Report Number: WUCS-90-02

1990-01-03

Unity-Style Proofs for Shared Dataspace Programs Using Dynamic Statements

Authors: H. Conrad Cunningham and Gruia-Catalin Roman

The term shared dataspace refers to the general class of programming languages in which the principal means of communication among the concurrent components of programs is a common, content-addressable data structure called a dataspace. In the programming language and artificial intelligence communities, there is considerable interest in such languages, e.g. logic-based languages, production rule systems, and the Linda language. However, these languages have not been the subject of extensive program verification research. This paper specifies a proof system for a shared dataspace programming notation called Swarm-- a program logic similar in style to that of UNITY. The paper then uses the logic to reason about a solution to the problem of labeling the connected equal-intensity regions of a digital image.

UNITY-STYLE PROOFS FOR SHARED DATASPACE PROGRAMS USING DYNAMIC STATEMENTS

H. Conrad Cunningham and Gruia-Catalin Roman

WUCS-90-02

January 1990

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

This report also appears as Technical Report UMCIS-1990-02, University of Mississippi, Department of Computer and Information Science, University, Mississippi.

Cunningham is with the Department of Computer and Information Science at The University of Mississippi.

UNITY-style Proofs for Shared Dataspace Programs Using Dynamic Statements

H. Conrad Cunningham

Department of Computer and
Information Science
UNIVERSITY OF MISSISSIPPI
302 Weir Hall
University, Mississippi 38677

(601) 232-5358
cunningham@cs.OLEMISS.edu

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@cs.WUSTL.edu

January 3, 1990

Abstract

The term *shared dataspace* refers to the general class of programming languages in which the principal means of communication among the concurrent components of programs is a common, content-addressable data structure called a dataspace. In the programming language and artificial intelligence communities, there is considerable interest in such languages, e.g., logic-based languages, production rule systems, and the Linda language. However, these languages have not been the subject of extensive program verification research. This paper specifies a proof system for a shared dataspace programming notation called Swarm—a programming logic similar in style to that of UNITY. The paper then uses the logic to reason about a solution to the problem of labeling the connected equal-intensity regions of a digital image.

Keywords

Concurrent/parallel programming, concurrent/parallel languages,
program verification, UNITY, shared dataspace

1 Introduction

Much of the research on verification of concurrent programs has focused on languages with semantics similar to Dijkstra's Guarded Commands [9] or Hoare's Communicating Sequential Processes [12]. Programs in such languages normally involve a moderate level of interaction among largely sequential segments of code, thus limiting the level of parallelism possible. An executing program accesses data entities (variables) by their names; the computation proceeds by transforming the state of these entities.

A few recent programming notations, such as UNITY [5], Action Systems [2], and event predicates [13], have banished sequentiality from the languages, but have retained the state-transition and named-variable concepts. While preserving many of the results of program verification research, these languages make concurrency the normal case, sequentiality the special case.

Interest has also grown in languages which access entities by content rather than by name, e.g., logic-based languages, production rule systems, and the Linda [1, 4] language. By reducing the naming bottleneck, the content-addressable approach seems to encourage higher degrees of concurrency and more flexible connections among components. As yet, program verification techniques for such languages have not been extensively researched.

Because we desire high degrees of concurrency while preserving the programming and verification advantages of the traditional imperative framework, we have focused our research on a new approach to concurrent computation. We are studying concurrent programming languages which employ the shared dataspace model [18], i.e., languages in which the primary means for communication among the concurrent components is a common, content-addressable data structure called a *shared dataspace*. Such languages can bring together a variety of programming styles (synchronous and asynchronous, static and dynamic) within a unified computational framework.

The main vehicle for this investigation is a programming model and notation called *Swarm* [21]. The design of Swarm was influenced by the previous work on Linda [1, 4], Associons [16, 17], OPS5 [3], and UNITY [5]. Following the simple approach taken by the UNITY model, we sought to base Swarm on a small number of constructs we believe are at the core of a large class of shared dataspace languages. The state of a Swarm program consists of a dataspace, i.e., a set of transaction statements and data tuples. Transactions specify a group of dataspace transformations that are performed concurrently.

Swarm is proving to be an excellent vehicle for investigation of the shared dataspace approach. We have defined the Swarm programming notation and specified a formal operational model based on a state-transition approach [7, 21]. Our research is exploring the implications of the shared dataspace approach and the Swarm model for algorithm development and programming methodology [23]. To facilitate formal verification of Swarm programs, we have developed an assertional programming logic and are devising proof techniques appropriate for the dynamic structure of Swarm [7]. In a related effort, we are investigating the use of the shared dataspace model as a basis for a new approach to the visualization of the dynamics of program execution [19, 20].

In this paper, we specify a proof system for Swarm similar in style to that of UNITY and illustrate its use by means of a proof for the problem of labeling equal-intensity regions of a digital image. UNITY uses an assertional programming logic built upon its simple computational model. By the use of logical assertions about program states, the programming logic frees the program proof from the necessity of reasoning about the execution sequences. Instead of annotating the program text with predicates as most assertional systems do, the UNITY logic seeks to extricate the proof from the text by relying upon proof of program-wide properties, e.g., global invariants and progress properties.

With the Swarm logic, we extend these ideas into the shared dataspace framework. Swarm's underlying computational model is similar to that of UNITY, but has a few key differences. A UNITY program consists of a static set of deterministic multiple-assignment statements acting upon a shared-variable state space. The statements in the set are executed repeatedly in a nondeterministic, but fair, order. Swarm is based on less familiar programming language primitives—nondeterministic transaction statements which act upon a dataspace of anonymous tuples—and extends the UNITY-like model to a dynamically varying set of statements. To incorporate these features, we define a proof rule for transaction statements to replace UNITY's rule for multiple-assignment statements, redefine UNITY's **ensures** relation to accommodate the creation and deletion of transaction statements, and replace UNITY's use of fixed-point predicates with other methods for determining program termination. Otherwise, the programming logics are identical.

