A UNITY-style Programming Logic for Shared Dataspace Programs

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

June 14, 2003

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 proof system to reason about a Swarm program to label the equal-intensity regions of a digital image.

Index Terms

Concurrent languages, concurrent programming, program verification, programming logic, UNITY, shared dataspace, Swarm

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 [11]. 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 [12], 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—logic-based languages (e.g., Prolog [23]), production rule systems (e.g., OPS5 [3]), and tuple space languages (e.g., Linda [1, 4]). The content-addressable approach seems to encourage higher degrees of concurrency and more flexible connections among components. Program verification techniques for such languages are just beginning to be researched.

Because we desire high degrees of concurrency while preserving the programming and verification convenience 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 [17], 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* [20]. The design of Swarm was influenced by the previous work on Linda [1, 4], Associons [15, 16], 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 paradigm. We have defined the Swarm programming notation and specified a formal operational model based on a state-transition approach [7, 20]. Our research is exploring the implications of the shared dataspace approach and the Swarm model for algorithm development and programming methodology [21]. 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, 22]. 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 [18, 19].

In this paper, we specify a proof system for Swarm similar in style to that of UNITY and illustrate its use by proving the correctness of a program to label the 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 many Hoare-style 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 has 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; this program first appeared in [20].

2 The Swarm Notation

In this section we introduce the Swarm notation and model by discussing a simple producer/consumer program. We first present a program expressed in a familiar imperative notation—a concurrent dialect of Dijkstra's Guarded Commands (GC) [9] language. We then construct a Swarm program to solve the same problem.

Consider a simple producer/consumer scenario where a producer process sends a sequence of values to a consumer process through a buffer. We want to construct a program which has the following characteristics:

1. The buffer never contains more than one value.
2. If the buffer is empty, then the next value will eventually be placed into the buffer by the producer.
3. If the buffer contains a value, then the value will eventually be removed from the buffer by the consumer.

Requirement 1 is a safety property that the program must satisfy and requirements 2 and 3 are progress properties.

A simple GC program having these characteristics is not difficult to state. Suppose process *Producer* sends a sequence of integer values to process *Consumer* via one-place buffer buf. If the integer variable x has been initialized to some integer value and the boolean flag *empty* to the value **true**, then we can express this two-process program as follows:

Producer ::
do $empty \longrightarrow buf, empty, x := x, \textbf{false}, x + 1$
[] $\neg empty \longrightarrow \textbf{skip}$
od

Consumer ::
do $\neg empty \longrightarrow y, empty := buf, \textbf{true}$
[] $empty \longrightarrow \textbf{skip}$
od

We assume that each assignment statement and loop test is executed atomically. We further assume that any execution of this two-process program can be modeled accurately by a fair interleaving of the atomic actions of the two processes.

Swarm is a shared dataspace programming model. Instead of expressing a computation in terms of a group of named variables, Swarm uses a finite 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 producer/consumer algorithm in Swarm? First, consider the representation of the data in the algorithm. To represent the buffer we introduce the tuple type buf. We let the single component of a buf tuple hold an integer value currently in-transit between the producer and the consumer. Since tuples of this type need only exist in the dataspace when there is a value that has been produced but not yet consumed, we do not need to introduce a Swarm analogue of the GC program's boolean flag *empty*.

Now we can focus on the representation of the processing actions. Consider the consumer process. We can specify the removal of a value from the buffer in the following way:

$$v : buf(v) \longrightarrow buf(v)\dagger$$

In this notation, the part of the construct to the left of the "$\longrightarrow$" is the query; the part to the right is the action. The identifier v designates a variable local to the query-action pair.

The execution of a Swarm query is similar to the evaluation of a clause in Prolog [23]. The query shown in the previous paragraph causes a search of the dataspace for a tuple of type buf. If one or more solutions to the query are found, then one of the solutions is chosen nondeterministically, the matched value is bound to the local variable v, and the action is performed with this binding. If no solution is found, then the query is said to fail and the specified action is not taken.

