Report Number: WUCS-89-05

1989-03-01

A UNITY-Style Programming Logic for a Shared Dataspace Language

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 extensive program verification research. This paper specifies a proof system for the shared dataspace language Swarm - a proof system similar in style to that of UNITY. The paper then uses the... Read complete abstract on page 2.

A UNITY-Style Programming Logic for a Shared Dataspace Language

H. Conrad Cunningham and Gruia-Catalin Roman

Complete 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 extensive program verification research. This paper specifies a proof system for the shared dataspace language Swarm - a proof system similar in style to that of UNITY. The paper then uses the proof system to reason about two different solutions to the problem of labelling the connection equal-intensity regions of the digital image.

A UNITY-STYLE PROGRAMMING LOGIC FOR A SHARED DATASPACE LANGUAGE

H. Conrad Cunningham
Gruia-Catalin Roman

WUCS-89-05

March 1989

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

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 the shared dataspace language Swarm - a proof system similar in style to that of UNITY. The paper then uses the proof system to reason about two different solutions to the problem of labeling the connected equal-intensity regions of a digital image.

As appeared in IEEE Transactions on Distributed and Parallel Computing 1, No. 3, July 1990, pp. 365-376.

A UNITY-style Programming Logic for a Shared Dataspace Language

H. Conrad Cunningham and Gruia-Catalin Roman

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri, U.S.A.

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 the shared dataspace language Swarm—a proof system similar in style to that of UNITY. The paper then uses the proof system to reason about two different solutions to the problem of labeling the connected equal-intensity regions of a digital image.

1 Introduction

Much of the research on verification of concurrent programs has focused on languages with semantics similar to Dijkstra's Guarded Commands [5] or Hoare's Communicating Sequential Processes [6]. 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 [4], Action Systems [2], and event predicates [7], 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] 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 [9], 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 language called *Swarm* [12]. The design of Swarm was influenced by the previous work on Linda [1], Associons [8], OPS5 [3], and UNITY [4]. 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.[1] 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 language and specified a formal operational model based on a state-transition

[1] For simplicity of presentation, we are ignoring the synchrony relation feature of Swarm. The results presented in this paper have been generalized to handle synchronic groups.

approach [12]. Our research is exploring the implications of the shared dataspace approach and the Swarm language design for algorithm development and programming methodology. 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 the Swarm language. In related efforts, we are investigating methods for implementing shared dataspace languages on hypercube multiprocessors [11], special architectures for shared dataspace languages, and the use of the shared dataspace model as a basis for a new approach to the visualization of the dynamics of program execution [10]. We also plan to study the implications of the Swarm research for verification of security in computing systems.

In this paper, we specify a proof system for Swarm similar in style to that of UNITY and illustrate its use by means of proofs for two different solutions to 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 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. 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 two solutions to the region labeling problem; these programs first appeared in [12].

2 The Swarm Language

Underlying the Swarm language is a state-transition model similar to that of UNITY, but recast into the shared dataspace framework. In the model, the state of a computation is represented by the contents of the *dataspace*, a set of content-addressable entities. The model partitions the dataspace into several subsets, of which the *tuple space*—a finite set of data *tuples*—and the *transaction space*—a finite set of *transactions*— are of significance to this presentation. An element of the dataspace is a pairing of a *type* name with a sequence of *values*. In addition, a transaction has an associated behavior specification.

Although actual 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. An executing transaction examines the dataspace and then, depending upon the results of the examination, can delete tuples (but not transactions) from the dataspace and insert new tuples and transactions into the dataspace. Unless a transaction explicitly reinserts itself into the dataspace, it is deleted as a by-product of its execution. Program execution continues until there are no transactions in the dataspace.

Before explaining the syntax and semantics of programs, we introduce a few frequently used basic constructs: constructors, predicates, and generators.

