Report Number: WUCS-89-28

1989-06-01

Toward Formal Verification of Rule-Based Systems: A Shared Dataspace Perspective

H. Conrad Cunningham and Gruia-Catalin Roman

Rule-based programs used in mission- and safety-critical applications need to be shown to be free of hazards. This paper discusses formal proof-techniques which promise to assist designers in this task. In this paper we show that the shared dataspace language Swarm has many key features in common with rule-based languages. We outline an assertional programming logic for Swarm programs and use the logic to reason about the correctness of a simple program. This logic is a suitable foundation for the development of techniques specific to present and future rule-based languages.

TOWARD FORMAL VERFICATION OF RULE-BASED SYSTEMS: A SHARED DATASPACE PERSPECTIVE

H. Conrad Cunningham
Gruia-Catalin Roman

WUCS-89-28

June 1989

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

ABSTRACT

Rule-based programs used in mission- and safety-critical applications need to be shown to be free of hazards. This paper discusses formal proof techniques which promise to assist designers in this task. In this paper we show that the shared dataspace language Swarm has many key features in common with rule-based languages. We outline an assertional programming logic for Swarm programs and use the logic to reason about the correctness of a simple program. This logic is a suitable foundation for the development of techniques specific to present and future rule-based languages.

Toward Formal Verification of Rule-based Systems: A Shared Dataspace Perspective

H. Conrad Cunningham and Gruia-Catalin Roman

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

Abstract

Rule-based programs used in mission- and safety-critical applications need to be shown to be free of hazards. This paper discusses formal proof techniques which promise to assist designers in this task. In this paper we show that the shared dataspace language Swarm has many key features in common with rule-based languages. We outline an assertional programming logic for Swarm programs and use the logic to reason about the correctness of a simple program. This logic is a suitable foundation for the development of proof techniques specific to present and future rule-based languages.

1 Introduction

During the past decade, rule-based languages [1, 5] have become increasingly popular both in the artificial intelligence research community and in the industrial community at large. Nevertheless, in the area of mission- and safety-critical software, rule-based systems have made few inroads. The general expectation is that rule-based systems will become a useful decision support tool for the human operator. Many safety-critical systems, however, need to operate with little (e.g., target tracking) or no (e.g., satellites, missiles) human interaction. Because these kinds of applications require stringent software validation, the lack of formal verification methods for rule-based languages makes such languages unacceptable to the safety minded developer.

Proof methods, however, represent an important and persistent concern for the programming language community at large. Significant progress has been made with respect to the verification of both sequential and concurrent programs. A large body of scientific knowledge has been accumulated. Its systematic application to the study and development of critical algorithms is widespread and, with the assistance of automatic theorem provers, the verification of small industrial-grade programs have been attempted. At this point the reader may wonder why the same formal verification methods had little impact on rule-based systems, despite the fact that both concurrent and rule-based programming use a common underlying operational model: interleaving of atomic actions.

The answer, we believe, lies with the distinct nature of the basic paradigms used by the two research communities. Concurrent programming is currently dominated by the shared-variable and message-passing paradigms. (In our opinion, remote procedure call and object-oriented programming address mostly packaging issues and do not represent fundamental paradigm shifts.) Both paradigms use name-based accessing of shared objects, i.e., variables or channels. In contrast, rule-based programming involves content-based addressing of entities stored in a database of facts. The key to easy transferability of formal verification results from concurrent programming to rule-based languages seems to rest with the ability to accommodate content-based addressing.

An equally significant, but subtle, distinction between rule-based and concurrent programming is the emphasis given to computation versus that given to communication. In rule-based programming, one rarely views the database of facts as a medium for communication among the rules. On the other hand, interprocess communication is always the central issue in the design of concurrent algorithms. We propose to unify these two perspectives on computing by reducing both computation and communication to the notion of atomic transformation of a content-addressable medium (called a shared dataspace) which includes the computing/communicating agents.

In several earlier papers we have expounded upon a model called the Shared Dataspace paradigm [6]. This model represents the underlying semantics of a simple language called Swarm [7]. The original motivation for the model was the simplification of the concurrent programming task (perhaps at the expense of more complex language implementations). The work

on the Linda [2] language and the growing interest in parallel computation among artificial intelligence researchers has encouraged our work.

Like many rule-based languages, Swarm represents data as tuples. However, unlike most rule-based languages, Swarm uses a dynamically varying set of production rules. In Swarm, these rules are called transactions. The set of data tuples existing at any point in a computation is called a tuple space; the set of transactions, a transaction space. Both the tuple space and the transaction space are subsets of the dataspace.

Each transaction has a tuple representation in the dataspace and a separate type-specific behavior definition. A transaction can query the entire dataspace, delete data tuples, and insert both data tuples and new transactions. Transactions are selected for execution in a fair manner and are deleted implicitly as part of their execution. To simulate the static set of productions of a rule-based program, one must initialize the transaction space to contain a transaction for each production rule and must define the behavior of each transaction to reinsert itself after each execution.

Recently we completed an assertional programming logic for Swarm [4]. The Swarm logic is similar to the programming logic for the UNITY [3] notation proposed by Chandy and Misra, but has been generalized to accommodate content-based accessing and the dynamic set of transactions. (The UNITY model assumes a fixed, finite set of conditional multiple-assignment statements which modify a fixed, finite set of variables.) In light of this new development, rule-based programs which map directly to Swarm programs can be subjected to formal verification. In this paper we illustrate the proof strategy for a simple region labeling program.