This paper consists of three parts. Section 2 overviews relevant aspects of the Swarm language and model. Section 3 introduces the Swarm programming logic. In section 4, we use the Swarm logic to verify a solution to the region-labeling problem.

2 The Swarm Notation

Our choice of the name *Swarm* evokes 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 (GC) [9] language. We then construct a Swarm program with similar semantics.

The algorithm given in Figure 1 (adapted from the one given in [11]) 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 **do** 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

$$(\!|\ k : \mathit{predicate} :: \mathit{assignment}\ )$$

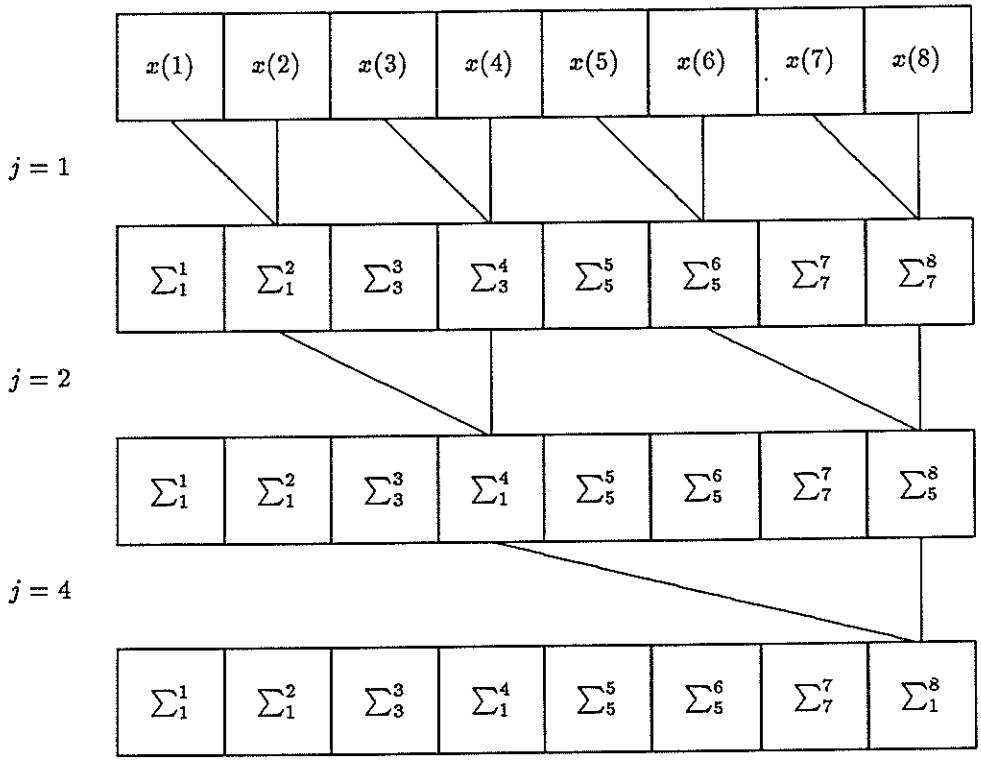
is a parallel assignment command. The *assignment* is executed in parallel for each value of k which satisfies the *predicate*; the entire construct is performed as one atomic action.

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 execute 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 [3].

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. Assuming that j and k are constants, we can express an instance of the array assignment in the **do** loop as a Swarm transaction in the following way:

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



j : **integer**;
$x(i : 1 \leq i \leq N)$: **array of integer**;

$j := 1$; $\langle k : 1 \leq k \leq N :: x(k) := A(k) \rangle$;
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

In the above notation, 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.

The execution of a Swarm query is similar to the evaluation of a clause in Prolog [24]. The query in the paragraph above 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:

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

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

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 that data tuples are inserted, 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 do command in Figure 1 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:

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

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

For a correct computation, the Swarm array summation program must be initialized with the following set of tuples in the dataspace:

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

In addition, the transaction $Sum(1)$ must also be present in the dataspace.

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 below:

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

Tuples marked by a † symbol in the query part are deleted if the entire query succeeds. The **program**, **tuple types**, and **transaction types** portions of the program declare program structures. The **initialization** section defines the contents of the initial dataspace.

3 A Programming Logic

This section presents an assertional programming logic for Swarm, building upon the formal model given in [7] and [21]. For simplicity in presentation, we do not consider the synchrony relation feature of Swarm, but we do keep the notation used here compatible with that needed

for the full language. The model and logic presented here have been generalized to incorporate synchronic groups [7, 22].

A Swarm dataspace can be partitioned into a finite tuple space and a finite transaction space. For a dataspace d, $\mathbf{Tp}.d$ denotes the tuple space of d, $\mathbf{Tr}.d$ the transaction space. The **tuple types** and **transaction types** sections of a program define the set of all possible tuple instances **TPS** and all possible transaction instances **TRS**.

Although actual parallel implementations of Swarm can overlap the execution of transactions, we have found the following program execution model to be convenient. The program begins execution with the 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. (This concept of fairness is called weak fairness in [10] and justice in [14].) Program execution continues until there are no transactions in the dataspace.

We model a Swarm program as a set of execution sequences, each of which is infinite and denotes one possible execution of the program. Let e denote one of these sequences. Each element e_i, $i \geq 0$, of e is an ordered pair consisting of a program dataspace $\mathbf{Ds}.\mathrm{e}_i$ and a set $\mathbf{Sg}.\mathrm{e}_i$ containing a single transaction chosen from $\mathbf{Tr.Ds}.\mathrm{e}_i$. (If $\mathbf{Tr.Ds}.\mathrm{e}_i = \emptyset$, then $\mathbf{Sg}.\mathrm{e}_i = \emptyset$.)

The transition relation predicate **step** expresses the semantics of the transactions in **TRS**; the values of this predicate are derived from the query and action parts of the transaction body. The predicate $\mathbf{step}(d, S, d')$ is *true* if and only if the transaction in set S is in dataspace d and the transaction's execution can transform dataspace d to a dataspace d'. (The predicate $\mathbf{step}(d, \emptyset, d')$ is *true* if and only if $\mathbf{Tr}.d = \emptyset$ and $d = d'$.)