Constructors. Swarm's ubiquitous constructor notation is used to form a set of entities and apply an operator to the elements of the set. It has the form:

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

The *operator* field is a commutative and associative operator such as $\exists$, $\forall$, $\Leftrightarrow$, Σ, Π, min, and max. The *operands* field is a list of operands (separated by semicolons) compatible with the operator. The *variables* field is a (possibly null) list of bound variables whose scopes are delimited by the brackets. An instance of the constructor corresponds to a set of values for the bound variables such that the *domain* predicate is satisfied. (If the domain is blank, then the constant true is taken for the predicate.) The operator is applied to the set of operands corresponding to all instances of the constructor; if there are no instances, then the result of the constructor is the identity element of the operator, e.g., 0 for Σ, 1 for Π, true for $\forall$, and false for $\exists$.

Predicates. Swarm predicates are first-order logical expressions constructed in the usual manner from other predicates, parentheses, and the logical operators $\wedge$, $\vee$, and $\neg$. (For convenience, a comma can be used in place of $\wedge$). Simple predicates include tests for the usual arithmetic relationships among values. Predicates can also examine the dataspace. For example, the predicate *has_label*(17,λ), where λ is a variable, examines the dataspace for an element of type *has_label* having two components, the first being the constant 17 and the second being an arbitrary value. If such an element exists, the predicate succeeds and the value of the second component of the matched element is bound to the variable λ.

Generators. A special form of the constructor, called a *generator*, is used to form a set of entities. For example, the generator

[P, L : Pixel(P), Pixel(L) : has_label(P,L)]

generates a set consisting a *has_label(P,L)* tuple for all values of P and L that satisfy the predicate *Pixel*. The domain predicate of a generator is not allowed to examine the dataspace.

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

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

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

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

```
program RegionLabel1 (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), P ≠ Q,
        [∃ x,y,a,b : P=(x,y), Q=(a,b) :
          a−1 ≤ x ≤ a+1, b−1 ≤ y ≤ b+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 : Pixel(P) :
      Label1(P) ≡
          ρ,λ1,λ2 :
            has_label(P,λ1)†, has_label(ρ,λ2),
            R_neighbors(P,ρ), λ1 > λ2
              → has_label(P,λ2)
        ‖ true→ Label1(P)
  ]
initialization
  [ P : Pixel(P) :
      has_label(P,P),
      has_intensity(P,Intensity(P)),
      Label1(P)
  ]
end
```

Figure 1: Nonterminating Region Labeling

Transaction types. The transaction types section declares the types of transactions that can exist in the transaction space. Each transaction type declaration defines a set of transaction *instances* that can be executed, inserted, or examined by a program. The program in Figure 1 declares a transaction type *Label1(P)*.

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

$$\begin{array}{ll} & variable\_list_1 : query_1 \quad \rightarrow action_1 \\ \| & \cdots \\ \| & variable\_list_n : query_n \quad \rightarrow action_n \end{array}$$

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

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

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

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

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

3 A Programming Logic

In this section we formalize our view of program execution and present an assertional programming logic. A more complete presentation of the formal model sketched below can be found in [12]. For simplicity in presentation, we are not considering 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.

A Swarm dataspace can be partitioned into a finite tuple space and a finite transaction space. For dataspace d, let $\mathbf{Tr}.d$ denote the transaction space of d. The **transaction types** section of a program defines the set of all possible transaction instances **TRS**.

A Swarm program can be modeled 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 $\mathbf{e}$ is an ordered pair consisting of a program dataspace $\mathbf{Ds}.\mathbf{e}_i$ and a set $\mathbf{Sg}.\mathbf{e}_i$ containing a single transaction chosen from $\mathbf{Tr}.\mathbf{Ds}.\mathbf{e}_i$. (If $\mathbf{Tr}.\mathbf{Ds}.\mathbf{e}_i = \emptyset$, then $\mathbf{Sg}.\mathbf{e}_i = \emptyset$.)

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'. (For more detail, see [12].)

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

- $\mathbf{Ds}.\mathbf{e}_0$ is a valid initial dataspace of the program.
- For $i \geq 0$, if $\mathbf{Tr}.\mathbf{Ds}.\mathbf{e}_i \neq \emptyset$
  then $\mathbf{step}(\mathbf{Ds}.\mathbf{e}_i, \mathbf{Sg}.\mathbf{e}_i, \mathbf{Ds}.\mathbf{e}_{i+1})$;
  otherwise $\mathbf{Ds}.\mathbf{e}_i = \mathbf{Ds}.\mathbf{e}_{i+1}$.
- $\mathbf{e}$ is *fair*, i.e.,

$$(\forall i, t : 0 \leq i \wedge t \in \mathbf{Tr}.\mathbf{Ds}.\mathbf{e}_i : (\exists j : j \geq i : \mathbf{Sg}.\mathbf{e}_j = \{t\} \wedge (\forall k : i \leq k \leq j : t \in \mathbf{Tr}.\mathbf{Ds}.\mathbf{e}_k)))$$

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; 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 [4]. 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 $(p \wedge \neg q)(e_i)$ 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."

