection is fair in the sense that every transaction in the transaction space at any point in the computation will eventually be chosen. An executing transaction examines the dataspace and then, depending upon the results of the examination, can delete tuples (but not transactions) from the dataspace and insert new tuples and transactions into the dataspace. Unless a transaction explicitly reinserts itself into the dataspace, it is deleted as a by-product of its execution. Program execution continues until there are no transactions in the dataspace.

The synchrony relation feature adds even more dynamism and expressive power to Swarm programs. It is a relation over the set of possible transaction instances. This relation may be examined and modified by programs in the same way as the tuple and transaction spaces are. To accommodate the synchrony relation, we extend the program execution model in the following way: whenever a transaction is chosen for execution, all transactions in the transaction space which are related to the chosen transaction by (the closure of) the synchrony relation are also chosen; all of the transactions that make up this set, called a synchronic group, are executed as if they comprised a single transaction.

By enabling asynchronous program fragments to be coalesced dynamically into synchronous subcomputations, the synchrony relation provides an elegant mechanism for structuring concurrent computations. This unique feature facilitates a programming style in which the granularity of the computation can be changed dynamically to accommodate structural variations in the input. This feature also suggests mechanisms for the programming of a mixed-modality parallel computer, i.e., a computer which can simultaneously execute asynchronous and synchronous computations. Perhaps architectures of this type could enable both higher performance and greater flexibility in algorithm design.

In this paper we show how to add this powerful capability to Swarm without compromising our ability to formally verify the resulting programs. The presentation is organized as follows. Section 2 reviews the basic Swarm notation. Section 3 introduces the notation for the synchrony relation and discusses the concept of a synchronic group. Section 4 illustrates the use of synchronic groups by means of a program for labeling regions in an image unbounded on one side. Section 5 reviews a UNITY-style assertional programming logic for Swarm without the synchrony relation and then generalizes the logic to accomodate synchronic groups.



```
$j$ : integer;
$x(i : 1 \leq i \leq N)$ : array of integer;
$\langle k : 1 \leq k \leq N :: x(k) := A(k) \rangle$;
$j := 1$;
do $j < N \longrightarrow$
      $\langle \| \ k : 1 \leq k \leq N \wedge k \bmod (j * 2) = 0 ::$
            $x(k) := x(k - j) + x(k) \rangle$;
      $j := j * 2$
od
```

Figure 1: Array Summation

2 Basic Swarm Notation

By choosing the name *Swarm* for our shared dataspace programming model, we evoke the image of a large, rapidly moving aggregation of small, independent agents cooperating to perform a task. In this section we introduce a notation for programming such computations. We first present an algorithm expressed in a familiar imperative notation—a parallel dialect of Dijkstra's Guarded Commands [5] language. We then construct a Swarm program with similar semantics.

The algorithm given in Figure 1 sums an array of N integers. For simplicity, we assume that N is a positive power of 2. In the program fragment, A is the "input" array of integers to be summed and x is an array of partial sums used by the algorithm. Both arrays are indexed by the integers 1 through N. At the termination of the algorithm, $x(N)$ is the sum of the values in the array A. The loop computes the sum in a tree-like fashion as shown in the diagram: adjacent elements of the array are added in parallel, then the same is done for the resulting values, and so forth until a single value remains. The construct

$$\langle \| \ k : \textit{predicate} :: \textit{assignment} \rangle$$

is a parallel assignment command. The *assignment* is executed in parallel for each value of k which satisfies the *predicate*; the entire construct is performed as one atomic action.

Swarm is a shared dataspace programming model. Instead of expressing a computation in terms of a group of named variables, Swarm uses a set of *tuples* stored in a *dataspace*. Each tuple is a pairing of a type name with a finite sequence of values; a program accesses a tuple by its content—type name and values—rather than by a specific name or address. Swarm programs compute by deleting existing tuples from, and inserting new tuples into, the dataspace. The *transactions* which specify these atomic dataspace transformations consist of a set of *query-action* pairs executed in parallel. Each query-action pair is similar to a production rule in a language like OPS5 [2].