We define **Exec** to be the set of all execution sequences e, as characterized above, which satisfy the following criteria:

- $\mathbf{Ds}.\mathrm{e}_0$ is a valid initial dataspace of the program.
- For $i \geq 0$, $\mathbf{step}(\mathbf{Ds}.\mathrm{e}_i, \mathbf{Sg}.\mathrm{e}_i, \mathbf{Ds}.\mathrm{e}_{i+1})$ is *true*.
- e is *fair*, i.e., $(\forall i, t : 0 \leq i \wedge t \in \mathbf{Tr.Ds}.\mathrm{e}_i ::$
$$(\exists j : j \geq i :: \mathbf{Sg}.\mathrm{e}_j = \{t\} \wedge (\forall k : i \leq k \leq j :: t \in \mathbf{Tr.Ds}.\mathrm{e}_k))).$$

Terminating computations are extended to infinite sequences by replication of the final dataspace.

Although we could use this formalism directly to reason about Swarm programs, we prefer to reason with assertions about program states rather than with execution sequences. The Swarm computational model is similar to that of UNITY [5]; hence, a UNITY-like 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 [5]. We use Hoare-style assertions of the form $\{p\}\ t\ \{q\}$ where p and q are predicates 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 $p(d)$ to denote the evaluation of predicate p with respect to dataspace d and the notation $(p.\wedge \neg q)(\mathrm{e}_i)$ to denote the evaluation of the predicate $p \wedge \neg q$ with respect to $\mathbf{Ds}.\mathrm{e}_i$. Below we also use the notation $[t]$ to denote the predicate "transaction instance t is in the transaction space."

UNITY assignment statements are deterministic; execution of a statement from a given state will always result in the same next state. This determinism, plus the use of named variables, enables UNITY's assignment proof rule to be stated in terms of the syntactic substitution of the source expression for the target variable name in the postcondition predicate. In contrast, Swarm transaction statements are nondeterministic; execution of a statement from a given dataspace may result in any one of potentially many next states. This arises from the nature of the transaction's queries. A query may have many possible solutions with respect to a given dataspace. The execution mechanism chooses any one of these solutions nondeterministically—fairness in this choice is *not* assumed. Since the state of a Swarm computation is represented by a set of tuples rather than a mapping of values to variables, finding a useful syntactic rule is difficult.

Accordingly, we define the meaning of the assertion $\{p\}\ t\ \{q\}$ for a given Swarm program in terms of the transition relation predicate **step** as follows:

$$\{p\}\ t\ \{q\} \quad \equiv \quad (\forall d, d' : \mathbf{step}(d, \{t\}, d') :: p(d) \Rightarrow q(d')).$$

Informally this 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. In terms of the execution sequences this rule means

$$(\forall e, i : e \in \mathbf{Exec} \wedge 0 \leq i :: p(e_i) \wedge \mathbf{Sg}.e_i = \{t\} \ \Rightarrow \ q(e_{i+1})).$$

As in UNITY's logic, the basic safety properties of a program are defined in terms of **unless** relations. The Swarm definition mirrors the UNITY definition:

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

Informally, 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*. (Remember **TRS** is the set of all possible transactions, not a specific transaction space.) In terms of the sequences this rule implies

$$(\forall e, i : e \in \textbf{Exec} \wedge 0 \leq i :: (p \wedge \neg q)(e_i) \Rightarrow (p \vee q)(e_{i+1}))\ .$$

From this we can deduce

$$\begin{array}{l}(\forall e, i : e \in \textbf{Exec} \wedge 0 \leq i :: \\ \qquad p(e_i) \Rightarrow \quad (\forall j : j \geq i :: (p \vee \neg q)(e_j)) \vee \\ \qquad\qquad\quad (\exists k : i \leq k :: q(e_k) \wedge (\forall j : i \leq j \leq k :: (p \wedge \neg q)(e_j))))\ .\end{array}$$

In other words, either (1) $p \wedge \neg q$ continues to hold indefinitely or (2) q holds eventually and p continues to hold at least until q holds.

Stable and **invariant** properties are fundamental notions of our proof theory. Both can be defined easily as follows:

$$\begin{array}{lcl}\textbf{stable } p & \equiv & p \textbf{ unless false} \\ \textbf{invariant } p & \equiv & (\mathit{INIT} \Rightarrow p) \wedge (\textbf{stable } p)\end{array}$$

Above *INIT* is a predicate which characterizes the valid initial states of the program. A stable predicate remains *true* once it becomes *true*—although it may never become *true*. Invariants are stable predicates which are *true* initially. Note that the definition of **stable** p is equivalent to

$$(\forall t : t \in \textbf{TRS} :: \{p\}\ t\ \{p\}\ ).$$

We also define **constant** properties such that

$$\textbf{constant } p \quad \equiv \quad (\textbf{stable } p) \wedge (\textbf{stable } \neg p).$$

We use the **ensures** relation to state the most basic progress (liveness) properties of programs. UNITY programs consist of a static set of statements. In contrast, Swarm programs consist of a dynamically varying set of transactions. The dynamism of the Swarm transaction space requires

a reformulation of the **ensures** relation. For a given program in the Swarm subset considered in this paper, the ensures relation is defined as follows:

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

Informally, 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 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.

In terms of the execution sequences the **ensures** rule implies

$$(\forall e, i : e \in \textbf{Exec} \wedge 0 \leq i :: p(e_i) \Rightarrow (\exists\, j : i \leq j :: q(e_j) \wedge (\forall\, k : i \leq k < j :: p(e_k)))).$$

The Swarm definition of **ensures** is a generalization of UNITY's definition. If $(\forall t : t \in \textbf{TRS} :: [t])$ is assumed to be invariant, then the Swarm **ensures** definition can be restated in a form similar to UNITY's **ensures**.

The **leads-to** property, denoted by the symbol $\longmapsto$, is commonly used in Swarm program proofs. 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:

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

In terms of the execution sequences, from $p \longmapsto q$, we can deduce

$$(\forall e, i : e \in \textbf{Exec} \wedge 0 \leq i :: p(e_i) \Rightarrow (\exists\, j : i \leq j :: q(e_j))).$$

Informally, $p \longmapsto q$ means once p becomes *true*, q will eventually become *true*. However, p is not guaranteed to remain *true* until q becomes *true*.