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:

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

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:

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

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:

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

From this we can deduce:

$$(\forall e, i : e \in \mathbf{Exec} \wedge 0 \leq i :$$
$$p(e_i) \Rightarrow (\forall j : j \geq i : (p \vee \neg q)(e_j)) \vee$$
$$(\exists k : i \leq k : q(e_k) \wedge$$
$$(\forall j : i \leq j \leq k : (p \wedge \neg q)(e_j))))$$

In other words, either $p \wedge \neg q$ continues to hold indefinitely or 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:

$$\textbf{stable } p \equiv p \textbf{ unless false}$$
$$\textbf{invariant } p \equiv (\mathit{INIT} \Rightarrow p) \wedge (\textbf{stable } p)$$

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:

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

We also define **constant** properties such that:

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

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:

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

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:

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

The Swarm definition of ensures is a generalization of UNITY's definition. To see this, note if $(\forall t : t \in \mathbf{TRS} : [t])$ is assumed to be invariant, the above 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:

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

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

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

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 post-conditions differently—often in terms of other stable properties. However, unlike UNITY programs, Swarm programs can *terminate*; a termination predicate $TERM$ can be defined as follows:

$$TERM \quad \equiv \quad (\forall t : t \in \mathbf{TRS} : \neg[t])$$

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 Chapter 3 of [4] can be proved for Swarm as well. We use the Swarm analogues of various UNITY theorems in the proofs in the next section.

4 Example Proofs

This section applies the programming logic given in the preceding section to the verification of two Swarm programs. The programs are two different solutions to a region labeling problem.

4.1 Setting Up the Problem

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 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 coordinate of a pixel in that region.

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 coordinate, 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 coordinate in its region. More formally, we define $POST$ as follows:

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

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

1. the characteristics of the problem, as described above, 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 two different programs to solve the region labeling problem. The programs differ on how the required progress is achieved.

The first program, *RegionLabel1*, uses a static set of transactions. Each transaction is "anchored" to a pixel in the image; the transactions are re-created upon their execution. Each transaction "pulls" a smaller label from a neighboring pixel to its own pixel. Eventually a region's winning label propagates throughout the region.

The second program, *RegionLabel2*, uses a dynamic set of transactions. Each transaction "carries" a pixel's label to a neighbor; the transaction does not re-create itself. New transactions are created whenever a label changes. As before, 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 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 can have only one intensity attribute; this value is constant and equal to $Intensity(p)$ throughout the computation. In terms of the Swarm programming logic, the program must satisfy the Intensity Invariant defined below.

Property 3 (Intensity Invariant)

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

(Above the "#" operator denotes the operation of counting the number of elements satisfying the quantification predicate.) The first conjunct of this invariant guarantees the uniqueness of the intensity attribute. The second conjunct guarantees the constancy of the attribute.

Each pixel can have only one label. This label is the coordinate of some pixel within the same region. We also require a pixel's label to be no larger than the pixel's own coordinate. These three requirements are captured in the Labeling Invariant stated below.

Property 4 (Labeling Invariant)

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

Both solutions to the region labeling problem 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:

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

For convenience we define the function XS (read "excess") on regions such that $XS(i)$ 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,

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

where the "Σ" and"$-$" operators denote componentwise summation and subtraction of the coordinates.

Using XS, the predicate *POST* can be restated

$$POST \equiv (\forall i : 1 \leq i \leq Nregions : XS(i) = \mathbf{0})$$

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

Given the definition of XS, we can derive a region-oriented corollary of the Labeling Invariant, the Region Labeling Invariant.