In the query-action pair shown above, the action part specifies that the tuple $buf(v)$, where v has the value bound by the query, is to be deleted from the dataspace. Swarm uses the † symbol to mark a tuple for deletion. An unmarked tuple form appearing in the action part indicates that the corresponding tuple is to be inserted. Several tuples may be deleted and inserted in one action; the appropriate tuple forms are separated by commas. Although the execution of a query-action pair is atomic, the effect of an action is as if all deletions are performed first, then all insertions.

In Swarm there is no concept of a process and there are no sequential programming constructs. Only transactions are available. Like data tuples, transactions are represented as tuple-like entities

in the dataspace. (We thus partition the dataspace into two finite subsets, the tuple space and the transaction space.) A transaction has a type name and a finite sequence of values called parameters. Transaction instances can be queried and inserted in the same way that data tuples are, but cannot be explicitly deleted.

The consumer process can be expressed by a transaction of type *Consumer* having one parameter, the value of the integer most recently read from the buffer. If the buffer contains a value, the transactions of this type can read the value, delete the buf tuple, and insert the appropriate *Consumer* transaction needed to continue the computation. If the buffer is empty, then the transaction reinserts itself—thus waiting for a value to appear in the buffer. We define the *Consumer* transaction type as follows:

$$
\begin{array}{lll}
Consumer(Y) \equiv & & \\
\quad & v : buf(v) & \longrightarrow buf(v)\dagger,\ Consumer(v) \\
\| & [\forall\, v :: \neg buf(v)] & \longrightarrow Consumer(Y)
\end{array}
$$

Note that we use the values of parameters as constants throughout the body of transaction. We call each individual query-action pair a *subtransaction* and the overall parallel construct connected by ‖-operators a *transaction*.

We model the execution of a Swarm program in the following way. 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. 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 and transactions are inserted. Unless the transaction explicitly reinserts itself into the dataspace, it is deleted as a by-product of its own execution—regardless of the success or failure of its component queries. Program execution continues until there are no transactions remaining in the dataspace.

Taking advantage of two other Swarm features, we can also express the *Consumer* transaction in the following equivalent, but more compact, way:

$$
\begin{array}{lll}
Consumer(Y) \equiv & & \\
\quad & v : buf(v)\dagger & \longrightarrow Consumer(v) \\
\| & \textbf{NOR} & \longrightarrow Consumer(Y)
\end{array}
$$

In this transaction type definition we have moved the † operator into the query part. As before, the tuple marked by the † is deleted if the entire query is successful. The **NOR** used in this definition is one of the special built-in predicates provided in Swarm. A **NOR** predicate succeeds if *none*

```
program P1_C1_B1
tuple types
    [V ::
        buf(V)
    ]
transaction types
    [X, Y ::
        Producer(X) ≡
                    v : buf(v)     ⟶  Producer(X)
                ‖   NOR            ⟶  buf(X), Producer(X+1) ;

        Consumer(Y) ≡
                    v : buf(v)†    ⟶  Consumer(v)
                ‖   NOR            ⟶  Consumer(Y)
    ]
initialization
        Producer(0) ; Consumer(0)
end
```

Figure 1: A Simple Producer/Consumer Program with a One-Place Buffer

of the other queries involving *only* regular predicates succeeds. In this example, **NOR** succeeds if the query involving $buf(v)$, the only regular query in the transaction, fails. The special built-in predicates are **OR**, **AND**, **NOR**, and **NAND**, meaning *any*, *all*, *none*, and *not-all*, respectively.

In a manner similar to the *Consumer* transaction, we can define a *Producer* transaction type to represent the producer process. Transactions of this type have one parameter, the value of the next integer to place into the buffer. The *Producer* transaction repeatedly "waits" until the buffer is empty, then inserts the next value. The full producer/consumer program is shown in Figure 1. Note that the program is initialized with the tuple space empty and transaction instances $Producer(0)$ and $Consumer(0)$ in the transaction space.

For this producer/consumer computation to make progress, i.e., pass a sequence of integer values from producer to consumer through the buffer, one constraint must be placed upon the selection of transactions for execution: every transaction present in the transaction space at any point in the computation must eventually be executed. That is, Swarm transactions must be selected for execution in a (weakly) fair manner [10].

