Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-89-43

1989-10-17

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 the UNITY-model 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. Taking advantage of the similarities of the Swarm and UNITY computational models, we have developed a programming logic for Swarm and UNITY computational models, we have developed a programming logic for Swarm which is... Read complete abstract on page 2.

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

Recommended Citation
Roman, Gruia-Catalin and Cunningham, H. Conrad, "The Synchronic Group: A Concurrent Programming Concept and its Proof Logic" Report Number: WUCS-89-43 (1989). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/753

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

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/753

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 the UNITY-model 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. Taking advantage of the similarities of the Swarm and UNITY computational models, we have developed a programming logic for Swarm and UNITY computational models, we have developed a programming logic for Swarm which is similar in style to that 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 relations. The Swarm programming logic is the first axiomatic proof system for a shared dataspace language, i.e. a language in which communication is accomplished via a shared content-addressable data structure. To our knowledge, no axiomatic-style proof systems have been published for Linda, a production rule language, or any other shared dataspace language.

THE SYNCHRONIC GROUP: A CONCURRENT PROGRAMMING CONCEPT AND ITS PROOF LOGIC

Gruia-Catalin Roman
H. Conrad Cunningham

WUCS-89-43

October 1989

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

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

Gruia-Catalin Roman

Department of Computer Science
Washington University
Campus Box 1045, Bryan 509
One Brookings Drive
Saint Louis, Missouri 63130-4899

(314) 889-6190
roman@wucs1.WUSTL.edu

H. Conrad Cunningham

Department of Computer Science
University of Mississippi
302 Weir Hall
University, Mississippi 38677

(601) 232-5358
hcc@candy.cs.OLEMISS.edu

October 17, 1989

Abstract

Swarm is a computational model which extends the UNITY model 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*. Taking advantage of 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. The Swarm programming logic is the first axiomatic proof system for a *shared dataspace language*, i.e., a language in which communication is accomplished via a shared content-addressable data structure. To our knowledge, no axiomatic-style proof systems have been published for Linda, production rule languages, or any other shared dataspace language.

1 INTRODUCTION

New programming approaches are needed to exploit the capabilities of multiple processors operating in parallel. Although multicomputer implementations of traditional sequential programming languages can effectively exploit some parallelism in programs, in general the conceptual frameworks underlying such languages limit the degree of parallelism that can be achieved on currently available architectures. To overcome the weaknesses of sequential languages, a wide variety of approaches to concurrent programming have been advanced, e.g., Ada[1] [3], CSP [11], FP [4], Concurrent Prolog [17], and Actors [1]. Although many of these approaches have strong features and clusters of advocates, there is no broad agreement on which approaches are the "winners" for concurrent programming. No approach has achieved a good balance among conceptual elegance, support for program verification, programming convenience, and various pragmatic issues. Research is still needed to explore new paradigms and concurrent programming techniques.

Landmark developments are rare—their full significance often not appreciated until a significant amount of time has passed. Although Hoare's paper "Communicating Sequential Processes" [11] did receive considerable immediate attention, few would have expected the language fragment he proposed to dominate a whole decade of research in concurrency. Although the CSP approach is still important, the concurrency research community seems to be turning in other directions. What current work will enjoy the same success? We probably will not know until another decade passes. However, one relatively recent development has the makings of an emerging success story—the kind of power and simplicity that can capture the imagination of both researchers and practitioners: Chandy and Misra's UNITY [7].

Chandy and Misra 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. With UNITY, their work has not been directed toward the development of a new programming language per se. Instead, 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

[1] Ada is a registered trademark of the United States Government (Ada Joint Program Office).

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. This led to an assertional programming logic which frees the program proof from the necessity of reasoning about the 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.

Despite its attractiveness, UNITY does have some shortcomings. The static set of statements inhibits the programmer's ability to cleanly specify dynamically evolving computations—those involving frequent and unpredictable creation of subcomputations. The fixed set of variables also makes the handling of problems with unstructured and unbounded data difficult. Although suited for the specification and verification of programs which use a shared-variable or message-passing approach, UNITY is less well suited for paradigms in which data elements are accessed by content rather than by name, e.g., rule-based systems.

We believe a model of concurrency which preserves the gains made by UNITY, while overcoming its limitations, can become a serious alternative to the two dominant concurrent programming paradigms, message-passing and shared variables. Our research has identified a concurrency model that can accomplish this. We call it the *shared dataspace paradigm*. This paradigm, first so named in [15], refers to 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. Gelernter's Linda [2, 6], Rem's Associons [13, 14], Kimura's Transaction Networks [12], our own Swarm model [16], and production rule languages such as OPS5 [5] all follow the shared dataspace approach.

The Swarm model is our primary vehicle for study of the shared dataspace paradigm. In designing Swarm, we attempted to merge 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 valid transaction instances). We replace UNITY's fixed set of variables with a set of tuples, and the fixed set of conditional 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 (all three subsets of) 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 (and restructuring) concurrent computations. This feature, unique to Swarm, 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, that is, a computer which can simultaneously execute asynchronous and synchronous computations. Perhaps architectures of this type could enable both higher performance and greater flexibility in the design of algorithms.

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 reviews a UNITY-style assertional programming logic for Swarm without the synchrony relation and then generalizes the logic to accomodate synchronic groups. Section 5 illustrates the use of synchronic groups by means of a program for labeling regions in an image unbounded on one side. Section 5 illustrates the use of synchronic groups using a program for labeling regions in an image unbounded to one side. The program proof in Section 6 is followed by a few concluding remarks in Section 7.

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 [9] language. We then construct a Swarm program with similar semantics.