UNITY makes extensive use of the *fixed-point* predicate FP which can be derived syntactically from the program text. Since FP predicates cannot be defined syntactically in Swarm, verifications of Swarm programs must formulate program postconditions differently—often in terms of

other **stable** properties. However, unlike UNITY programs, Swarm programs can terminate; a termination predicate **termination** can be defined as follows:

$$\textbf{termination} \equiv (\forall t : t \in \mathbf{TRS} :: \neg[t]).$$

Other than the cases pointed out above (i.e., transaction rule, **ensures**, and *FP*), the Swarm logic is identical to UNITY's logic. The theorems (not involving *FP*) developed in Section 3 of [5] can be proved for Swarm as well. We use the Swarm analogues of various UNITY theorems in the proof in the next section.

4 A Region-Labeling Example

This section applies the programming logic given in Section 3 to the verification of a Swarm program. The program shown is a solution to the region-labeling problem, one of the alternative Swarm solutions shown in [21]. In this section, we formally define the problem and correctness criteria, elaborate the program data structures, and then state the program and argue that it satisfies the correctness criteria.

4.1 The Correctness Criteria

Region labeling is a two-dimensional version of the classical leader election problem [6, 15]. A region-labeling program receives as input a digitized image. Each point in the image is called a *pixel*. The pixels are arranged in a rectangular grid of size N pixels in the x-direction and M pixels in the y-direction. An xy-coordinate on the grid uniquely identifies each pixel. Also provided as input to the program is the intensity (brightness) attribute associated with each pixel. The size, shape, and intensity attributes of the image remain constant throughout the computation.

The concepts of *neighbor* and *region* are important in this discussion. Two different pixels in the image are said to be *neighbors* if their x-coordinates and their y-coordinates each differ by no more than one unit. A *connected equal-intensity region* is a set of pixels from the image satisfying the following property: for any two pixels in the set, there exists a path with those pixels as endpoints such that all pixels on the path have the same intensity and any two consecutive pixels are neighbors. For convenience, we use the term region to mean a connected equal-intensity region.

The goal of the computation is to assign a label to each pixel in the image such that two pixels have the same label if and only if they are in the same region. Furthermore, we require the programs herein to label all the pixels in a region with the smallest coordinates of a pixel in that region. (Comparisons of pixel coordinates are in terms of the lexicographic ordering where, for example, $(x, y) < (a, b) \equiv x < a \vee (x = a \wedge y < b)$.)

Since the number of pixels in the image is finite, there are a finite number of regions. Without loss of generality, we identify the regions with the integers 1 through $Nregions$. We define function R such that:

$$R(i) = \{p : \text{pixel } p \text{ is in region } i :: p\}$$

From the graph theoretic properties of the image, we see that the $R(i)$ sets are disjoint. We also define the "winning" pixel on each region, i.e., the pixel with the smallest coordinates, as follows:

$$w(i) = (\min p : p \in R(i) :: p)$$

We represent the input intensity values for the pixels in the image by the array of constants $Intensity(p)$.

We define the predicates $INIT$ and $POST$. $INIT$ characterizes the valid initial states of the computation, $POST$ the desired final state, i.e., the state in which each pixel is labeled with the smallest pixel coordinates in its region. More formally, we define $POST$ such that

$$POST \quad \equiv \quad (\forall i : 1 \leq i \leq Nregions :: (\forall p : p \in R(i) :: p \text{ is labeled } w(i))).$$

The key correctness criteria for a region-labeling program are as follows:

1. the characteristics of the problem and solution strategy are represented faithfully by the program structures,
2. the computation always reaches a state satisfying $POST$,
3. after reaching a state satisfying $POST$, subsequent states continue to satisfy $POST$.

In terms of our programming logic, we state the latter two criteria as the Labeling Completion and Labeling Stability properties defined below. As we specify the problem further, we elaborate the first criterion.

Property 1 (Labeling Completion) $INIT \longmapsto POST$

Property 2 (Labeling Stability) stable $POST$

4.2 The Data Structures

To develop a programming solution to the region-labeling problem, we need to define data structures to store the information about the problem. In Swarm, data structures are built from sets of tuples (and transactions). Thus we define the tuple types *has_intensity* and *has_label*: tuple $has\_intensity(P, I)$ associates intensity value I with pixel P; tuple $has\_label(P, L)$ associates label L with pixel P. These types are defined over the set of all pixels in the image. To be faithful to the region-labeling problem, we constrain the relationships among the pixels and tuples by means of invariants.

To simplify the statement of properties and proofs, we implicitly restrict the values of variables that designate region identifiers and pixel coordinates. If not explicitly quantified, region identifier variables (e.g., i) are implicitly quantified over the set of region identifiers 1 through $Nregions$, and pixel coordinate variables (e.g., p and q) over all the pixels *in the image*. Because of this simplification, we do not prove any properties of areas "outside" of the image.

Each pixel p can have only one associated intensity value; this value is equal to the constant $Intensity(p)$ throughout the computation. In terms of the Swarm programming logic, the program must satisfy the intensity invariant defined below. In this invariant the "#" operator denotes the operation of counting the number of elements satisfying the quantification predicate.

Property 3 (Intensity Invariant)

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

The first conjunct of this invariant guarantees that only one intensity attribute is associated with each pixel, i.e., there is a single *has_intensity* tuple for each pixel p. The second conjunct guarantees the constancy of the attribute.

Only one label (*has_label* tuple) can be associated with each pixel. This label is the coordinates of some pixel within the same region. We also require a pixel's label to be no larger than the pixel's own coordinates. These three requirements are captured in the Labeling Invariant stated below.