Property 5 (Region Labeling Invariant)

invariant

$$0 \leq XS(i) \leq (\Sigma\, p : p \in R(i) : p - w(i))$$

Proof (Region Labeling Invariant): This assertion follows from the Labeling Invariant and the definition of XS. ∎

We consider a region labeling program which uses the *has_intensity* and *has_label* tuple types to be correct if it satisfies the Labeling Completion, Labeling Stability, Intensity Invariant, and Labeling Invariant properties. For each of the two programs given in the following subsections we prove these properties. The proofs of these properties require us to define and prove additional properties.

4.3 A Static Solution

The Swarm program *RegionLabel*1 is given in Figure 1. This program defines a transaction type *Label*1 as follows:

```
[ P : Pixel(P) :
   Label1(P) ≡
        ρ, λ1, λ2 :
          has_label(P,λ1)†, has_label(ρ, λ2),
          R_neighbors(P,ρ), λ1 > λ2
                → has_label(P,λ2)
     ||
        true
                → Label1(P)
]
```

The *has_intensity* and *has_label* tuple types and the *Pixel* and *R_neighbors* predicates are defined in Figure 1. The predicate *Pixel*(P) is *true* for every pixel P in the image and *false* otherwise. The predicate *R_neighbors*(x, y) is *true* if and only if pixel x pixel y are neighbors in the image (as described previously) and have *equal intensity attributes.*

The program is initialized as follows:

```
[ P : Pixel(P) :
      has_label(P,P),
      has_intensity(P,Intensity(P)),
      Label1(P)
]
```

Each pixel is labeled with its own coordinate. A *Label*1(P) transaction is created for each pixel P in the image.

As noted earlier, verifying the correctness of *RegionLabel*1 requires the proof of the Intensity Invariant, Labeling Invariant, Labeling Stability, and Labeling Completion properties. In proving these, we introduce and prove other properties.

Proof (Intensity Invariant): Prove

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

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

Since the Regional Labeling Invariant is a corollary of the Labeling Invariant (as proved earlier), the proof for the Labeling Invariant below also establishes the Region Labeling Invariant for *RegionLabel*1.

Proof (Labeling Invariant): For convenience, we rewrite the invariant assertion as three conjuncts:

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

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 and third conjuncts are *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. No transaction deletes a *has_label*$(p, *)$ tuple without inserting a *has_label*$(p, *)$ tuple, and vice versa. Thus the number of *has_label*$(p, *)$ tuples remains constant.

(2) Consider the second conjunct of the invariant. Any transaction which changes pixel p's label sets it to the value of a neighbor's label in the same region.

(3) Consider the third conjunct of the invariant. Any transaction which changes a pixel's label sets the label to a smaller value. Suppose a pixel's label is decreased below the region's $w(i)$. This introduces a contradiction because of part 2 and the definition of $w(i)$ as the minimum pixel coordinates in the region. Therefore, all three conjuncts are stable. ∎

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 $p \in R(i) \;\wedge\; \mathit{has\_label}(p, w(i))$

Proof: 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 (Labeling Stability): We must prove the property:

stable $POST$

The stability of the assertion $XS(i) = 0$, for any region i, follows from the Pixel Label Stability property for each pixel in the region, the unless Conjunction Theorem from [4], and the definition of XS. Applying the Conjunction Theorem again for the regions in the image, we prove the stability of $POST$. ∎

The remaining proof obligation for $RegionLabel1$ 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 as a whole.

The following definition is convenient for expression of the properties in this proof:

$$BOUNDARY(i,p,q) = \\ p \in R(i) \wedge q \in R(i) \wedge neighbors(p,q) \wedge \\ (\exists l, m : l > m : \\ has\_label(p,l) \wedge has\_label(q,m))$$

To prove Labeling Completion, we first seek to prove a Regional Progress property, $XS(i) \geq 0 \longmapsto XS(i) = 0$. We can prove this by induction using the simpler property $0 < XS(i) = k \longmapsto XS(i) < k$. This, in turn, we can prove using the Incremental Labeling property defined below. The Incremental Labeling property guarantees that, whenever $BOUNDARY(i,p,q) \wedge XS(i) > 0$, there is a transaction in the dataspace which will decrease $XS(i)$.