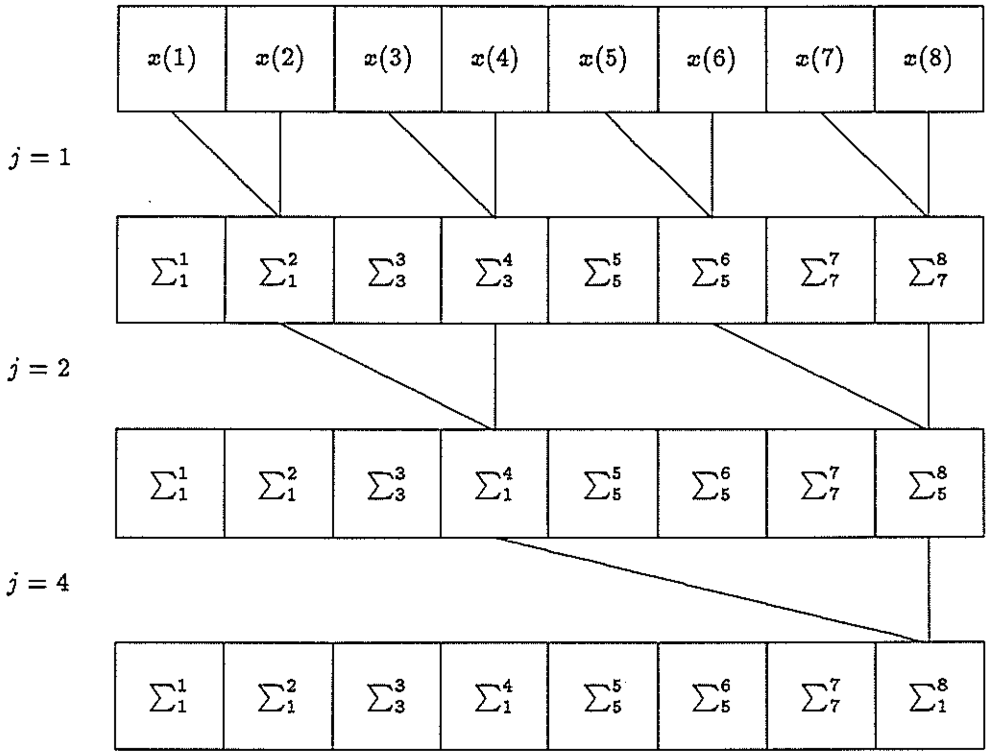
The algorithm given in Figure 1 (adapted from the one given in [10]) sums an array of N integers. For simplicity of presentation, we assume that N is a power of 2. In the program fragment, A is the "input" array of integers to be summed and x is an array of partial sums used by the algorithm. Both arrays 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

$$\langle \| \; k : \mathit{predicate} :: \mathit{assignment} \rangle$$

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* called 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


$x(1)$ $x(2)$ $x(3)$ $x(4)$ $x(5)$ $x(6)$ $x(7)$ $x(8)$
$j = 1$
$\sum_1^1$ $\sum_1^2$ $\sum_3^3$ $\sum_3^4$ $\sum_5^5$ $\sum_5^6$ $\sum_7^7$ $\sum_7^8$
$j = 2$
$\sum_1^1$ $\sum_1^2$ $\sum_3^3$ $\sum_1^4$ $\sum_5^5$ $\sum_5^6$ $\sum_7^7$ $\sum_5^8$
$j = 4$
$\sum_1^1$ $\sum_1^2$ $\sum_3^3$ $\sum_1^4$ $\sum_5^5$ $\sum_5^6$ $\sum_7^7$ $\sum_1^8$

```
$j$ : 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: A Parallel Array-Summation Algorithm using Guarded Commands

set of *query-action* pairs executed in parallel. Each query-action pair is similar to a production rule in a language like OPS5 [5].

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:

$$v1, v2 : x(k-j, v1), x(k, v2) \quad \longrightarrow \quad x(k, v2)\dagger, x(k, v1+v2)$$

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 [18]. 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 † 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:

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

We call each individual query-action pair a *subtransaction* and the overall parallel 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.

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

Figure 2: A Parallel Array-Summation Program in Swarm

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 can be incorporated into a transaction. We define a transaction type called *Sum* as follows:

```
$Sum(j) \equiv$
        $[\| \ k : 1 \le k \le N \wedge k \bmod (j * 2) = 0 ::$
                $v1, v2 : x(k - j, v1), x(k, v2) \longrightarrow x(k, v2)\dagger, x(k, v1 + v2)\ ]$
        $\|$          $j < N \longrightarrow Sum(j * 2)$
```

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.

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. (Picture an undirected graph with transaction instances represented as nodes and a synchrony relationship between transaction instances as an edge between the corresponding nodes.) The reflexive transitive closure of the synchrony relation is thus an equivalence relation. (The equivalence classes are the connected components of the undirected graph mentioned above.) 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

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

Figure 3: A Parallel Array Summation Program Using Synchronic Groups

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

deletes the relationship. Note that the closure relation $\approx$ can be examined, but that only the base synchrony relation $\sim$ can be directly modified. (The dynamic creation of a synchrony relationship between two transactions can be pictured as the insertion of an edge in the undirected graph noted above, and the deletion of a relationship as the removal of an edge.)

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 A PROGRAMMING LOGIC

The Swarm computational model is similar to that of UNITY [7]; 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 paper we follow the notational conventions for UNITY in [7]. We use Hoare-style assertions of the form $\{p\}\ t\ \{q\}$ where p and q are predicates over the dataspace and t is a transaction instance. Properties and inference rules are often 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 [8]. We summarize them below. The Swarm programming logics have been defined so that the theorems proved for UNITY in [7] can also be proved for Swarm.

1. $\{p\}\ t\ \{q\}$.

   The "Hoare triple" means that, whenever the precondition p is *true* and transaction instance t is in the transaction space, all dataspaces which can result from execution of transaction t satisfy postcondition q.

2. $p\ \textbf{unless}\ q \ \equiv\ (\forall t : t \in \text{TRS} : \{p \wedge \neg q\}\ t\ \{p \vee q\})$.

   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. $\textbf{stable}\ p \ \equiv\ p\ \textbf{unless false}$.

   This means that, if p becomes *true*, it remains *true* forever.

4. $\textbf{invariant}\ p \ \equiv\ (\mathit{INIT} \Rightarrow p) \wedge (\textbf{stable}\ p)$.

   Invariants are properties which are *true* at all points in the computation.

5. $p\ \textbf{ensures}\ q \ \equiv\ (p\ \textbf{unless}\ q) \wedge (\exists t : t \in \text{TRS} : (p \wedge \neg q \Rightarrow [t]) \wedge \{p \wedge \neg q\}\ t\ \{q\})$.

   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. (NOTE: The second part of this definition guarantees q will eventually become *true*. This follows from the characteristics of the Swarm execution model. The only way a transaction is removed from the dataspace is as a by-product of its execution; the fairness assumption guarantees that a transaction in the transaction space will eventually be executed.)

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:

   - $$\frac{p \textbf{ ensures } q}{p \longmapsto q}$$
   - $$\frac{p \longmapsto q, \;\; q \longmapsto r}{p \longmapsto r} \qquad \textbf{(transitivity)}$$
   - *For any set W*, $$\frac{(\forall\, m \;:\; m \in W \;:\; p(m) \longmapsto q)}{(\exists\, m \;:\; m \in W \;:\; p(m)) \longmapsto q} \qquad \textbf{(disjunction)}$$

7. **termination** $\equiv$ $(\forall t : t \in \text{TRS} : \neg[t])$.

   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.

We define a basic "Hoare triple" for synchronic groups

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

meaning 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. (For a formal definition of the synchronic group rule, see [8].)

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:

$$p \textbf{ unless } q \;\equiv\; (\forall S : S \in \text{ SG } : \{p \wedge \neg q\}\; S\; \{p \vee q\}).$$

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 following way:

$$\begin{aligned} p \text{ ensures } q \;\equiv\;& (p \text{ unless } q) \;\wedge \\ & (\exists t : t \in \mathrm{TRS} : (p \wedge \neg q \Rightarrow [t]) \;\wedge \\ & \qquad (\forall S : S \in \mathrm{SG} \wedge t \in S : \{p \wedge \neg q\}\; S\; \{q\}\,)\,). \end{aligned}$$

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.

5 A REGION LABELING EXAMPLE

In this section we address the problem of labeling the equal-intensity regions of a digital image unbounded on one side. The image to be processed is arranged on a grid with M rows and an infinite number of columns. 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 bound, the length of each equal-intensity region, i.e., the number of columns intersected by the region, is assumed to be finite and bounded, but of unknown value. For convenience, we let the constant *MaxLen* designate this value for the image to be processed (but 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 values of all integers used in the program must also be bounded above (and below). (Since the current definition of Swarm does not contain true input or output operations, we must simulate these operations. We do not impose the bounded-values restriction on "counters" used to record the current position in the input or output 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 leftmost (smallest x-coordinate) column of the image in the window. 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. (We use the 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.

Figure 5 shows a Swarm program, named *Unbounded*, which uses this sliding window strategy for labeling the regions of an unbounded region. Program parameter M denotes the number of rows in the image. Parameter *Intensity* is an array of input intensity values indexed by the pixel coordinates; parameters *Lo* and *Hi* denote the bounds on these intensity values. The definitions section introduces named constants and "macros." Predicates $Pixel(P)$, $neighbors(P,Q)$, $R\_neighbors(P,Q)$, $on\_left(p)$, and $on\_right(p)$ and the constant *ONE* allow the other sections to be expressed in a more concise and readable fashion. The tuple types section declares the types of tuples that can exist in the tuple space. The program declares three tuple types: *has_label*

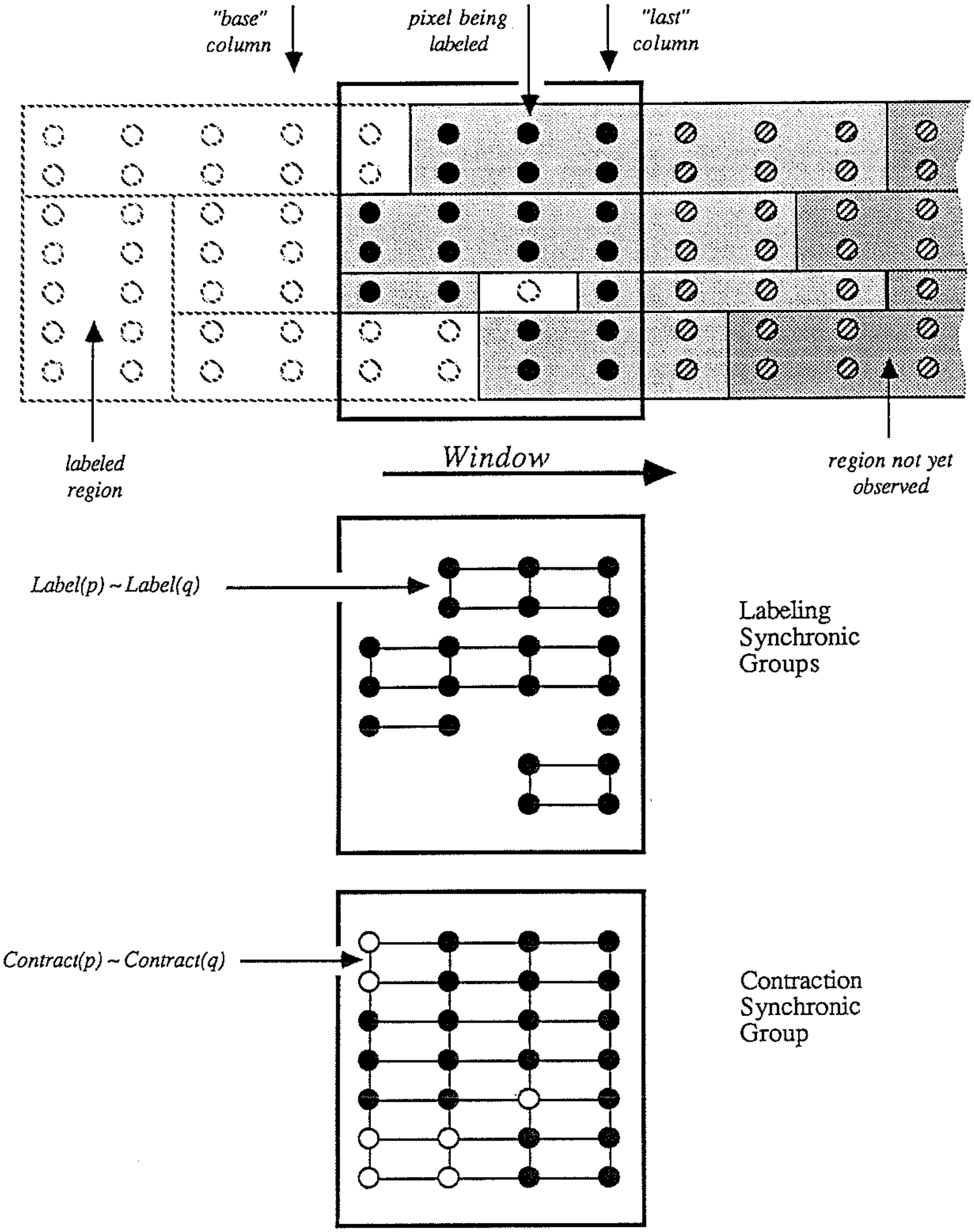


Figure 4: The Window Metaphor and Related Synchronic Groups