Property 4 (Labeling Invariant)

invariant $(\#\, q :: has\_label(p, q)) = 1 \wedge$
$(p \in R(i) \wedge has\_label(p, l) \Rightarrow l \in R(i) \wedge w(i) \leq l \leq p)$

Our strategy for solving the region-labeling problem is to exploit the Labeling Invariant to achieve the desired postcondition: initially every pixel is labeled with its own coordinates; each label is decreased toward the $w(i)$ for the region i around the pixel.

For convenience we define the function *excess* on regions such that $excess(i)$ is the total amount the labels on region i exceed the desired labeling (all pixels in the region labeled with the "winning" pixel). More formally,

$$excess(i) \quad = \quad (\Sigma\, p, l : p \in R(i) \wedge has\_label(p, l) :: l - w(i))$$

where the "Σ" and "$-$" operators denote component-wise summation and subtraction of the coordinate pairs.

Using *excess*, the predicate *POST* can be restated as

$$POST \quad \equiv \quad (\forall i : 1 \leq i \leq Nregions :: excess(i) = 0)$$

where 0 denotes the coordinates (0,0).

4.3 The Program and Its Proof

We now state a "dynamic" program to solve the region-labeling problem and show that it meets the correctness criteria. By dynamic, we mean that the contents of the transaction space vary during the computation. An alternative solution with a "static" transaction space is verified in [7] and [8]. A program for an image which is "growing" on one side is studied in [7] and [22].

In the *RegionLabel* program shown in Figure 2, each transaction "carries" a pixel's label to a neighbor; the transaction does not reinsert itself. New transactions are created whenever a label changes. A region's winning label eventually propagates throughout the region.

Verifying the correctness of *RegionLabel* requires the proof of the four properties noted previously: the Intensity Invariant, Labeling Invariant, Labeling Stability, and Labeling Completion properties. The proofs of these properties require us to define and prove other properties.

¶ **Proof of the Intensity Invariant.** Prove

$$(\#\, b :: has\_intensity(p, b)) = 1 \;\wedge\; has\_intensity(p, Intensity(p))$$

is invariant. Clearly the assertion holds at initialization. Also no transaction deletes or inserts *has_intensity* tuples. Hence, the invariant holds for the program. ∎

Because *Label* transactions "carry" the neighbor's label as a parameter rather than examining both *has_label* tuples, the proof of the Labeling Invariant requires a similar property defined for *Label* transactions, the Transaction Label Invariant shown below.

```
program RegionLabel(M, N, Lo, Hi, Intensity :
        1 ≤ M, 1 ≤ N, Lo ≤ Hi, Intensity(ρ : Pixel(ρ)),
        [∀ ρ : Pixel(ρ) :: Lo ≤ Intensity(ρ) ≤ Hi] )
definitions
  [ P, Q, L ::
    Pixel(P) ≡
      [∃ x, y : P = (x, y) :: 1 ≤ x ≤ N, 1 ≤ y ≤ M];
    neighbors(P, Q) ≡
      Pixel(P), Pixel(Q), (0,0) < |P − Q| ≤ (1, 1);
    R_neighbors(P, Q) ≡
      neighbors(P, Q), [∃ ι :: has_intensity(P, ι), has_intensity(Q, ι)]
  ]
tuple types
  [ P, L, I : Pixel(P), Pixel(L), Lo ≤ I ≤ Hi ::
    has_label(P, L);
    has_intensity(P, I)
  ]
transaction types
  [ P, L : Pixel(P), Pixel(L) ::
    Label(P, L) ≡
        [|| δ : P = L, neighbors(P, δ) ::
            ι : has_intensity(P, ι), has_intensity(δ, ι)
                → Label(δ, P)
        ]
      ||
            λ : has_label(P, λ)†, λ > L
                → has_label(P, L)
      ||
        [|| δ : δ ≠ L, neighbors(P, δ) ::
            λ, ι : has_intensity(P, ι), has_intensity(δ, ι), has_label(P, λ), λ > L
                → Label(δ, L)
        ]
  ]
initialization
  [ P : Pixel(P) ::
    has_label(P, P);
    has_intensity(P, Intensity(P));
    Label(P)
  ]
end
```

Figure 2: A Dynamic Region-Labeling Program in Swarm

Property 5 (Transaction Label Invariant)

invariant $p \in R(i) \wedge Label(p,l) \Rightarrow l \in R(i) \ \wedge \ w(i) \leq l$

¶ **Proof of the Transaction Label Invariant.** The only transactions existing initially are the $Label(p,p)$ transactions for each pixel p. Thus the LHS of the implication is *false* for $p \neq l$; for $p = l$ both the LHS and the RHS are *true*. Thus the invariant holds initially. A transaction $Label(p,l)$ can only create transactions of the form $Label(q,l)$ where q is a neighbor of p in the same region. Thus the invariant is preserved. ∎

¶ **Proof of the Labeling Invariant.** We must prove

$$(\#\, q :: has\_label(p,q)) = 1 \ \wedge \ (p \in R(i) \wedge has\_label(p,l) \Rightarrow l \in R(i)) \wedge w(i) \leq l \leq p)$$

is invariant. Initially each pixel p is uniquely labeled p, hence the first conjunct holds. For the initial dataspace the left-hand-side (LHS) of the implications in the second conjuncts is *false* for $p \neq l$; for $p = l$ both the LHS and the RHS (right-hand-side) are *true*. Thus the assertion holds initially. We prove the stability of each conjunct separately.

(1) Consider the first conjunct of the invariant assertion. No transaction deletes a $has\_label(p,*)$ tuple without inserting a $has\_label(p,*)$ tuple, and vice versa. Thus the number of tuples $has\_label(p,*)$ remains constant.

(2) Consider the second conjunct of the invariant. Because of the Transaction Label Invariant, any transaction which changes pixel p's label sets it to the smaller coordinates of another pixel in the same region. ∎

To prove the stability of the "winning" label assignment for the image as a whole (the Labeling Stability property), we first prove the stability of the "winning" label assignment for individual pixels. This more basic property is the Pixel Label Stability property shown below.

Property 6 (Pixel Label Stability) stable $p \in R(i) \wedge has\_label(p, w(i))$

¶ **Proof of Pixel Label Stability.** No transaction increases a label. By the Labeling Invariant no transaction decreases the label of a pixel in region i below $w(i)$. ∎

Given the Pixel Label Stability property we can now prove the Labeling Stability property.