How can we express the array-summation algorithm in Swarm? To represent the array x, we introduce tuples of type x in which the first component is an integer in the range 1 through N, the second a partial sum. We can express an instance of the array assignment in the do loop as a Swarm transaction in the following way:

$$\begin{aligned} &v1, v2 : x(k-j, v1), x(k, v2) \\ &\qquad \longrightarrow x(k, v2)\dagger, x(k, v1+v2) \end{aligned}$$

Above, the part to the left of the $\longrightarrow$ is the query; the part to the right is the action. The identifiers $v1$ and $v2$ designate variables that are local to the query-action pair. (For now, assume that j and k are constants.)

The execution of a Swarm query is similar to the evaluation of a clause in Prolog [9]. The above query causes a search of the dataspace for two tuples of type x whose component values have the specified relationship—the comma separating the two tuple predicates is interpreted as a conjunction. If one or more solutions are found, then one of the solutions is chosen nondeterministically and the matched values are bound to the local variables $v1$ and $v2$ and the action is performed with this binding. If no solution is found, then the transaction is said to fail and none of the specified actions are taken.

The action of the above transaction consists of the deletion of one tuple and the insertion of another. The $\dagger$ operator indicates that the tuple $x(k, v2)$, where $v2$ has the value bound by the query, is to be deleted from the dataspace. The unmarked tuple form $x(k, v1 + v2)$ indicates that the corresponding tuple is to be inserted. Although the execution of a transaction is atomic, the effect of an action is as if all deletions are performed first, then all insertions.

The parallel assignment command of the algorithm can be expressed similarly in Swarm:

```
program $ArraySum(N, A:$
    $[\exists p : p > 0 :: N = 2^p],\ A(i : 1 \le i \le N))$
tuple types
    $[i, s : 1 \le i \le N :: x(i, s)]$
transaction types
    $[j : j > 0 ::$
      $Sum(j) \equiv$
        $[\| \ k : 1 \le k \le N \wedge k \bmod (j*2) = 0 ::$
              $v1, v2 : x(k-j, v1)\dagger, x(k, v2)\dagger$
                  $\longrightarrow x(k, v1+v2)\ ]$
        $\|$    $j*2 < N \longrightarrow Sum(j*2)$
    $]$
initialization
    $Sum(1);\ [i : 1 \le i \le N :: x(i, A(i))]$
end
```

Figure 2: A Swarm Program

$$\begin{aligned} &[\| \ k : 1 \le k \le N \wedge k \bmod (j*2) = 0 :: \\ &\quad v1, v2 : x(k-j, v1), x(k, v2) \\ &\qquad \longrightarrow x(k, v2)\dagger, x(k, v1+v2)\,] \end{aligned}$$

We call each individual query-action pair a *subtransaction* and the overall construct a *transaction*. As with the parallel assignment, the entire transaction is executed atomically. The cumulative effect of executing a transaction is as if the subtransactions are executed synchronously: all queries are evaluated first, then the indicated tuples are deleted, and finally the indicated tuples are inserted.

Like data tuples, transactions are represented as tuple-like entities in the dataspace. A transaction instance has an associated type name and a finite sequence of values called parameters. A subtransaction can query and insert transaction instances in the same way as data tuples are, but transactions cannot be explicitly deleted. Implicitly, a transaction is deleted as a by-product of its own execution—regardless of the success or failure of its component queries.

Two aspects of the array-summation program's do command have not been translated into Swarm—the doubling of j and the conditional repetition of the loop body. Both of these can be incorporated into a transaction. We define a transaction type called *Sum* as follows:

$$\begin{aligned} &Sum(j) \equiv \\ &\quad [\| \ k : 1 \le k \le N \wedge k \bmod (j*2) = 0 :: \\ &\qquad v1, v2 : x(k-j, v1), x(k, v2) \\ &\qquad\quad \longrightarrow x(k, v2)\dagger, x(k, v1+v2)\,] \\ &\quad \| \quad j*2 < N \longrightarrow Sum(j*2) \end{aligned}$$

Thus transaction $Sum(j)$, representing one iteration of the loop, inserts a successor which represents the next iteration.

For a correct computation, the Swarm array-summation program must be initialized with the following tuple space:

$$\{\ x(1, A(1)), x(2, A(2)), \cdots, x(N, A(N))\ \}$$

The initial transaction space consists of the transaction $Sum(1)$.