```
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
   $[P, Q, L ::$
      $Pixel(P) \equiv [\exists c, r : P = (c, r) :: c \geq 1, 1 \leq r \leq M];$
      $neighbors(P, Q) \equiv$
         $Pixel(P), Pixel(Q), P \neq Q,$
         $[\exists x, y, a, b : P = (x, y), Q = (a, b) :: a - 1 \leq x \leq a + 1, b - 1 \leq y \leq b + 1];$
      $R\_neighbors(P, Q) \equiv$
         $neighbors(P, Q), [\exists \iota :: has\_intensity(P, \iota), has\_intensity(Q, \iota)];$
      $on\_left(P) \equiv Pixel(P), [\exists r :: P = (1, r)];$
      $on\_right(P) \equiv Pixel(P), [\exists c, r : final(c) :: P = (c, r)];$
      $ONE \equiv (1, 0)$
   ]
tuple types
   $[P, L, I, C : Pixel(P), Pixel(L), Lo \leq I \leq Hi, C \geq 1 ::$
      $has\_label(P, L);$
      $has\_intensity(P, I);$
      $final(C)$
   ]
transaction types
   $[P, Next : Pixel(P), Next > 1 ::$
      $Label(P) \equiv \cdots;$
      $Expand(Next) \equiv \cdots;$
      $Contract(P) \equiv \cdots$
   ]
initialization
   $[P : on\_left(P) ::$
      $has\_intensity(P, Intensity(P)), has\_label(P, P), Label(P), Contract(P)],$
   $[P, Q : on\_left(P), neighbors(P, Q), on\_left(Q), Intensity(P) = Intensity(Q) ::$
      $Label(P) \sim Label(Q)],$
   $[P, Q : on\_left(P), neighbors(P, Q), on\_left(Q) :: Contract(P) \sim Contract(Q)],$
   $Expand(2), final(1)$
end
```

Figure 5: An Unbounded Region Labeling Program in Swarm