¶ **Proof of Labeling Stability.** We must prove the property stable $POST$. The stability of the assertion $excess(i) = 0$, for any region i, follows from the Pixel Label Stability property for each pixel in the region, the **unless** Conjunction Theorem from [5], and the definition of *excess*.

Applying the Conjunction Theorem again for the regions in the image, we prove the stability of *POST*. ∎

The remaining proof obligation for *RegionLabel* is the Labeling Completion property, a progress property using leads-to. We use the following methodology: (1) focus on the completion of labeling on a region-by-region basis, (2) find and prove an appropriate low-level **ensures** property for pixels in a region, (3) use the **ensures** property to prove the completion of labeling for regions, and (4) combine the regional properties to prove the Labeling Completion property for the image.

To prove Labeling Completion, we first seek to prove $excess(i) \geq 0 \longmapsto excess(i) = 0$. However, a stronger formulation of this property may be easier to prove. Initially there does not exist any transaction which can change a label anywhere in the region. The $Label(p, p)$ transactions initiate the label propagation from each pixel p. However, once transaction $Label(w(i), w(i))$ has executed for each region, there are transactions in the transaction space that decrease $excess(i)$. Moreover, $Label(w(i), w(i))$ is never regenerated by the computation (because of the $\delta \neq P$ restriction in the transaction definition). Thus we seek to prove the property $\neg Label(w(i), w(i)) \wedge excess(i) \geq 0 \longmapsto excess(i) = 0$. We can prove this property using the Incremental Labeling **ensures** property defined later.

We evoke the following metaphor to set up the proof for the Incremental Labeling property. An area of $w(i)$-labeled pixels grows around the $w(i)$ pixel for each region; at the boundary of this growing area is a wavefront of *Label* transactions labeling pixels with $w(i)$.

The following definition is convenient for expression of the properties that follow:

$$\begin{aligned} BOUNDARY(i,p,q) \quad = \quad & p \in R(i) \wedge q \in R(i) \wedge neighbors(p,q) \wedge has\_label(p, w(i)) \wedge \\ & (\exists\, l : l > w(i) :: has\_label(q, l)) \end{aligned}$$

The predicate $BOUNDARY(i, p, q)$ is *true* if and only if p and q are neighboring pixels in region i such that p is labeled with the winning pixel and q has a greater label.

The Incremental Labeling **ensures** property guarantees that, when the assertion $excess(i) > 0$ is *true* under appropriate conditions, there is a transaction in the dataspace which will decrease $excess(i)$.

Property 7 (Incremental Labeling)

$$\neg Label(w(i), w(i)) \wedge BOUNDARY(i, p, q) \wedge 0 < excess(i) = k \quad \textbf{ensures} \quad excess(i) < k$$

From the definition of **ensures** given in Section 3, we must prove:

1. LHS **unless** RHS (where LHS and RHS denote the left- and right-hand-sides of the **ensures** relation);

2. when $LHS \wedge \neg RHS$, there is a transaction in the transaction space which will, when executed, establish the RHS (if not already established).

Accordingly, we divide the proof into an **unless**-part and an **exists**-part.

¶ **Proof of the Incremental Labeling Property (unless part).** All transactions either leave the labels unchanged or decrease one label by some amount. No transaction creates a $Label(w(i), w(i))$ transaction. Hence, LHS **unless** RHS holds for the program. ∎

To prove the existential part of the Incremental Labeling property, we need to show there exists a transaction in the transaction space which, when executed, will decrease $excess(i)$. We evoke the wavefront metaphor described above. The Transaction Wavefront invariant guarantees the existence of $Label(*, w(i))$ transactions along the boundary of the wavefront.

Property 8 (Transaction Wavefront)

$$\textbf{invariant } \neg Label(w(i), w(i)) \wedge BOUNDARY(i, p, q) \Rightarrow Label(q, w(i))$$

To prove this property, we need to prove (1) the wavefront gets started and (2) the wavefront remains in existence until the region is completely labeled with $w(i)$. More formally, we state these concepts as the Startup and Boundary Stability properties defined below.

Property 9 (Startup) $Label(w(i), w(i))$ **unless** $(BOUNDARY(i, p, q) \Rightarrow Label(q, w(i)))$

¶ **Proof of the Startup Property.** To prove this property, we must show

$$\{LHS \wedge \neg RHS\}\ t\ \{LHS \vee RHS\}$$

is *true* for all transactions $t \in$ **TRS**. (LHS and RHS are the left- and right-hand-sides of the **unless** assertion.) The precondition can only be *true* for $p = w(i)$ and q a neighbor of $w(i)$ because of the Winning Label Initiation invariant (proved below). $Label(w(i), w(i))$ creates $Label(q, w(i))$, thus establishing the RHS of the **unless** assertion. All other transactions leave $Label(w(i), w(i))$ *true*. ∎

In the proof above we needed to know that when $Label(w(i), w(i))$ transactions exist the wavefront has not been started; this is the Winning Label Initiation property.

Property 10 (Winning Label Initiation)

$$\textbf{invariant}\ \ Label(w(i), w(i)) \wedge p \in R(i) \wedge p \neq w(i)$$
$$\Rightarrow \neg has\_label(p, w(i)) \wedge \neg Label(p, w(i))$$

¶ **Proof of Winning Label Initiation.** The invariant is trivially *true* for single pixel regions. Consider multi-pixel regions. Both the LHS and RHS are *true* initially. $Label(w(i), w(i))$ falsifies the LHS. No transaction can make the LHS *true*. ∎

Property 11 (Boundary Stability) $\textbf{stable}\ BOUNDARY(i, p, q) \Rightarrow Label(q, w(i))$

¶ **Proof of Boundary Stability.** We need to prove $(\forall t : t \in \textbf{TRS} :: \{I\}\ t\ \{I\})$ where I is the implication in the property definition. We need only consider cases in which I is *true* as the precondition.