Since each x tuple is only referenced once during a computation, we can modify the *Sum* subtransactions to delete both x tuples that are referenced. A complete *ArraySum* program with this modification is given in Figure 2. (If a tuple in a query is marked by a dagger, then, if the overall query succeeds, the marked tuple will be deleted as a part of the action.)

```
program $ArraySumSynch(N, A :$
    $[\exists p : p > 0 :: N = 2^p],\ A(i : 1 \leq i \leq N))$
tuple types
    $[i, s : 1 \leq i \leq N :: x(i, s)]$
transaction types
    $[k, j : 1 \leq k \leq N, 1 \leq j < N ::$
      $Sum(k, j) \equiv$
                $v1, v2 : x(k - j, v1)\dagger, x(k, v2)\dagger$
                    $\longrightarrow x(k, v1 + v2)$
            $\|$   $j * 2 < N, k \bmod (j * 4) = 0$
                    $\longrightarrow Sum(k, j * 2)$
    $]$
initialization
    $[i : 1 \leq i \leq N :: x(i, A(i))];$
    $[k : 1 \leq k \leq N, k \bmod 2 = 0 :: Sum(k, 1)];$
    $[k, j : 1 \leq k < N, 1 \leq j < N ::$
        $Sum(k, j) \sim Sum(k + 1, j)];$
end
```

Figure 3: A Synchronic Group Program

3 Synchronic Groups

In our discussion so far we have ignored the third component of a Swarm program's state—the *synchrony relation.* The interaction of the synchrony relation with the execution mechanism provides a dynamic form of the $\|$ operator. The synchrony relation is a symmetric, irreflexive relation on the set of valid transaction instances. The reflexive transitive closure of the synchrony relation is thus an equivalence relation. When one of the transactions in an equivalence class is chosen for execution, then all members of the class which exist in the transaction space at that point in the computation are also chosen. This group of related transactions is called a *synchronic group.* The subtransactions making up the transactions of a synchronic group are executed as if they were part of the same transaction.

The synchrony relation can be examined and modified in much the same way as the tuple and transaction spaces can. The predicate

$$Sum(i) \sim Sum(j)$$

in the query of a subtransaction examines the synchrony relation for a transaction instance $Sum(i)$ that is directly related to an instance $Sum(j)$. Neither transaction instance is required to exist in the transaction space. The operator $\approx$ can be used in a predicate to examine whether transaction instances are related by the closure of the synchrony relation.

Synchrony relationships between transaction instances can be inserted into and deleted from the relation. The operation

$$Sum(i) \sim Sum(j)$$

in the action of a subtransaction creates a dynamic coupling between transaction instances $Sum(i)$ and $Sum(j)$ (where i and j must have bound values). If two instances are related by the synchrony relation, then

$$(Sum(i) \sim Sum(j))\dagger$$

deletes the relationship. Note that the closure relation $\approx$ can be examined, but that only the base synchrony relation $\sim$ can be directly modified. Initial synchrony relationships can be specified by putting appropriate insertion operations into the initialization section of the Swarm program.

Figure 3 shows a version of the array-summation program which uses synchronic groups. The subtransactions of $Sum(j)$ have been separated into distinct transactions $Sum(k, j)$ coupled by the synchrony relation. For each phase j, all transactions associated with that phase are structured into a single synchronic group. The computation's effect is the same as that of the earlier program.

4 Region Labeling

In this section we address the problem of labeling the equal-intensity regions of a digital image unbounded on one side. The image consists of pixels arranged on a grid with M rows and an infinite number of columns. An intensity (brightness) attribute is associated with each pixel. We identify the pixels by coordinates with x-values 1 or larger and y-values in the range 1 through M. Although the full image is assumed to extend to the right without