```
$Label(P) \equiv$
          $\rho, \lambda1, \lambda2 : has\_label(P, \lambda1)\dagger, R\_neighbors(P, \rho), has\_label(\rho, \lambda2), \lambda2 < \lambda1$
                  $\longrightarrow has\_label(P, \lambda2)$
  $\|$          $on\_right(P) \longrightarrow$ skip
  $\|$          OR  $\longrightarrow Label(P)$
  $\|$      $\iota, \lambda :$ NOR, $has\_intensity(P, \iota)\dagger, has\_label(P, \lambda)\dagger \longrightarrow$ skip
  $\|$          NOR $\longrightarrow [\rho : neighbors(P, \rho) :: (Label(P) \sim Label(\rho))\dagger]$
```

Figure 6: Unbounded Region Labeling—*Label* Transaction

pairs a pixel with a label, *has_intensity* pairs a pixel with its intensity value, and *final* records the x-coordinate of the rightmost column of the window.

Program *Unbounded* uses three transaction types—*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. Figures 6, 7, and 8 show the details of these transaction definitions.

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 neighboring 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.

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

Figure 7: Unbounded Region Labeling—*Expand* Transaction

The special *global* predicates AND, OR, NAND, and NOR (having the same meanings as in digital logic design) are used in transaction queries. These special predicates examine the success status of all the simultaneously executed subtransaction queries which do not involve global predicates, i.e., the *local* queries. For example, the predicate OR succeeds if any of the local queries in the transaction also succeed; NOR (not-or) succeeds if none of the local queries succeed. The scope of the global predicates extends to all local subtransactions in the synchronic group.

Now we take a closer look at the details of the transaction definitions. A *Label*(P) transaction (Figure 6) consists of five subtransactions. The first two subtransactions involve local queries. If pixel P has a neighbor pixel (in the same region) which has a smaller label, then the first subtransaction relabels P to the neighbor's label. The second subtransaction succeeds when pixel P is on the right boundary of the window. This test is part of the detection strategy for labeling completion. A region is not yet complete if there can exist more pixels in the region that have not yet been input. The remaining three transactions use the special global predicates OR and NOR. The third subtransaction's query succeeds when *any* of the *local* subtransactions of any transaction *in the synchronic group* succeeds; on success it reinserts the *Label*(P) transaction. Gradually the smallest label will propagate throughout the region during the successive executions of the *Label*

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

Figure 8: Unbounded Region Labeling—*Contract* Transaction

transactions on a region. The fourth and fifth subtransaction queries succeed when *none* of the local subtransactions in the synchronic group succeeds. In this case the region has been completely labeled and all information about the region can be deleted. (In its current form this program does not generate any "output.")

Only one *Expand* transaction (Figure 7) exists at any point in the computation; it is not in a synchronic group with any other transaction. The *Expand*(*Next*) transaction simulates an input operation—bringing the *Next* column of pixels from the "input" array *Intensity* into the dataspace—and builds appropriate synchronic groups of *Label* and *Contract* transactions. The input of a new column is enabled when there exists some region which spans the width of the window; *Expand* detects this situation by testing for a pixel on the right boundary of the window which is labeled by a pixel's coordinates from the left boundary of the window.

A *Contract*(P) transaction contracts the window when the leftmost column has been completely processed; it consists of one local and several global subtransactions. The local subtransaction succeeds when pixel P is on the left boundary of the window and the dataspace contains tuples associated with P. If the local subtransaction fails for all pixels in the window, then the leftmost column of the window is empty. All pixels in the column have been completely processed; thus the entire window can be shifted one column to the right. The global subtransactions accomplish this shifting by decrementing by one column the pixel coordinates recorded in tuples and synchrony relation links. The first and second global subtransactions also keep the window contraction activity alive by reinserting the *Contract*(P) transaction.

6 PROOF OUTLINE

Although at any point in the computation the program only has access to a narrow window imposed upon the image, we find the use of the full unbounded image to be convenient in reasoning about the program. In the statement of properties we use pixel coordinates with respect to the beginning of the full image and identify the regions of the image with integers beginning with 1. We define $R(i)$ to be the set of pixels in region i; $w(i)$ to be the "winning" pixel for region i—the pixel with smallest coordinates. For convenience, we also define *left*(i) and *right*(i) to be the leftmost and rightmost column numbers of region i. We also let *col*(p) denote the column, or x-coordinate, of pixel p, the predicate *col_gone*(c) assert that no tuples associated with pixels in column c exist in the dataspace, and the predicate *reg_gone*(i) assert that region i has its final labeling and the associated tuples have been deleted. Unless otherwise stated we assume that free variables occurring in property assertions are universally quantified implicitly over all valid values of the appropriate type, e.g., p and q over all pixels in the full image, i and j over all regions, c and d over all columns, and b over all intensity values.

We augment the program with auxiliary statements and data structures to capture additional information about history of the computation. The auxiliary variable *base* always points to the column immediately to the left of the current window. The variable is initialized to zero; it is incremented by one each time the *Contract* synchronic group shifts the left side of the window by one unit. The variable *last* always points to the rightmost column of the window. The variable is initialized to one; it is incremented by one whenever the *Expand* transaction brings another column into the dataspace.

To complement the *Intensity* array, we add a *pix_label* array; both of these arrays are indexed by the absolute pixel coordinates. Whenever a *Label* transaction changes a *has_label* tuple for a pixel, the corresponding *pix_label* array element is changed to the corresponding label value. The *pix_label* array is not changed upon deletion of the *has_label* tuple.

6.1 Important Invariants