For pixels p and q which are not equal-intensity neighbors or for single pixel regions, the predicate $BOUNDARY(i, p, q)$ is always *false*. Thus I is always *true* and, hence, the stable property holds.

Let p and q be neighbor pixels in a multi-pixel region. There are the two cases to consider.
(1) LHS of I *false*. In this case, only transactions which make the LHS *true* can violate the property. Because of the Labeling Invariant and Pixel Label Stability properties, the only transaction that can make $BOUNDARY(i, p, q)$ *true* is $Label(p, w(i))$. This transaction creates $Label(q, w(i))$, thus establishing the RHS of the implication.
(2) Both LHS and RHS of I *true*. Only transactions which falsify the RHS can violate the property. The only transaction that can falsify the RHS is $Label(q, w(i))$. This transaction also changes the label of q to $w(i)$, thus falsifying the predicate $BOUNDARY(i, p, q)$. ∎

¶ **Proof of the Transaction Wavefront Invariant.** We must show the assertion

$$\neg Label(w(i), w(i)) \wedge BOUNDARY(i, p, q) \Rightarrow Label(q, w(i))$$

is invariant. The property holds initially because $INIT \Rightarrow Label(w(i), w(i))$. From the Startup property, we know

$$Label(w(i), w(i)) \quad \textbf{unless} \quad (BOUNDARY(i, p, q) \Rightarrow Label(q, w(i))).$$

From the Boundary Stability property we know

$$(BOUNDARY(i, p, q) \Rightarrow Label(q, w(i))) \quad \textbf{unless} \quad \textbf{false}.$$

Using the Cancellation Theorem for **unless** [5], we conclude the invariant, i.e.,

$$Label(w(i), w(i)) \vee (BOUNDARY(i,p,q) \Rightarrow Label(q, w(i))) \quad \textbf{unless} \quad \textbf{false}.$$

∎

¶ **Proof of the Incremental Labeling Property (exists part).** We must show there is a transaction $t \in \mathbf{TRS}$ such that

$$(PRE \Rightarrow [t]) \ \wedge \ \{PRE\}\ t\ \{excess(i) < k\}$$

where PRE is

$$\neg Label(w(i), w(i)) \ \wedge \ BOUNDARY(i,p,q) \ \wedge \ 0 < excess(i) = k.$$

Because of the Transaction Wavefront invariant, we know $Label(q, w(i))$ is in the transaction space. Execution of this transaction establishes $excess(i) < k$. ∎

Thus the Incremental Labeling property holds for program *RegionLabel*. We now use this property to prove labeling completion for each region in the image. More formally, we prove the Regional Progress property defined below.

Property 12 (Regional Progress)

$$\neg Label(w(i), w(i)) \wedge excess(i) \geq 0 \quad \longmapsto \quad excess(i) = 0$$

The proof of the Regional Progress property needs an additional property, the Boundary Invariant. The Boundary Invariant guarantees the existence of the boundary between the completed (labeled with $w(i)$) and uncompleted areas.

Property 13 (Boundary Invariant)

$$\textbf{invariant}\ excess(i) > 0 \Rightarrow (\exists\, p, q :: BOUNDARY(i,p,q))$$

¶ **Proof of the Boundary Invariant.** For single pixel regions $excess(i) = 0$ holds invariantly; hence the Boundary Invariant holds.

Consider multi-pixel regions. Initially $excess(i) > 0$. Because of the Pixel Label Stability property, the invariance of $has\_label(w(i), w(i))$ is clear. When $excess(i) > 0$, because of the definition of *excess* and the Labeling Invariant, there must be some pixel x in region i which has a label greater than $w(i)$. Thus along any neighbor-path from x to $w(i)$ within region i, there must be two neighbor pixels, p and q, such that p has label $w(i)$ and q a larger label. ∎

¶ **Proof of the Regional Progress Property.** The progress property $excess(i) = 0 \longmapsto excess(i) = 0$ is obvious, thus the only remaining proof obligation is

$$\neg Label(w(i), w(i)) \wedge excess(i) > 0 \quad \longmapsto \quad excess(i) = 0.$$

From the Incremental Labeling progress property we know

$$\neg Label(w(i), w(i)) \wedge BOUNDARY(i, p, q) \wedge 0 < excess(i) = k \quad \textbf{ensures} \quad excess(i) < k.$$

Because of the Boundary Invariant we also know

$$excess(i) > 0 \Rightarrow (\exists\, p, q :: BOUNDARY(i, p, q))$$

Using the disjunction rule for leads-to over the set of neighbor pixels p and q in region i, we deduce

$$\neg Label(w(i), w(i)) \wedge 0 < excess(i) = k \quad \longmapsto \quad excess(i) < k.$$

Since $\neg Label(w(i), w(i))$ is stable, we can rewrite the assertion above as

$$\neg Label(w(i), w(i)) \wedge excess(i) > 0 \wedge excess(i) = k \longmapsto$$
$$(\neg Label(w(i), w(i)) \wedge excess(i) > 0 \wedge excess(i) < k) \vee excess(i) = 0.$$

The metric $excess(i)$ is well-founded. Thus, using the induction principle for leads-to, we conclude the Regional Progress property. ∎

Given the Regional Progress and Labeling Stability properties, the proof the Labeling Completion property is straightforward.