Property 7 (Incremental Labeling)

$$BOUNDARY(i,p,q) \wedge 0 < XS(i) = k \\ \text{ensures } XS(i) < k$$

From the definition of the ensures property in the previous section, we must:

1. prove LHS unless RHS (where LHS and RHS denote the left- and right-hand-sides of the ensures relation);
2. prove, when $LHS \wedge \neg RHS$, there exists a transaction in the transaction space which will, when executed, establish the RHS (if it hasn't already been established).

We prove these parts separately.

Proof (Incremental Labeling—unless part): All transactions either leave the labels unchanged or decrease one label by some amount. Hence, the unless property

$$BOUNDARY(i,p,q) \wedge 0 < XS(i) = k \\ \text{unless } XS(i) < k$$

holds for the program. ∎

The proof of the existential part of the ensures needs an additional property, the Static Transaction Space invariant. The Static Transaction Space invariant guarantees there is always a $Label1$ transaction "anchored" on every pixel in the image.

Property 8 (Static Transaction Space)

invariant $Label1(p)$

Proof: Initially the property holds. Every transaction always re-creates itself and never creates any other transactions. ∎

Given the Static Transaction Space invariant, we can now prove the existential part of the Incremental Labeling property.

Proof (Incremental Labeling—exists part): We must show there is a $t \in$ **TRS** such that

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

where PRE is

$$BOUNDARY(i,p,q) \wedge 0 < XS(i) = k.$$

By the Static Transaction Space invariant, a $Label1(p)$ transaction is in the transaction space. Execution of this transaction establishes $XS(i) < k$. ∎

Thus the Incremental Labeling property holds for $RegionLabel1$. 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 9 (Regional Progress)

$$XS(i) \geq 0 \quad \longmapsto \quad XS(i) = 0$$

The proof of the Regional Progress property needs an additional property, the Boundary Invariant. The Boundary Invariant guarantees that, when $XS(i) > 0$, there exist neighbor pixels in the region which have unequal labels.

Property 10 (Boundary Invariant)

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

Proof: For single pixel regions $XS(i) = 0$ holds invariantly; hence the Boundary Invariant holds.

Consider multi-pixel regions. Initially $XS(i) > 0$. Because of the Pixel Label Stability property, the

invariance of $has\_label(w(i), w(i))$ is clear. When $XS(i) > 0$, because of the definition of XS 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, which have unequal labels. ∎

Proof (Regional Progress): Since $XS(i) = 0 \longmapsto XS(i) = 0$ is obvious, only $XS(i) > 0 \longmapsto XS(i) = 0$ remains to be proven.

From the Incremental Labeling progress property we know

$$BOUNDARY(i, p, q) \wedge 0 < XS(i) = k \text{ ensures } XS(i) < k.$$

Because of the Boundary Invariant, we also know

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

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

$$0 < XS(i) = k \longmapsto XS(i) < k$$

which can be rewritten as

$$XS(i) > 0 \wedge XS(i) = k \longmapsto (XS(i) > 0 \wedge XS(i) < k) \vee XS(i) = 0.$$

$XS(i)$ is a well-founded metric. Thus, using the induction principle for leads-to [4], we conclude the Regional Progress property. ∎

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

Proof (Labeling Completion): Prove the assertion $INIT \longmapsto POST$. Clearly,

$$INIT \Rightarrow (\forall i :: XS(i) \geq 0).$$

Hence, for each region i,

$$INIT \text{ ensures } XS(i) \geq 0.$$

From the Regional Progress property,

$$XS(i) \geq 0 \longmapsto XS(i) = 0.$$

The Labeling Stability property, the Completion Theorem for leads-to [4], and the transitivity of leads-to allow us to conclude $INIT \longmapsto POST$. ∎

The proof of program *RegionLabel*1 is now complete. We have shown the program satisfies the required properties.

4.4 A Dynamic Solution

This section states and verifies the correctness of a dynamic solution to the region labeling problem. Unlike *RegionLabel*1, the contents of the transaction space vary during the computation. The progress proof must take this into account. This program also terminates; thus we can illustrate a method for proving termination of Swarm programs.