Formally, we relate the values of the tuples in the dataspace to the auxiliary structures with the Window Intensity, Window Label, and Window Boundary invariants. In addition, the Window Integrity invariant requires that the window be at least one column wide. (For pixel coordinates p in the full image, the notation p' denotes the expression $p - (base, 0)$.)

Property 1 (Window Intensity)

invariant $(\# n :: has\_intensity(p', n)) \leq 1 \wedge$
$(has\_intensity(p', b) \Rightarrow Intensity(p) = b)$

Property 2 (Window Label)

invariant $(\# t :: has\_label(p', t)) \leq 1 \wedge (has\_label(p', l') \Rightarrow pix\_label(p) = l)$

Property 3 (Window Boundary)

invariant $final(c) \equiv (c = last - base)$

Property 4 (Window Integrity)

invariant $0 \leq base < last$

In addition to the Window properties, we constrain a pixel's label to be the coordinates of some pixel within the same region. Moreover, we require the label to be no larger than the pixel's own coordinates. We formalize this constraint as the Labeling Invariant.

Property 5 (Labeling Invariant)

invariant $p \in R(i) \wedge pix\_label(p) = l \;\Rightarrow\; l \in R(i) \;\wedge\; w(i) \leq l \leq p$

To faithfully represent the problem, the labeling of all pixels to the left of the window must be complete and the associated tuples must be deleted. We formalize this requirement, in a slightly stronger way, as the Completion Invariant.

Property 6 (Completion Invariant)

invariant $p \in R(i) \;\wedge\; col(p) \leq base \;\Rightarrow\; reg\_gone(i)$

The four Window invariants and the Labeling and Completion invariants comprise a first correctness criterion—the faithfulness of the program structures to the problem.

The second criterion for correctness of the program is the Labeling Stability property. This safety property asserts that, once a pixel is labeled with the winning pixel for its region, the pixel's label will not change as the computation proceeds.

Property 7 (Labeling Stability)

stable $p \in R(i) \wedge pix\_label(p) = w(i)$

The third criterion for correctness is the Bounded Window property. This property asserts that the window is at most one column wider than the maximum length for individual regions.

Property 8 (Bounded Window)

invariant $last - base \leq MaxLen + 1$

In the programming logic, unless properties must be proved with respect to the set of all possible synchronic groups of a program. To simplify this proof process, we can specify properties which characterize the actual structures of the synchronic groups that can arise during a computation. Once these synchronic group properties have been verified, we can use them in the proofs of other properties.

The unbounded region labeling program has three types of transactions—*Label*, *Expand*, and *Contract*. One or more transactions of a single type are combined to form a synchronic group. Groups consisting of different types of transactions are not allowed by the program. This notion is formalized as the Synchronic Group Integrity invariant.

Property 9 (Synchronic Group Integrity)

$$\text{invariant} \quad \begin{array}{l} \neg(Label(p) \sim Expand(c)) \wedge \\ \neg(Expand(c) \sim Contract(q)) \wedge \\ \neg(Contract(q) \sim Label(p)) \end{array}$$

At any point during the computation there exists a single *Expand* transaction. It is associated with the column of the image immediately to the right of the window—the next column to be inserted. This transaction comprises a single element synchronic group. We formalize this property as the Expand Group invariant.

Property 10 (Expand Group)

invariant $(\# c :: Expand(c)) = 1 \wedge Expand(last + 1)$

The Contract Group invariant specifies the structure of the synchronic groups involving *Contract* transactions. At any point during the computation there exists a single synchronic group of this type. The group includes one transaction for each pixel visible in the window.

Property 11 (Contract Group)

$$\text{invariant} \quad \begin{array}{l} (\forall p : base < col(p) \leq last : Contract(p')) \wedge \\ (\forall p, q : Contract(p') \wedge Contract(q') : Contract(p') \approx Contract(q')) \end{array}$$

The structures of the *Label* groups are more complex. The portion of an unfinished region visible in the window may be divided into one or more subregions by the right boundary of the window. At any point during the computation, for each unfinished subregion there exists a synchronic group which exactly covers the subregion. (There is a $Label(p)$ transaction for each pixel p of the subregion; no additional transactions are part of the group.) The Label Group invariant formally characterizes the structures of this type of synchronic group. (The $neighbors(p,q)$ predicate used in this property is defined in the **definitions** section of the program *Unbounded.*)

Property 12 (Label Group)

invariant

$$\begin{aligned}&(\forall p, q : p \in R(i) \wedge q \in R(i) :\\ &\qquad \neg reg\_gone(i) \wedge neighbors(p,q) \wedge col(p) \leq last \wedge col(q) \leq last\\ &\qquad \equiv\\ &\qquad Label(p') \wedge Label(q') \wedge (Label(p') \sim Label(q'))\,)\end{aligned}$$

There are two proof obligations in proof of an invariant: showing that the initial state satisfies the property and showing that all synchronic groups preserve the property. Once the synchronic group invariants above have been proven, they can be used as theorems in the proofs of other properties. We do not prove any of the invariants here; most of the proofs are not difficult and are given in [8].

6.2 Progress Proof

The fourth criterion for correctness of the region labeling program is the Labeling Completion property. This progress property asserts that, when the computation is begun in a valid initial state, it will eventually reach a state in which labeling of any finite prefix of the image will be finished.

Property 13 (Labeling Completion)

$$INIT \wedge C > 0 \longrightarrow base \geq C$$

The Labeling Completion property asserts that any execution of the unbounded region labeling program will actually label the regions. Specifically, the property guarantees that any finite prefix of the columns of the full image will eventually be labeled and the associated data tuples deleted. *In terms of the sliding window metaphor, the window will eventually slide to the right of any* *arbitrary prefix of the full image.* Because of the Completion Invariant, we can conclude that the portion of the image to the left of the window has been labeled as desired.

We use the following approach for this progress proof. To show that the window eventually slides past a finite prefix, we show that the left boundary of the window will always eventually advance one column. For the left boundary to advance past a column, all pixels in that column must be finished, i.e., labeled with the winning label and the associated data tuples removed. Because of the Completion Invariant, we only need to consider left-anchored regions, regions which begin in the leftmost column of the window and extend to the right. These regions will eventually be completed and all pixels removed.