To illustrate other characteristics of the Swarm model and notation, we can now modify the producer/consumer program in various ways. Figure 2 shows a producer/consumer program which

program $P1\_C1\_Bn(N : N > 0)$
tuple types
$[I, V : 0 \leq I < N ::$
$buf(I, V)$
$]$
transaction types
$[\, I, X, Y : 0 \leq I < N ::$
$Producer(I, X) \equiv$
$v : buf(I, v) \longrightarrow Producer(I, X)$
$\| \quad \mathbf{NOR} \longrightarrow buf(I, X),\ Producer((I{+}1) \bmod N, X{+}1)\ ;$

$Consumer(I, Y) \equiv$
$v : buf(I, v)\dagger \longrightarrow Consumer((I{+}1) \bmod N, v)$
$\| \quad \mathbf{NOR} \longrightarrow Consumer(I, Y)$
$]$
initialization
$Producer(0, 0)\ ;\ Consumer(0, 0)$
end

Figure 2: A Producer/Consumer Program with an N-Place Buffer

uses an N-place buffer. For this program we add a component to the buf tuple to record the buffer position at which the value is stored. A parameter is also added to each of the transactions to record the next buffer position to be operated upon. Note that the values of the first components of each are restricted to the range 0 through $N - 1$ inclusive.

To enable several producers to supply a single consumer through an N-place buffer, the next sequence number for an element of the buffer must be shared by all of the producer transactions. In Figure 3 we add a tuple of type *next* to coordinate the sequencing among the producers. (A comma separating predicates is interpreted as a logical **and** connective.) To allow differentiation among the producers, we also add parameter K to both the *Producer* transaction and the buf tuple.

We might ask whether the *Consumer* transaction in Figure 3 receives a fair merge of the sequences generated by the producers. Unfortunately, the fairness in selection of a transaction for execution is insufficient to guarantee fair merging of the sequences. For example, consider two producers, $P1$ and $P2$, consumer C, and a one-place buffer ($N = 1$). The cyclical scheduling $P1 \rightarrow P2 \rightarrow C$ of the three transactions satisfies the fairness requirement for transaction scheduling, but C never receives a value generated by $P2$. Similar problem schedules can be found for any

program $P2\_C1\_Bn(N : N > 0)$
tuple types
$[I, K, V : 0 \leq I < N,\ K \geq 0 ::$
$buf(I, K, V)$;
$next(I)$
$]$
transaction types
$[\, I, K, X, Y : 0 \leq I < N,\ K \geq 0 ::$
$Producer(K, X) \equiv$
$i, j, v : next(i), buf(i, j, v) \longrightarrow Producer(K, X)$
$\| \quad i : \mathbf{NOR}, next(i)\dagger \longrightarrow buf(i, K, X), Producer(K, X{+}1),$
$next((i{+}1) \bmod N)$;

$Consumer(I, Y) \equiv$
$j, v : buf(I, j, v)\dagger \longrightarrow Consumer((I{+}1) \bmod N, v)$
$\| \quad \mathbf{NOR} \longrightarrow Consumer(I, Y)$
$]$
initialization
$next(0)$; $Producer(1, 0)$; $Producer(2, 0)$; $Consumer(0, 0)$
end

Figure 3: Two Producers Using an N-Place Buffer

bounded-length buffer. However, if an unbounded buffer is used, the fairness of the transaction scheduling will guarantee that the consumer will receive a fair merge of the sequences from the two producers.

3 A Programming Logic

This section presents an assertional programming logic for Swarm, building upon the formal model given in [7] and [20]. The model and logic presented here focus on the Swarm notation as described in the previous section.

Execution Model. 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**.

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 $\mathbf{e}$ denote one of these sequences. Each element

$\mathbf{e}_i$, $i \geq 0$, of sequence $\mathbf{e}$ is an ordered pair consisting of a program dataspace $\mathbf{Ds.e}_i$ and a set $\mathbf{Sg.e}_i$ containing a single transaction chosen from $\mathbf{Tr.Ds.e}_i$. (If $\mathbf{Tr.Ds.e}_i = \emptyset$, then $\mathbf{Sg.e}_i = \emptyset$.)