The Swarm program *RegionLabel*2 is like *RegionLabel*1 except that transaction type *Label*2 replaces *Label*1. Transaction type *Label*2 is defined as follows:

```
[ P, L : Pixel(P), Pixel(L) :
  Label2(P,L) ≡
    [|| δ : P = L, neighbors(P,δ) :
      ι : has_intensity(P,ι), has_intensity(δ,ι)
          → Label2(δ,P)
    ]
    ||
      λ : has_label(P,λ)†, λ > L
          → has_label(P,L)
    ||
    [|| δ : δ ≠ L, neighbors(P,δ) :
      λ,ι : has_label(P,λ), λ > L,
        has_intensity(P,ι), has_intensity(δ,ι)
          → Label2(δ,L)
    ]
]
```

Each $Label2(P, L)$ transaction consists of three groups of subtransactions. In the first and third groups we use a new Swarm feature, the subtransaction generator. These groups include a subtransaction for each $\delta \neq L$ such that $neighbors(P, \delta)$.

The *RegionLabel*2 program is initialized as follows:

```
[ P : Pixel(P) :
    has_label(P,P),
    has_intensity(P,Intensity(P)),
    Label2(P,P)
]
```

A $Label2(P, P)$ transaction is created initially for each pixel P.

Verifying the correctness of *RegionLabel*2 requires the proof of the same four properties as *RegionLabel*1: the Intensity Invariant, Labeling Invariant, Labeling Stability, and Labeling Completion properties.

Proof (Intensity Invariant): Similar to the Intensity Invariant proof for *RegionLabel*1. ∎

Because *Label*2 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*2

transactions, the Transaction Label Invariant shown below.

Property 11 (Transaction Label Invariant)

$$\textbf{invariant} \quad p \in R(i) \wedge \mathit{Label2}(p,l) \Rightarrow l \in R(i) \wedge w(i) \leq l$$

Proof: The only transactions existing initially are the $\mathit{Label2}(p,p)$ 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 $\mathit{Label2}(p,l)$ can only create transactions of the form $\mathit{Label2}(q,l)$ where q is a neighbor of p. Thus the invariant is preserved. ∎

Proof (Labeling Invariant): The proof is similar to the Labeling Invariant proof for $\mathit{RegionLabel1}$ except the Transaction Label Invariant is used to prove the stability of the second and third parts of the invariant. ∎

Proof (Labeling Stability): Similar to the Labeling Stability proof for $\mathit{RegionLabel1}$. ∎

So far the proofs of the properties have been almost identical to the proofs for $\mathit{RegionLabel1}$. The remaining proof obligation is the Labeling Completion progress property. We follow the same methodology as with $\mathit{RegionLabel1}$.

To prove Labeling Completion, we first seek to prove $XS(i) \geq \mathbf{0} \longmapsto XS(i) = \mathbf{0}$. 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 $\mathit{Label2}(p,p)$ transactions initiate the label propagation from each pixel p. However, once transaction $\mathit{Label2}(w(i),w(i))$ has executed for each region, there are transactions in the transaction space that decrease $XS(i)$. Moreover, $\mathit{Label}(w(i),w(i))$ is never regenerated by the computation (because of the $\delta \neq P$ restriction in the transaction definition). Thus we seek to prove the property $\neg\mathit{Label2}(w(i),w(i)) \wedge XS(i) \geq \mathbf{0} \longmapsto XS(i) = \mathbf{0}$. We can prove this property using the D-Incremental Labeling ensures property defined later.

We evoke the following metaphor to set up the proof for the D-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 $\mathit{Label2}$ transactions labeling pixels with $w(i)$.

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

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

The D-Incremental Labeling ensures property guarantees that, when $XS(i) > \mathbf{0}$ under appropriate conditions, there is a transaction in the dataspace which will decrease $XS(i)$.

Property 12 (D-Incremental Labeling)

$$\begin{aligned}&\neg\mathit{Label2}(w(i),w(i)) \wedge \mathit{BOUNDARY}(i,p,q) \wedge \\ &\quad \mathbf{0} < XS(i) = k \\ &\qquad \textbf{ensures } XS(i) < k\end{aligned}$$

As with $\mathit{RegionLabel1}$, we divide the proof into an unless-part and an exists-part.