To show that labeling of a left-anchored region will eventually finish, we must prove:

- if the region extends beyond the right boundary of the window, eventually all missing columns will be inserted into the window,
- if the region is completely contained within the window, it will eventually be labeled and deleted.

If the region extends beyond the right boundary, then eventually a *Label* synchronic group will propagate a label from the left boundary across to the right boundary. This enables the input of the next column. Eventually a left-anchored region will be completely within the window. The same label propagation mechanism will then eventually complete the labeling and remove the region from the dataspace.

Below we sketch a proof of the Labeling Completion property. For pedagogical reasons, we proceed in a top-down fashion. We first outline how a higher level leads-to property can be proved using other leads-to, **ensures**, and **unless** properties, then we address each unproven property in a similar fashion. To keep track of the outstanding proof obligations, we will list periodically the properties requiring proof in a box as shown below.

Properties to prove: Labeling Completion.

¶ Proof of Labeling Completion. *To show that the window eventually slides past a finite prefix, we show that the left boundary of the window will always eventually advance one column.*

The Labeling Completion property is proved by an induction needing a simpler "one-step" property:

Property 14 $base = c \longmapsto base = c + 1$

∎

Properties to prove: 14.

¶ **Proof of Property 14.** *For the left boundary of the window to advance past a column, all pixels in that column must be labeled with the winning pixel and deleted from the dataspace. All pixels in the column will eventually be labeled and deleted.*

Consider two cases for the leftmost column of the window: $col\_gone(c+1)$ and $\neg col\_gone(c+1)$, where $col\_gone(c+1)$ is *true* if and only if there do not exist any tuples in the dataspace associated with the pixels in column $c+1$. Note that $base = c$ unless $base = c+1$.

(1) Case $col\_gone(c+1)$. This case is covered by the following property:

Property 15 $\quad base = c \wedge col\_gone(c+1)$ **ensures** $base = c+1$

(2) Case $\neg col\_gone(c+1)$. First, using the leads-to property 16 below and the unless property noted above, apply the Progress-Safety-Progress (PSP) Theorem [7], then apply the Cancellation Theorem for Leads-to [7] using case (1).

Property 16 $\quad base = c \wedge \neg col\_gone(c+1) \longmapsto col\_gone(c+1)$

∎

Properties to prove: 15, 16.

As an **ensures**, property 15 has two proof obligations. Let *LHS* and *RHS* refer to the left- and right-hand sides of the **ensures** assertion. (1) We must show *LHS* **unless** *RHS*. (2) We must also show that, whenever $LHS \wedge \neg RHS$ is *true*, there exists a transaction in the dataspace such that execution of any synchronic group containing that transaction will always establish *RHS* as *true*.

¶ **Proof of Property 15.** *The left boundary of the window will be advanced when the leftmost column has been processed and all pixels removed.*

Prove the assertion $base = c \wedge col\_gone(c+1)$ **ensures** $base = c+1$.

(1) Unless part. Clearly $base = c$ **unless** $base = c+1$ and **stable** $c+1 \leq last \wedge col\_gone(c+1)$. Since $base < last$ by the Window Integrity invariant, we can conclude

$$base = c \wedge col\_gone(c+1) \textbf{ unless } base = c+1$$

using the Simple Conjunction Theorem for Unless [7].

(2) Exists part. Because of the Window Integrity and Contract Group invariants, we know there exists a pixel P, $P = (c+1, 1)$, such that

$$base = c \wedge col\_gone(c+1) \;\Rightarrow\; Contract(P').$$

Because of the Synchronic Group Integrity and Contract Group invariants, it is easy to see that all synchronic groups containing *Contract(P')* establish $base = c+1$ when the precondition $base = c \wedge col\_gone(c+1)$ is *true*. ▮

Properties to prove: 16.

¶ **Proof of Property 16.** *The unfinished pixels in the leftmost column of the window are in regions which begin in that column and extend to the right. Labeling of these regions will eventually be completed and all pixels removed.*

Prove the assertion $base = c \wedge \neg col\_gone(c+1) \longmapsto col\_gone(c+1)$. Consider two cases for a region i which intersects column $c+1$: $left(i) \leq c$ and $left(i) = c+1$.

(1) Case $left(i) \leq c$. Because of the Completion Invariant and definition of *reg_gone*,

$$base = c \wedge left(i) \leq c \;\Rightarrow\; (\forall p : p \in R(i) \wedge col(p) = c+1 : pix\_gone(p))$$

where $pix\_gone(p)$ is *true* when there are no *has_label* or *has_intensity* tuples in the dataspace which are associated with pixel p. The Implication Theorem for Leads-to [7] allows the "$\Rightarrow$" to be replaced by a "$\longmapsto$".

(2) Case $left(i) = c+1$. The following property is the essence of this case.

Property 17 $\quad base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \;\longmapsto\; reg\_gone(i)$

Because of property 17, the definition of *reg_gone*, and the Implication Theorem, we can deduce

$$\begin{aligned} &base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \;\longmapsto \\ &\quad (\forall p : p \in R(i) \wedge col(p) = c+1 : pix\_gone(p)\,). \end{aligned}$$

Since $col(p) \leq last \wedge pix\_gone(p)$ is stable and $base < last$ is invariant, we apply the Completion Theorem [7] over the regions intersecting column $c+1$ to deduce

$$base = c \wedge \neg col\_gone(c+1) \;\longmapsto\; col\_gone(c+1)).$$

▮

Properties to prove: 17.

For convenience, we define $excess(i)$ to be the total amount the labels on region i exceed the desired labeling (all pixels in the region labeled with the "winning" pixel). More formally,

$$excess(i) \;=\; (\Sigma\, p : p \in R(i) : pix\_label(p) - w(i)\,)$$

where the "Σ" and"$-$" operators denote component-wise summation and subtraction of the coordinates. We use *excess* as to measure the amount of labeling work remaining to be done on a region. In assertions involving *excess* we often use $\mathbf{0}$ (boldface zero) to denote the pair $(0,0)$.

¶ **Proof of Property 17.** *To show that a left-anchored region eventually is finished, we must prove: (a) if the region extends beyond the right boundary of the window, then eventually all missing columns will be inserted into the window; (b) if the region is completely contained within the window, then it will eventually be labeled completely with the winning pixel and all pixels deleted.*

Prove the assertion $base = c \land \neg col\_gone(c+1) \land left(i) = c+1 \;\longmapsto\; reg\_gone(i)$. Consider three cases for the state of left-anchored regions:

- $right(i) < last \land excess(i) = \mathbf{0}$,
- $right(i) < last \land excess(i) > \mathbf{0}$,
- $right(i) \geq last$.

Note that $base = c \land \neg col\_gone(c+1) \land left(i) = c+1$ **unless** $reg\_gone(i)$ and **stable** $right(i) < last$.

(1) Case $right(i) < last \land excess(i) = \mathbf{0}$. This case is covered by the following **ensures** property:

Property 18

$$\begin{aligned} &base = c \land \neg col\_gone(c+1) \land left(i) = c+1 \land right(i) < last \land excess(i) = \mathbf{0} \\ &\qquad\qquad \textbf{ensures } reg\_gone(i) \end{aligned}$$

(2) Case $right(i) < last \land excess(i) > \mathbf{0}$. First apply the PSP Theorem [7] using the leads-to property below and the conjunction of **unless** and **stable** properties noted above, then apply the Cancellation Theorem for Leads-to [7] using case (1).

Property 19 *If the region is completely contained within the window, the winning pixel's label is gradually propagated throughout the region. (Proof omitted.)*

$$\begin{aligned} &base = c \land \neg col\_gone(c+1) \land left(i) = c+1 \land right(i) < last \land excess(i) > \mathbf{0} \\ &\qquad\qquad \longmapsto\; excess(i) = \mathbf{0} \end{aligned}$$

(3) Case $right(i) \geq last$. First apply the PSP Theorem [7] using the leads-to property 20 below and the unless property noted above, then apply the Cancellation Theorem for Leads-to [7] using the disjunction of cases (1) and (2).

Property 20

$$\begin{aligned}&base = c \land \neg col\_gone(c+1) \land left(i) = c+1 \land right(i) \geq last\\ &\qquad\qquad \longmapsto\ right(i) < last\end{aligned}$$

∎

Properties to prove: 18, 20.

¶ **Proof of Property 18.** *If the region is labeled with the winning pixel, it will eventually be deleted.*

Prove the assertion

$$\begin{aligned}&base = c \land \neg col\_gone(c+1) \land left(i) = c+1 \land right(i) < last \land excess(i) = 0\\ &\qquad\qquad \text{ensures } reg\_gone(i).\end{aligned}$$

(1) Unless part. Consider the synchronic groups allowed by the synchronic group invariants. Clearly *Expand* and *Contract* groups and *Label* groups for regions other than i preserve the *LHS* of the ensures. The *Label* group on region i will establish $reg\_gone(i)$ as *true* for the given precondition.

(2) Exists part. Because of the Window Integrity and Label Group invariants, we know

$$base = c \land \neg col\_gone(c+1) \land left(i) = c+1 \land right(i) < last \ \Rightarrow\ Label(w(i)').$$

Because of the Synchronic Group Integrity and Label Group invariants, it is easy to see that all synchronic groups containing $Label(w(i)')$ establish $reg\_gone(i)$ when the precondition $excess(i) = 0$ is *true*. ∎

Properties to prove: 20.

¶ **Proof of Property 20.** *If the region extends beyond the right boundary of the window, missing columns will gradually be inserted into the window.*

Prove the assertion

$$base = c \land \neg col\_gone(c+1) \land left(i) = c+1 \land right(i) \geq last \ \longmapsto\ right(i) < last.$$

Note that

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) \geq last \textbf{ unless } right(i) < last.$$

First, apply the PSP Theorem [7] using the leads-to property 21 below and the **unless** property noted above, then apply the Induction Principle for Leads-to [7].

Property 21

$$\begin{aligned} &base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) \geq last \wedge last = d \\ &\qquad\qquad \longmapsto \; last = d+1 \end{aligned}$$

∎

Properties to prove: 21.

Property 21 means that, if the region extends beyond the right boundary, then eventually a *Label* synchronic group will propagate a label from the left boundary across to the right boundary. This enables the input of the next column. We omit the proof of this property here. The proof can be found in [8].

7 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, production rule languages, or any other shared dataspace language. Taking advantage of the similarities between 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.

Swarm's synchrony relation is an elegant new language construct for dynamically organizing concurrency. Sometimes programmers want to organize the concurrency in a program to match the structure of the program's "input" data. These data may be sparse, loosely structured, or unbounded in some manner. Sometimes programmers may also want to alter the structure of the concurrency as a result of a previous subcomputation. The dynamically modifiable synchrony relation, in conjunction with dynamically created transaction instances, provides a simple mechanism for achieving such program structures.

Acknowledgements: This work was supported by the Department of Computer Science, Washington University, Saint Louis, Missouri. We thank Jerome R. Cox, department chairman, for his support and encouragement. We also thank Ken Cox for his suggestions concerning this article. 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] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Massachusetts, 1986.

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

[3] ANSI, Inc. *Reference Manual for the Ada Programming Language.* American National Standards Institute, Inc., Washington, D.C., January 1983. ANSI/MIL-STD-1815A-1983.

[4] J. Backus. Can programming languages be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

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

[6] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

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

[8] H. C. Cunningham. *The Shared Dataspace Approach to Concurrent Computation: The Swarm Programming Model, Notation, and Logic.* PhD thesis, Washington University, Department of Computer Science, St. Louis, Missouri, August 1989. Advisor: G.-C. Roman.

[9] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[10] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[11] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[12] T. D. Kimura. Visual programming by transaction network. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 648–654. IEEE, January 1988.

[13] M. Rem. Associons: A program notation with tuples instead of variables. *ACM Transactions on Programming Languages and Systems*, 3(3):251–262, July 1981.

[14] M. Rem. The closure statement: A programming language construct allowing ultraconcurrent execution. *Journal of the ACM*, 28(2):393–410, April 1981.

[15] G.-C. Roman. Language and visualization support for large-scale concurrency. In *Proceedings of the 10th International Conference on Software Engineering*, pages 296–308. IEEE, April 1988.

[16] G.-C. Roman and H. C. Cunningham. A shared dataspace model of concurrency—Language and programming implications. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 270–279. IEEE, June 1989.

[17] E. Shapiro. Concurrent Prolog: A progress report. *Computer*, 19(8):44–58, August 1986.

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