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 $\mathbf{e}$, as characterized above, which satisfy the following criteria:

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

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 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 $p(d)$ to denote the evaluation of predicate p with respect to dataspace d and the notation $(p \wedge \neg q)(\mathbf{e}_i)$ to denote the evaluation of the predicate $p \wedge \neg q$ with respect to $\mathbf{Ds.e}_i$. Below we also use the notation $[t]$ to denote the predicate "transaction instance t is in the transaction space."

Transaction Rule. UNITY's basic construct is the assignment statement. Hence, its programming logic is built around the well-known proof rule for the assignment statement. Similarly, the programming logic for Swarm is built around a proof rule for its basic construct—the trans-

action. However, because of the nature of the transaction, the formulation of the transaction rule differs from that of the assignment.

Swarm transaction statements are nondeterministic; execution of a transaction from a given state (i.e., dataspace) may result in any one of potentially many next states. This arises because a transaction's query may have many possible solutions with respect to a given dataspace. When multiple solutions exist, the execution mechanism chooses one of the solutions nondeterministically. (The Swarm model requires that transactions be selected for execution fairly, but does not require that the choice among alternative solutions to a transaction's query be fair.)

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:

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

Informally this means that, whenever the dataspace satisfies the precondition predicate p and transaction instance t is in the transaction space, all dataspaces which can result from execution of transaction t satisfy postcondition q. In terms of the execution sequences this rule means

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

Safety Properties. As in UNITY's logic, the basic safety properties of a program are specified in terms of **unless** relations. The Swarm **unless** rule mirrors the UNITY rule [14]:

$$\frac{\langle \forall\, t : t \in \mathbf{TRS} :: \{p \wedge \neg q\}\ t\ \{p \vee q\} \rangle}{p\ \mathbf{unless}\ q}$$

Informally, p **unless** q means that, if predicate 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, the premise of this rule means

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

From this we can deduce

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

In other words, p **unless** q means that, if p becomes *true*, then 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 key notions of the proof theory. If a stable predicate becomes *true* at any point during an execution of a program, then it continues to hold throughout the remainder of the execution. An invariant is a stable predicate which is *true* initially and, hence, holds invariantly throughout the execution of the program. Both can be defined easily as follows:

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

The predicate $INIT$ is a predicate which characterizes the valid initial states of the program. Note that the **stable** rule can be restated as

$$\langle \forall\, t : t \in \textbf{TRS} :: \{p\}\ t\ \{p\} \rangle.$$

That is, stable property p is preserved by all possible transactions of the program.

If both p and $\neg p$ are stable, we call p a constant property. That is,

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

For an example of a safety property, consider the producer/consumer program $P1\_C1\_B1$ from Section 2. The first requirement given was that "the buffer never contains more than one value." Using the Swarm logic, we can state this requirement formally as follows:

$$\textbf{invariant}\ \ \langle \#v :: buf(v) \rangle \leq 1$$

In this invariant the # operator denotes the operation of counting the number of elements (buf tuples) satisfying the quantification predicate (**true** is this case).

To prove this predicate (call it p) to be invariant we must:

- show that $INIT \Rightarrow p$ holds,
- show that p is stable, i.e., for all transactions t, $\{p\}t\{p\}$ holds.

Because the tuple space is empty at initialization, $INIT \Rightarrow p$ clearly holds. Consider the stability part of the proof. *Consumer* transactions never insert buf tuples. Neither do *Producer* transactions when buf tuples are already present in the tuple space and then only a single buf tuple. Thus p is an invariant of the $P1\_C1\_B1$ program.

Progress Properties. As with the UNITY logic, program progress (liveness) properties are stated in terms of the **ensures** and **leads-to** relations. The Swarm logic uses an **ensures** relation to characterize small-scale progress, i.e., the progress achieved by the execution of individual transactions. The logic uses a **leads-to** relation to characterize larger-scale computational progress, i.e., the progress achieved by sequence of transaction executions.