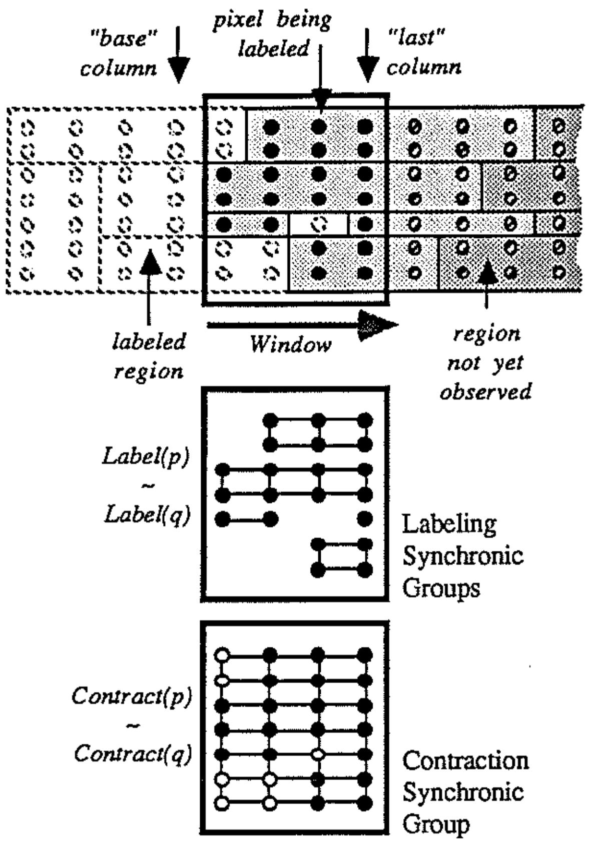


Figure 4: The Window Metaphor

```
program $Unbounded(M, Lo, Hi, Intensity:$
        $M \geq 1,\ Lo \leq Hi,\ Intensity(\rho : Pixel(\rho)),$
        $[\forall \rho : Pixel(\rho) :: Lo \leq Intensity(\rho) \leq Hi])$
definitions
  $[P, Q, L ::$
    $Pixel(P) \equiv$
      $[\exists c, r : P = (c, r) :: c \geq 1, 1 \leq r \leq M];$
    $adjacent(P, Q) \equiv$
      $Pixel(P), Pixel(Q), (0,0) < |P - Q| \leq (1,1);$
    $neighbors(P, Q) \equiv adjacent(P, Q),$
      $[\exists \iota :: has\_intensity(P, \iota),$
        $has\_intensity(Q, \iota)];$
    $on\_left(P) \equiv Pixel(P), [\exists r :: P = (1, r)];$
    $on\_right(P) \equiv$
      $Pixel(P), [\exists c, r : final(c) :: P = (c, r)];$
    $ONE \equiv (1, 0)$
  $]$
tuple types ...
transaction types ...
initialization ...
end
```

Figure 5: Region Labeling—Program Header

bound, we assume that the length (i.e., the number of columns intersected) of each connected region of equal-intensity pixels to be finite and bounded above by the constant *MaxLen*. (We do not allow a program to use this constant directly.)

We desire a program which labels the regions of unbounded images of this type. The program must not use an unbounded amount of space: the number of tuples and transactions existing at any point during the computation must be bounded above by some constant; the range of values of the integers used in the program must also be bounded. (Since Swarm does not have an input operation, we do not impose the bounded-values restriction on the "counter" used to record the current position in the input stream.)

To keep the number of tuples and transactions bounded, we adopt a sliding window metaphor for our solution to the problem. (See Figure 4.) The window is a contiguous group of columns from the image. At any point in the computation, the window contains all pixels currently being processed. The program stores information about these pixels in the dataspace. The computation begins with the window positioned over the leftmost (smallest x-coordinate) column of the image. As a computation proceeds, the window expands to the right—the column of the image immediately to the right of the window is inserted into the window when the pixels in that column are "needed." The program needs the new column when some region extends across all columns of the window. The window also contracts from the left—the leftmost column of the window is deleted when all pixels in the column have been "completed." A pixel is complete when all pixels in its region have been labeled with the region's label. (For convenience, we use the lexicographically smallest coordinates of a pixel in the region as the region's label.) The window thus slides across the image from left to right; the maximum width of the window is *MaxLen* $+ 1$.

For the size of the numbers used by the program to be bounded, the program cannot use the absolute coordinate system of the full image. Thus, for the pixels in the window, we adopt a new coordinate system—the program addresses pixels relative to the leftmost column of the window. When the program expands the window, all information inserted into the dataspace concerning the new pixels must use window-relative x-coordinates. When the program contracts the window, it must also modify all information concerning the pixels in the window to reflect the new coordinate system base.