Proof (D-Incremental Labeling—unless): All transactions either leave the labels unchanged or decrease one label by some amount. No transaction creates a $\mathit{Label}(w(i),w(i))$ transaction. Hence, LHS unless RHS holds for the program. ∎

To prove the existential part of the D-Incremental Labeling property, we need to show there exists a transaction in the transaction space which, when executed, will decrease $XS(i)$. We evoke the wavefront metaphor described above. The Transaction Wavefront invariant guarantees the existence of $\mathit{Label2}(*,w(i))$ transactions along the wavefront boundary.

Property 13 (Transaction Wavefront)

$$\begin{aligned}&\textbf{invariant} \\ &\quad \neg\mathit{Label2}(w(i),w(i)) \wedge \mathit{BOUNDARY}(i,p,q) \\ &\qquad \Rightarrow \mathit{Label2}(q,w(i))\end{aligned}$$

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 14 (Startup)

$$\begin{aligned}&\mathit{Label2}(w(i),w(i)) \text{ unless} \\ &\quad (\mathit{BOUNDARY}(i,p,q) \Rightarrow \mathit{Label2}(q,w(i)))\end{aligned}$$

Proof: 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). $\mathit{Label2}(w(i),w(i))$ creates $\mathit{Label2}(q,w(i))$, thus establishing the RHS of the unless assertion. All other transactions leave $\mathit{Label2}(w(i),w(i))$ *true*. ∎

In the proof above we needed to know that when $\mathit{Label2}(w(i),w(i))$ transactions exist the wavefront has not been started; this is the Winning Label Initiation property.

Property 15 (Winning Label Initiation)

$$\textbf{invariant } Label2(w(i), w(i)) \wedge p \in R(i) \wedge p \neq w(i) \Rightarrow \neg has\_label(p, w(i)) \wedge \neg Label2(p, w(i))$$

Proof: 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 16 (Boundary Stability)

$$\textbf{stable } BOUNDARY(i, p, q) \Rightarrow Label2(q, w(i))$$

Proof: 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, $BOUNDARY(i, p, q)$ 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 $Label2(p, w(i))$. This transaction creates $Label2(q, w(i))$, 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 $Label2(q, w(i))$. This transaction also changes the label of q to $w(i)$, thus falsifying the predicate $BOUNDARY(i, p, q)$. ∎

Proof (Transaction Wavefront): We must show

$$\neg Label2(w(i), w(i)) \wedge BOUNDARY(i, p, q) \Rightarrow Label2(q, w(i))$$

is invariant. The property holds initially because $INIT \Rightarrow Label2(w(i), w(i))$. From the Startup property, we know

$$Label2(w(i), w(i)) \textbf{ unless } (BOUNDARY(i, p, q) \Rightarrow Label2(q, w(i))).$$

From the Boundary Stability property we know

$$(BOUNDARY(i, p, q) \Rightarrow Label2(q, w(i))) \textbf{ unless false.}$$

Using the Cancellation Theorem for unless [4], we conclude the following:

$$Label2(w(i), w(i)) \vee (BOUNDARY(i, p, q) \Rightarrow Label2(q, w(i))) \textbf{ unless false}$$

Thus the invariant is preserved. ∎

Proof (D-Incremental Labeling—exists): We must show there is a $t \in \textbf{TRS}$ such that

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

where PRE is

$$\neg Label2(w(i), w(i)) \wedge BOUNDARY(i, p, q) \wedge 0 < XS(i) = k.$$

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

Thus the D-Incremental Labeling property holds for $RegionLabel2$. As with $RegionLabel1$, we now use this property to prove labeling completion for each region in the image. More formally, we prove the D-Regional Progress property defined below.

Property 17 (D-Regional Progress)

$$\neg Label2(w(i), w(i)) \wedge XS(i) \geq 0 \longmapsto XS(i) = 0$$

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

Property 18 (D-Boundary Invariant)

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

Proof: Similar to the proof of the Boundary Invariant for $RegionLabel1$. ∎

Proof (D-Regional Progress): The progress property $XS(i) = 0 \longmapsto XS(i) = 0$ is obvious, thus only

$$\neg Label2(w(i), w(i)) \wedge XS(i) > 0 \longmapsto XS(i) = 0$$

remains to be proven.