UNITY programs consist of a static set of assignment 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 UNITY **ensures** relation. For a given program in the Swarm notation considered in this paper, the **ensures** relation is defined with the following rule of inference:

$$\frac{p \textbf{ unless } q \,, \langle \exists\, t : t \in \textbf{TRS} :: (p \wedge \neg q \Rightarrow [t]) \wedge \{p \wedge \neg q\}\ t\ \{q\}\rangle}{p \textbf{ ensures } q}$$

Informally, p **ensures** q, 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*. Because of the fairness criterion for Swarm execution sequences, the second part of this definition guarantees q will eventually become *true*. The only way transaction t can be removed from the dataspace is as a by-product of its execution; the fairness criterion guarantees that if transaction t is in the transaction space it will eventually be executed.

In terms of the execution sequences, the premise of the **ensures** rule means

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

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

As with UNITY program proofs, the **leads-to** relation, denoted by the symbol $\longmapsto$, is commonly used in Swarm 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, $$\frac{\langle \forall\ m \ : \ m \ \in \ W \ :: \ p(m) \ \longmapsto \ q\rangle}{\langle \exists\ m \ : \ m \ \in \ W \ :: \ p(m)\rangle \ \longmapsto \ q} \qquad (\textbf{disjunction})$$

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

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

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*.

For an example of a progress property, again consider the program $P1\_C1\_B1$ from Section 2. The second requirement given was that "if the buffer is empty, then a new value will eventually be placed into the buffer by the producer." Using the Swarm logic, we can formally state the essence of this requirement as follows:

$$Producer(i) \wedge \langle \#v :: buf(v) \rangle = 0 \;\longmapsto\; buf(i) \wedge \langle \#v :: buf(v) \rangle = 1.$$

In this property, variable i is universally quantified over all integers. For convenience, we call the predicates on the left- and right-hand sides of this leads-to $p(i)$ and $q(i)$, respectively.

The above leads-to property can be directly derived from the property $p(i)$ **ensures** $q(i)$. To prove this ensures property we must:

- show that $p(i)$ **unless** $q(i)$ holds,
- show that there is a transaction instance t such that $p(i) \wedge \neg q(i) \Rightarrow [t]$ and $\{p(i) \wedge \neg q(i)\}\ t\ \{q(i)\}$ hold.

The unless part of the proof is easy to see. Neither *Producer* nor *Consumer* transactions falsify $p(i)$ without making $q(i)$ true.

For the existential part of the proof, we must show prove that, when $p(i)$ holds, there is always a transaction in the transaction space which will always establish $q(i)$ upon execution. But the $Producer(i)$ transaction will do this. Since $p(i) \Rightarrow Producer(i)$, we thus conclude the existential part.

Termination. UNITY programs do not terminate in the traditional sense. However, most useful UNITY programs will reach a *fixed point*, i.e., a state in which further statement executions will not change the values of any of the variables. A fixed-point predicate, *FP*, can be derived syntactically from the program text.

Swarm programs may also reach fixed points, but, unfortunately, *FP* predicates cannot be defined syntactically from the transaction definitions. Thus, proofs of Swarm programs must formulate program postconditions differently—often in terms of other **stable** properties. But,

unlike UNITY programs, we have defined termination in the Swarm model: a Swarm program terminates when the transaction space becomes empty. A termination predicate **termination** can thus be defined as follows:

$$\textbf{termination} \quad \equiv \quad \langle \forall\, t : t \in \textbf{TRS} :: \neg[t] \rangle.$$

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 nontrivial Swarm program. The program shown is a solution to a region-labeling problem, one of the alternative Swarm solutions given in [20]. 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 variant of the classical leader election problem [6, 13]. 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 program 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 $R(i)$ denotes all of the pixels in region i, i.e.,

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

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:

$$w(i) = \langle \mathbf{min}\ p : p \in R(i) :: p \rangle$$

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

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

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** $POST$

In this section we specify a program to solve the region-labeling problem. The *RegionLabel* program uses a dynamic set of transactions. 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.