¶ **Proof of Labeling Completion.** Prove the assertion $INIT \longmapsto POST$. Clearly,

$$INIT \;\Rightarrow\; (\forall i :: excess(i) \geq 0 \wedge Label(w(i), w(i))).$$

Hence, for each region i,

$$INIT \quad \textbf{ensures} \quad excess(i) \geq 0 \wedge Label(w(i), w(i)).$$

From the transaction definition, it is easy to see

$$Label(w(i), w(i)) \quad \textbf{ensures} \quad \neg Label(w(i), w(i)).$$

Hence,

$$Label(w(i), w(i)) \wedge excess(i) \geq 0 \;\; \textbf{ensures}$$
$$excess(i) = 0 \vee (\neg Label(w(i), w(i)) \wedge excess(i) > 0).$$

From the Regional Progress property,

$$\neg Label(w(i), w(i)) \wedge excess(i) \geq 0 \quad \longmapsto \quad excess(i) = 0.$$

The Cancellation Theorem for leads-to [5] allows us to deduce

$$Label(w(i), w(i)) \wedge excess(i) \geq 0 \quad \longmapsto \quad excess(i) = 0.$$

The Labeling Stability property, the Completion Theorem for leads-to [5], and the transitivity of leads-to allow us to conclude $INIT \longmapsto POST$. ∎

Above we have shown program *RegionLabel* satisfies the four criteria for correctness of region-labeling programs. However, we can also prove this program terminates. We define the predicate **termination** such that

$$\textbf{termination} \quad \equiv \quad (\forall p, l :: \neg Label(p, l)).$$

Since we have already established the Labeling Completion property, we need only prove $POST \longmapsto$ **termination.** Again we can prove this leads-to property using an **ensures** property, the Transaction Flushing property below.

Property 14 (Transaction Flushing)

$$POST \wedge Label(p, l) \wedge 0 < (\#\, q, m :: Label(q, m)) = k \ \textbf{ensures} \ (\#\, q, m :: Label(q, m)) < k$$

¶ **Proof of the Transaction Flushing Property.** By the Transaction Label Invariant, we know $q \in R(i) \wedge Label(q, m) \Rightarrow m \in R(i) \wedge w(i) \leq m$ The $POST$ predicate means all *Label* transactions will fail. Thus the RHS of the **ensures** property is established. ∎

¶ **Proof of Termination.** $POST \longmapsto$ **termination.** The Transaction Flushing property and the disjunction rule for leads-to allow us to deduce

$$POST \ \wedge \ 0 < (\#\, q, m :: Label(q, m)) = k \quad \longmapsto \quad (\#\, q, m :: Label(q, m)) < k.$$

The count of the transactions in the transaction space is a well-founded metric, thus we deduce $POST \longmapsto$ **termination** by induction. ∎

5 Conclusion

In this paper, we have specified a proof system for the shared dataspace programming notation called Swarm. To our knowledge, this is the first such proof system for a shared dataspace

language. To illustrate the proof system, we used it to verify the correctness of a program for labeling the connected, equal-intensity regions of a digital image.

Like UNITY, the Swarm proof system uses an assertional programming logic which relies upon proof of program-wide properties, e.g., global invariants and progress properties. We define the Swarm logic in terms of the same logical relations as UNITY (**unless**, **ensures**, and **leads-to**), but must reformulate several of the concepts to accommodate Swarm's distinctive features. We define a proof rule for transaction statements to replace UNITY's well-known rule for multiple-assignment statements, redefine the **ensures** relation to accommodate the creation and deletion of transaction statements, and replace UNITY's use of fixed-point predicates with other methods for determining program termination. We have constructed our logic carefully so that most of the theorems developed for UNITY can be directly adapted to the Swarm logic.

We have generalized the programming logic presented in this paper to handle another unique feature of Swarm, the synchronic group [7, 22]. Synchronic groups are dynamically constructed groups of transactions which are executed synchronously as if they were a single atomic transaction. We believe synchronic groups enable novel approaches to the organization of concurrent computations, particularly in situations where the desired computational structure is dependent upon the data.

The Swarm programming model, notation, and logic provide a foundation for several other promising research efforts. We believe visualization can play a key role in exploration of concurrent computation. Companion papers [19, 20] outline a declarative approach to visualization of the dynamics of program execution—an approach which represents properties of an executing program's state as visual patterns on a graphics display. We are also studying the relationship of Swarm to other approaches, e.g., rule-based systems [8], UNITY, and Linda. Swarm is proving to be an excellent research vehicle.

Acknowledgements

This work was supported by the Department of Computer Science, Washington University, Saint Louis, Missouri. The authors express their gratitude to Jerome R. Cox, department chairman, for his support and encouragement. We thank Jayadev Misra, Jan Tijmen Udding, Ken Cox, Howard Lykins, Wei Chen, Will Gillett, Michael Kahn, and Liz Hanks for their suggestions

concerning this article. We also thank the Department of Computer and Information Science at The University of Mississippi for enabling the first author to continue this work.

References

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

[2] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[3] 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.

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

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

[6] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, May 1979.

[7] 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.

[8] H. C. Cunningham and G.-C. Roman. Toward formal verification of rule-based systems: A shared dataspace perspective. Technical Report WUCS–89–28, Washington University, Department of Computer Science, St. Louis, Missouri, June 1989.

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

[10] N. Francez. *Fairness.* Springer-Verlag, New York, 1986.

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

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

[13] S. S. Lam and A. U. Shankar. A relational notation for state transition systems. Technical Report TR–88–21, University of Texas at Austin, Department of Computer Sciences, Austin, Texas, May 1988. Revised August 1988.

[14] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent computation. In S. Even and O. Kariv, editors, *ICALP '81: Automata, Languages, and Programming*, pages 264–277, New York, 1981. Springer-Verlag. Lecture Notes in Computer Science #115.

[15] G. LeLann. Distributed systems, towards a formal approach. In *Information Processing 77*, pages 155–160. North-Holland, New York, 1977.

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

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

[18] 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.

[19] G.-C. Roman and K. C. Cox. A declarative approach to visualizing concurrent computations. *Computer*, 22(10):25–36, October 1989.

[20] G.-C. Roman and K. C. Cox. Declarative visualization in the shared dataspace paradigm. In *Proceedings of the 11th International Conference on Software Engineering*, pages 34–43. IEEE, May 1989.

[21] 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.

[22] G.-C. Roman and H. C. Cunningham. The synchronic group: A concurrent programming concept and its proof logic. Technical Report WUCS–89–43, Washington University, Department of Computer Science, St. Louis, Missouri, October 1989.

[23] G.-C. Roman and H. C. Cunningham. Programming metaphors supported by a shared dataspace model of concurrency. Technical Report WUCS–90–1, Washington University, Department of Computer Science, January 1990. To appear in *IEEE Transactions on Software Engineering.*

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