From the D-Incremental Labeling progress property we know

$$\neg Label2(w(i), w(i)) \wedge BOUNDARY(i, p, q) \wedge 0 < XS(i) = k \textbf{ ensures } XS(i) < k.$$

Because of the D-Boundary Invariant we also know

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

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

$$\neg Label2(w(i), w(i)) \wedge 0 < XS(i) = k \longmapsto XS(i) < k.$$

Since $\neg Label2(w(i), w(i))$ is stable, we can rewrite the assertion above as

$$\neg Label2(w(i), w(i)) \wedge XS(i) > 0 \wedge XS(i) = k \longmapsto (\neg Label2(w(i), w(i)) \wedge XS(i) > 0 \wedge XS(i) < k) \vee XS(i) = 0.$$

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

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

Proof (Labeling Completion): Prove the assertion $INIT \longmapsto POST$. Clearly,

$$INIT \Rightarrow (\forall i :: XS(i) \geq 0 \wedge Label2(w(i), w(i))).$$

Hence, for each region i,

$$INIT \text{ ensures } XS(i) \geq 0 \wedge Label2(w(i), w(i)).$$

From the transaction definition, it is easy to see

$$Label2(w(i), w(i)) \text{ ensures } \neg Label2(w(i), w(i)).$$

Hence,

$$Label2(w(i), w(i)) \wedge XS(i) \geq 0 \text{ ensures } XS(i) = 0 \vee (\neg Label2(w(i), w(i)) \wedge XS(i) > 0).$$

From the D-Regional Progress property,

$$\neg Label2(w(i), w(i)) \wedge XS(i) \geq 0 \longmapsto XS(i) = 0.$$

The Cancellation Theorem for leads-to [4] allows us to deduce

$$Label2(w(i), w(i)) \wedge XS(i) \geq 0 \longmapsto XS(i) = 0.$$

The Labeling Stability property, the Completion Theorem for leads-to [4], and the transitivity of leads-to allow us to conclude $INIT \longmapsto POST$. ∎

Above we have shown program *RegionLabel2* satisfies the four criteria for correctness of region labeling programs. However, we can also prove this program terminates. We define the termination predicate $TERM$ as follows:

$$TERM \equiv (\forall\, p, l :: \neg Label2(p, l))$$

Since we have already established the Labeling Completion property, we need only prove

$$POST \longmapsto TERM.$$

Again we can prove this leads-to property using an ensures property, the Transaction Flushing property below.

Property 19 (Transaction Flushing)

$$POST \wedge Label2(p, l) \wedge 0 < (\#\, q, m :: Label2(q, m)) = k \text{ ensures } (\#\, q, m :: Label2(q, m)) < k$$

Proof: By the Transaction Label Invariant, we know:

$$q \in R(i) \wedge Label2(q, m) \Rightarrow m \in R(i) \wedge w(i) \leq m$$

The $POST$ predicate means all $Label2$ transactions will fail. Thus the RHS of the ensures property is established. ∎

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

$$POST \wedge 0 < (\#\, q, m :: Label2(q, m)) = k \longmapsto (\#\, q, m :: Label2(q, m)) < k.$$

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

5 Conclusions

In this paper, we have specified a proof system for a large subset of the shared dataspace language Swarm. To our knowledge, this is the first such proof system for a shared dataspace language. To illustrate the logic, we used it to verify the correctness of two programs 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. 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 language design, formal model, and programming logic provide a foundation for several other promising research efforts. We believe visualization can play a key role in exploration of concurrent computation. A companion paper [10] outlines 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. Building on our formal studies of Swarm, we are also investigating implementation techniques [11] and special architectures for execution of shared dataspace programs. We also plan to study the impact of the Swarm research on verification of security in computing systems and on expert systems technologies. 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, and Liz Hanks for their suggestions concerning this article.

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

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

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

[7] 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.

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

[9] 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.

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

[11] G.-C. Roman and K. C. Cox. Implementing a shared dataspace language on a message-based multiprocessor. In *Proceedings of the 5th International Workshop on Software Specification and Design.* IEEE, May 1989.

[12] 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 270–9. IEEE, June 1989.