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.

Property 3 (Intensity Invariant)

$$\textbf{invariant} \quad \langle \#\, b :: has\_intensity(p, b) \rangle = 1 \ \wedge \ has\_intensity(p, Intensity(p))$$

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)

$$\textbf{invariant} \quad \langle \#\, q :: has\_label(p,q) \rangle = 1 \wedge \\ (p \in R(i) \wedge has\_label(p,l) \Rightarrow l \in R(i) \ \wedge \ w(i) \leq l \leq p)$$

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.

We can now restate the predicate $POST$ in terms of the data structures as follows:

$$POST \quad \equiv \quad \langle \forall\, i : 1 \leq i \leq Nregions :: \langle \forall\, p : p \in R(i) : has\_label(p, w(i)) \rangle \rangle$$

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,

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

where the "Σ" and"$-$" operators denote component-wise summation and subtraction of the coordinate pairs. Using $excess$, the predicate $POST$ can be restated as

$$POST \quad \equiv \quad \langle \forall\, i : 1 \leq i \leq Nregions :: excess(i) = \mathbf{0} \rangle$$

where $\mathbf{0}$ denotes the coordinates (0,0).

We consider a region labeling program which uses the tuple types *has_intensity* and *has_label* to be correct if it satisfies the Labeling Completion, Labeling Stability, Intensity Invariant, and Labeling Invariant properties.

4.3 A Swarm Program

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 studied in [7] and [8]. A program for an image which is "growing" on one side is studied in [7] and [22].

The program *RegionLabel* (shown in Figure 4) initially associates a transaction instance with each pixel in the image. When an executing transaction detects that the pixel's label may need to be propagated to a neighbor, it inserts transactions to "carry" the label to that neighbor. When the image is labeled as desired, all transactions will "fail"—resulting in termination of the program.

program $RegionLabel(M, N, Lo, Hi, Intensity :$
$\quad 1 \leq M,\ 1 \leq N, Lo \leq Hi,\ Intensity(\rho : Pixel(\rho)),$
$\quad [\forall\, \rho : Pixel(\rho) :: Lo \leq Intensity(\rho) \leq Hi]\ )$
definitions
$[\, P, Q, L ::$
$\quad Pixel(P)\ \equiv$
$\quad\quad [\exists\, x, y : P = (x, y) :: 1 \leq x \leq N,\ 1 \leq y \leq M]\ ;$
$\quad neighbors(P, Q)\ \equiv$
$\quad\quad Pixel(P),\ Pixel(Q),\ (0, 0) < |P - Q| \leq (1, 1)\ ;$
$\quad R\_neighbors(P, Q)\ \equiv$
$\quad\quad neighbors(P, Q),\ [\exists\, \iota :: has\_intensity(P, \iota), has\_intensity(Q, \iota)]$
$]$
tuple types
$[\, P, L, I : Pixel(P), Pixel(L), Lo \leq I \leq Hi ::$
$\quad has\_label(P, L)\ ;$
$\quad has\_intensity(P, I)$
$]$
transaction types
$[\, P, L : Pixel(P),\ Pixel(L) ::$
$\quad Label(P, L)\ \equiv$
$\quad\quad [\!|\ \delta : P = L,\ neighbors(P, \delta) ::$
$\quad\quad\quad \iota : has\_intensity(P, \iota),\ has\_intensity(\delta, \iota)$
$\quad\quad\quad\quad \rightarrow\ Label(\delta, P)$
$\quad\quad]$
$\quad\quad \|$
$\quad\quad\quad \lambda : has\_label(P, \lambda)\dagger,\ \lambda > L$
$\quad\quad\quad\quad \rightarrow\ has\_label(P, L)$
$\quad\quad \|$
$\quad\quad [\!|\ \delta : \delta \neq L,\ neighbors(P, \delta) ::$
$\quad\quad\quad \lambda, \iota : has\_intensity(P, \iota),\ has\_intensity(\delta, \iota),\ has\_label(P, \lambda),\ \lambda > L$
$\quad\quad\quad\quad \rightarrow\ Label(\delta, L)$
$\quad\quad]$
$]$
initialization
$[\, P : Pixel(P) ::$
$\quad has\_label(P, P)\ ;$
$\quad has\_intensity(P, Intensity(P))\ ;$
$\quad Label(P, P)$
$]$
end