The region labeling program uses three tuple types and three transaction types. The three tuple types are *has_label*, which pairs a pixel with a label; *has_intensity*, which pairs a pixel with its intensity value; and *final*, which records the x-coordinate of the rightmost column of the window. The three transaction types are *Label*, *Expand*, and *Contract*. The transactions of type *Label* carry out the labeling of the pixels of the image; transactions of type *Expand* and *Contract* implement the window expansion and contraction operations of the sliding window strategy. Note that the computation begins with the window positioned over a single column—the first column of the image. Figure 6 shows the details of these transaction definitions. Some of the predicates used in these definitions appear in Figure 5.

To organize the computation, we take advantage of the synchronic group feature of Swarm. For instance, we use a synchronic group to contract the window. The program creates a *Contract* transaction for each pixel in the window, either at initialization or when a new column is brought into the window by an *Expand* transaction, and links all of these transactions together into a synchronic group. When executed, this group simultaneously decrements the x-coordinates for all information recorded for each pixel in the window.

The program also uses synchronic groups of *Label* transactions to carry out the labeling of the regions and to detect when the labeling of a region is complete. The program creates a *Label* transaction for each pixel of the window, either at initialization or when a new column is brought into the window by an *Expand* transaction, and links the transactions for adjacent pixels of the same intensity into the same synchronic group. When one of these *Label* synchronic groups is executed, it either changes the labels of one or more pixels to a lower value or, when it detects that labeling of the region is complete, deletes all information concerning the region from the dataspace.

The *special* predicates OR, AND, NOR, and NAND, meaning *any*, *all*, *none*, and *not-all*, respectively, can be used in transaction queries. These special predicates examine the success status of all the simultaneously executed subtransaction queries which do not involve special predicates, i.e., the *regular* queries. For example, the predicate OR succeeds if any one of the regular queries in any transaction of the synchronic group also succeeds; NOR (not-or) succeeds if none of the regu-

```
$Label(P) \equiv$
        $\rho, \lambda 1, \lambda 2:$
          $has\_label(P, \lambda 1)\dagger, neighbors(P, \rho),$
          $has\_label(\rho, \lambda 2), \lambda 2 < \lambda 1$
              $\longrightarrow has\_label(P, \lambda 2)$
    $\|$     $on\_right(P) \longrightarrow$ skip
    $\|$     OR $\longrightarrow Label(P)$
    $\|$   $\iota, \lambda:$ NOR, $has\_intensity(P, \iota)\dagger,$
              $has\_label(P, \lambda)\dagger$
              $\longrightarrow$ skip
    $\|$     NOR
              $\longrightarrow [\, \rho : adjacent(P, \rho) ::$
                  $(Label(P) \sim Label(\rho))\dagger \,]$

$Expand(Next) \equiv$
    $\rho, \lambda, c : on\_right(\rho), has\_label(\rho, \lambda),$
          $on\_left(\lambda), final(c)\dagger$
      $\longrightarrow$
        $[\, r, \tau : 1 \leq r \leq M, \tau = (c+1, r) ::$
          $has\_intensity(\tau, Intensity((Next, r))),$
          $has\_label(\tau, \tau),$
          $Label(\tau),$
          $[\, \delta : adjacent(\tau, \delta), \delta \leq (c+1, M),$
            $Intensity(\tau) = Intensity(\delta) ::$
              $Label(\tau) \sim Label(\delta) \,],$
          $Contract(\tau),$
          $[\, \delta : adjacent(\tau, \delta), \delta \leq (c+1, M) ::$
            $Contract(\tau) \sim Contract(\delta) \,],$
          $final(c+1),$
          $Expand(Next+1)$
        $]$

$Contract(P) \equiv$
        $\iota : on\_left(P), has\_intensity(P, \iota)$
              $\longrightarrow$ skip
    $\|$     OR $\longrightarrow Contract(P)$
    $\|$   $c$ :NOR, $final(c)\dagger$
              $\longrightarrow final(c-1), Contract(P)$
    $\|$   $\iota$ :NOR, $has\_intensity(P, \iota)\dagger$
              $\longrightarrow has\_intensity(P-ONE, \iota)$
    $\|$   $\lambda$ :NOR, $has\_label(P, \lambda)\dagger$
              $\longrightarrow has\_label(P-ONE, \lambda-ONE),$
                  $Label(P-ONE)$
    $\|$ $[\|$ $\rho : adjacent(P, \rho) ::$
          NOR, $(Label(P) \sim Label(\rho))\dagger$
              $\longrightarrow Label(P-ONE)$
                  $\sim Label(\rho-ONE) \,]$
```

Figure 6: Region Labeling—Transactions

lar queries succeed. The evaluation of the special queries do not affect each other.