Figure 4: A Dynamic Region-Labeling Program in Swarm

A *Label* transaction has three groups of subtransactions. The first group starts the propagation of the pixel's label to neighbors in the same region. When a smaller label is propagated to the transaction's associated pixel from a neighbor, the second group relabels the pixel. When the associated pixel is relabeled, the third group of subtransactions propagates the new label to neighbors by inserting the appropriate transaction instances. In Figure 4 we use Swarm's subtransaction generator construct to specify the groups of subtransactions in a compact form. We also use predicates defined in the **definitions** section to simplify expression of the transaction queries.

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

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

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.

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 left-hand-side (LHS) of the invariant implication is *false* for $p \neq l$; for $p = l$ both the LHS and the RHS (right-hand-side) 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

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

is invariant. Initially each pixel p is uniquely labeled p, hence the first conjunct holds. For the initial dataspace the LHS of the implication in the second conjunct is *false* for $p \neq l$; for $p = l$

both the LHS and the RHS 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 $excess(i) = \mathbf{0}$, 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 $excess(i) \geq \mathbf{0} \longmapsto excess(i) = \mathbf{0}$. 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 $\neg Label(w(i), w(i)) \wedge excess(i) \geq \mathbf{0} \longmapsto excess(i) = \mathbf{0}$. 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:

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

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 $excess(i) > \mathbf{0}$ is *true* under appropriate conditions, there is a transaction in the dataspace which will decrease $excess(i)$.

Property 7 (Incremental Labeling)

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

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)

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)

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) **stable** $BOUNDARY(i, p, q) \Rightarrow Label(q, w(i))$

¶ Proof of Boundary Stability. We need to prove $\langle \forall\, t : t \in \textbf{TRS} :: \{I\}\ t\ \{I\} \rangle$ 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 $t \in \textbf{TRS}$ such that

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

where PRE is

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

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)

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

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)

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

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

Consider multi-pixel regions. Initially $excess(i) > \mathbf{0}$. Because of the Pixel Label Stability property, the invariance of $has\_label(w(i), w(i))$ is clear. When $excess(i) > \mathbf{0}$, 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 $excess(i) = \mathbf{0} \longmapsto excess(i) = \mathbf{0}$ is obvious, thus the only remaining proof obligation is

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

From the Incremental Labeling progress property we know

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

Because of the Boundary Invariant we also know

$$excess(i) > \mathbf{0} \Rightarrow \langle \exists\, p, q :: BOUNDARY(i, p, q) \rangle$$

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

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

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

$$\begin{array}{l} \neg Label(w(i), w(i)) \wedge excess(i) > \mathbf{0} \wedge excess(i) = k \ \longmapsto \\ \quad (\neg Label(w(i), w(i)) \wedge excess(i) > \mathbf{0} \wedge excess(i) < k) \vee excess(i) = \mathbf{0}. \end{array}$$

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,

$$INIT \;\Rightarrow\; \langle \forall\, i :: excess(i) \geq \mathbf{0} \wedge Label(w(i), w(i)) \rangle.$$

Hence, for each region i,

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

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,

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

From the Regional Progress property,

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

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

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

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

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

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)

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

¶ 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

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

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 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 [18, 19] 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 the editor, the referees, 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] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[12] 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.

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

[14] J. Misra. Soundness of the substitution axiom. Notes on UNITY 14–90, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, March 1990.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] G.-C. Roman and H. C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. Technical Report WUCS–90–1, Washington University, Department of Computer Science, St. Louis, Missouri, February 1990. To appear in *IEEE Transactions on Software Engineering*.

[22] G.-C. Roman and H. C. Cunningham. The synchronic group: A concurrent programming concept and its proof logic. In *Proceedings of the 10th International Conference on Distributed Computing Systems*. IEEE, May 1990.

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