In *Label(P)*, for instance, OR forces the transaction to be reinserted when any *Label* transaction within P's region relabels a pixel or when there might be a portion of the region to the right of the processing window. The NOR, on the other hand, detects when the labeling of a region is complete and causes the tuples and synchrony relation entries for the region to be deleted. Similarly, in *Contract(P)*, the OR queries force the reinsertion of the entire *Contract* synchronic group whenever any transaction in the group detects a pixel in the first column of the window; when labeling of all pixels in the first column is complete, the NOR query leads to a one column shift in the x-coordinate of all entities present in the dataspace.

A proof of this program, using the logic presented in the next section, appears in [4].

5 Programming Logic

The Swarm computational model is similar to that of UNITY [3]; hence, a UNITY-style assertional logic seems appropriate. However, we cannot use the UNITY logic directly because of the differences between the UNITY and Swarm frameworks.

In this section we follow the notational conventions for UNITY in [3]. Properties and inference rules are written without explicit quantification; these are universally quantified over all the values of the free variables occurring in them. We use the notation $[t]$ to denote the predicate "transaction instance t is in the transaction space," TRS to denote the set of all possible transactions (not a specific transaction space), and *INIT* to denote the initial state of the program.

The proof rules for the subset of Swarm without the synchrony relation are given in [4]. We summarize them below. The Swarm programming logics have been defined so that the theorems proved for UNITY in [3] can also be proved for Swarm.

1. $\{p\}\ t\ \{q\}$.
   The "Hoare triple" means that, whenever the dataspace satisfies the precondition predicate p and transaction instance t is in the transaction space, all dataspaces which can result from execution of transaction t satisfy the postcondition predicate q.

2. p **unless** $q\ \equiv$
   $\langle \forall t : t \in \text{TRS} :: \{p \wedge \neg q\}\ t\ \{p \vee q\}\rangle$.
   This means that, if p is *true* at some point in the computation and q is not, then, after the next step, p remains *true* or q becomes *true*.

3. **stable** $p \equiv p$ **unless** **false**.
   This means that, if p becomes *true*, it remains *true* forever.

4. **invariant** $p \equiv (\mathit{INIT} \Rightarrow p) \wedge$ (**stable** p).
   Invariants are properties which are *true* at all points in the computation.

5. p **ensures** $q\ \equiv$
   (p **unless** q) $\wedge$
   $\langle \exists t : t \in \text{TRS} ::$
   $(p \wedge \neg q \Rightarrow [t]) \wedge \{p \wedge \neg q\}\ t\ \{q\}\rangle$.
   This means that, if p is *true* at some point in the computation, then (1) p will remain *true* as long as q is *false*, and (2) if q is *false*, there is at least one transaction in the transaction space which can, when executed, establish q as *true*. The "$p \wedge \neg q \Rightarrow [t]$" requirement generalizes the UNITY definition of **ensures** to accomodate Swarm's dynamic transaction space.

6. $p \longmapsto q$.
   This, read p *leads-to* q, means that, once p becomes *true*, q will eventually become *true*. (However, p is not guaranteed to remain *true* until q becomes *true*.) As in UNITY, the assertion $p \longmapsto q$ is *true* if and only if it can be derived by a finite number of applications of the following inference rules:

   - $\dfrac{p \text{ ensures } q}{p \longmapsto q}$
   - $\dfrac{p \longmapsto q,\ q \longmapsto r}{p \longmapsto r}$ (transitivity)
   - *For any set W,* (disjunction)
     $\dfrac{\langle \forall m : m \in W :: p(m) \longmapsto q\rangle}{\langle \exists m : m \in W :: p(m)\rangle \longmapsto q}$

7. **termination** $\equiv \langle \forall t : t \in \text{TRS} :: \neg[t]\rangle$.
   Unlike UNITY programs, Swarm programs can *terminate* when the transaction space is empty.

The logic we defined for Swarm programs may be generalized to accomodate synchronic groups. This involves the addition of a synchronic group rule and redefinition of the **unless** and **ensures** relations. The other elements of the logic are the same.

In [4], we define the "Hoare triple" for synchronic groups

$$\{p\}\ S\ \{q\}$$

to mean that, whenever the precondition p is *true* and S is a synchronic group of the dataspace, all dataspaces which can result from execution of group S satisfy postcondition q.

A key difference between this logic and the previous logic is the set over which the properties must be proved. For example, the previous logic required that, in proof of an **unless** property, an assertion be proved for all possible transactions, i.e., over the set **TRS**. On the other hand, this generalized logic requires the proof of an assertion for all possible synchronic groups of the program, denoted by **SG**.

For the synchronic group logic, we define the logical relation **unless** as follows:

$$p \textbf{ unless } q \equiv$$
$$\langle \forall S : S \in \textbf{SG} :: \{p \wedge \neg q\}\ S\ \{p \vee q\}\rangle.$$

If synchronic groups are restricted to single transactions, this definition is the same as the definition given for the earlier subset Swarm logic.

We define the **ensures** relation as follows:

$$\begin{aligned} & p \textbf{ ensures } q \equiv \\ & \quad (p \textbf{ unless } q) \wedge \\ & \quad \langle \exists t : t \in \textbf{TRS} :: (p \wedge \neg q \Rightarrow [t]) \wedge \\ & \qquad \langle \forall S : S \in \textbf{SG} \wedge t \in S :: \\ & \qquad\quad \{p \wedge \neg q\}\ S\ \{q\}\rangle\rangle. \end{aligned}$$

This definition requires that, when $p \wedge \neg q$ is *true*, there exists a transaction t in the transaction space such that all synchronic groups which can contain t will establish q when executed from a state in which $p \wedge \neg q$ holds. Because of the fairness criterion, transaction t will eventually be chosen for execution, and hence one of the synchronic groups containing t will be executed. Instead of requiring that we find a single "statement" which will eventually be executed and establish the desired state, this rule requires that a group of "statements" (i.e, set of synchronic groups) be found such that each will establish the desired state and that one of them will eventually be executed. If synchronic groups are restricted to single transactions, this definition is the same as the definition for the subset Swarm logic.

6 Conclusions

The Swarm programming logic is the first axiomatic proof system for a shared dataspace "language." To our knowledge, no axiomatic-style proof systems have been published for Linda, rule-based languages, or any other shared dataspace language. Exploiting the similarities of the Swarm and UNITY computational models, we have developed a programming logic for Swarm which is similar in style to that of UNITY. The Swarm logic uses the same logical relations as UNITY, but the definitions of the relations have been generalized to handle the dynamic nature of Swarm, i.e., dynamically created transactions and the synchrony relation. In this paper we have shown how one can extend the proof logic for Swarm to accomodate the dynamic formation of synchronic groups specified by the runtime redefinition of the synchrony relation.

Acknowledgements: This work was supported by the Department of Computer Science, Washington University, Saint Louis, Missouri. The unbounded region labeling program used in this paper is based on a program developed by Rose Fulcomer Gamble and the first author.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, Aug. 1986.

[2] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Mass., 1985.

[3] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Mass., 1988.

[4] H. C. Cunningham. *The Shared Dataspace Approach to Concurrent Computation: The Swarm Programming Model, Notation, and Logic.* PhD thesis, Washington University, St. Louis, Aug. 1989.

[5] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[6] M. Rem. Associons: A program notation with tuples instead of variables. *ACM Trans. Program. Lang. Syst.*, 3(3):251–62, July 1981.

[7] G.-C. Roman. Language and visualization support for large-scale concurrency. In *Proc. 10th Int. Conf. on Software Engineering*, pages 296–308. IEEE, Apr. 1988.

[8] G.-C. Roman and H. C. Cunningham. A shared dataspace model of concurrency—Language and programming implications. In *Proc. 9th ICDCS*, pages 270–9. IEEE, June 1989.

[9] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Mass., 1986.