Because of the generality of Swarm's underlying formal model, we believe it feasible to customize the Swarm programming logic to the specifics of individual rule-based languages. Even so, some may argue that formal program verification is not a practical technique for rule-based programs—that definition of correctness criteria in terms of predicate calculus may not always be feasible for artificial intelligence programs and that the correctness proofs for large, complex programs will be unwieldy. We choose not to engage ourselves in this controversy here, except to observe that rule-based programs used in safety-critical applications will need to be shown to be free of hazards. The proof technique illustrated in this paper promises to assist designers with this task.

The remainder of this paper gives an overview of the Swarm language, introduces the proof rules, and illustrates the proof strategies on a very simple program. Features of language and logic not used in the example are ignored. More complete information is given in [7].

2 The Swarm Language

In this section we introduce the Swarm language by means of a simple example. Consider the task of assigning unique labels to connected, equal-intensity regions of some digital image. Each point in the image, called a pixel, has three attributes: a unique coordinate on a rectangular grid, an intensity value, and a label. When the labeling is completed, all pixels belonging to the same region must have the same label—the region label. For the sake of brevity, we assume that initially the label attribute equals the pixel's coordinate. We start with a generic rule-based solution and gradually reformulate it as a Swarm program.

The information associated with each pixel can be stated as two facts:

1. each pixel P has intensity I (constant and specific to P),
2. each pixel P has label L (initially equal to P).

The exact representation of these facts depends upon the specific notation system being used. In Swarm, for instance, the two types of facts are treated as data tuples stored in the dataspace and having the forms

$$has\_intensity(P, I) \text{ and } has\_label(P, L)$$

where *has_intensity* and *has_label* are tuple type names.

We turn next to the labeling process. If we can assume some total ordering over the domain of coordinates, e.g., lexicographical order, one single rule suffices:

IF
  there is a pixel P which has a (eight-connected) neighbor Q of equal intensity and the label of Q is smaller than the label of P

THEN
  replace the label of P with the label of Q.

Successive applications of this rule result in the propagation of the smallest label present in some region to all pixels belonging to that region. The same result, however, could be obtained by employing multiple instances of this rule, perhaps one for each pixel P in the image. We find the latter solution more attractive because it makes explicit the opportunities for concurrent execution. To accommodate the notion that each rule is specific to a particular pixel P, we can modify the above rule slightly to get:

IF

the pixel P has an (eight-connected) neighbor Q of equal intensity and the label of Q is smaller than the label of P

THEN

replace the label of P with the label of Q.

In Swarm, the rule above can be represented as a transaction type called *Label* parameterized by the pixel coordinate P. The behavior exhibited by transactions of this type is a direct encoding of the rule above; the query and the action are separated by the symbol $\rightarrow$.

$$
\begin{array}{l}
Label(P) \equiv \\
\quad \rho, \lambda 1, \lambda 2 : \\
\quad\quad has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2), \\
\quad\quad R\_neighbors(P, \rho), \lambda 1 > \lambda 2 \\
\quad\quad\quad \rightarrow has\_label(P, \lambda 2)
\end{array}
$$

The identifiers ρ, $\lambda 1$, and $\lambda 2$ are free variables which are bound by the query, if successful. The action involves a tuple deletion marked by the symbol $\dagger$ (the old label for P) and an insertion (the new label for P). By definition all tuple deletions performed by a transaction are assumed to precede any insertions. The total effect of each transaction execution is an atomic transformation of the dataspace.

At initialization, the dataspace must contain an instance of the transaction type *Label* for each pixel P. Unfortunately, in accordance with the Swarm semantics, each firing of a transaction results in the its deletion from the dataspace unless an explicit reinsertion of the transaction is specified in the behavior definition. To accomplish this, one needs to redefine *Label* to force the reinsertion of the transaction being executed:

$$
\begin{array}{l}
Label(P) \equiv \\
\quad \rho, \lambda 1, \lambda 2 : \\
\quad\quad has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2), \\
\quad\quad R\_neighbors(P, \rho), \lambda 1 > \lambda 2 \\
\quad\quad\quad \rightarrow has\_label(P, \lambda 2) \\
\quad \| \ \text{true} \rightarrow Label(P)
\end{array}
$$

The symbol $\|$ is used to separate subtransactions within a single transaction. Operationally, the subtransactions synchronize after the individual query evaluations and again after completing individual tuple deletions. The effect of the second subtransaction is to keep the rule in the dataspace forever.

Once the labeling is completed none of the transactions will succeed, but they will continue to be selected for execution. We could have added a termination detection mechanism to the program, but it is not essential to demonstrating the verification methodology.

This illustrates one important distinction between rule-based languages such as OPS5 and Swarm. Rule-based languages operate using a sequential match-select-execute cycle with selection rules designed to help the program make rapid progress toward its computational goals. Termination generally takes place when none of the rules can be applied. On the other hand, each Swarm transaction is made up of one or more subtransactions that execute concurrently; the execution mechanism selects transactions for execution in a fair manner, i.e., it eventually selects every transaction in the transaction space. Program termination occurs when the transaction space becomes empty. These distinctions are the result of the dynamic and concurrent nature of Swarm. Many rule-based systems still have a sequential programming view of the world.

A complete region labeling program is shown in a later section as Figure 1. It includes program name specification, formal parameters, auxiliary definitions used to simplify query definitions, tuple and transaction type definitions, and initialization. The purpose of this section was simply to illustrate the relation between rule-based programming and shared dataspace and to provide a simple example which will serve as a vehicle for introducing a formal verification strategy. Swarm's ability to simulate directly any particular rule-based system is not a concern here.

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 [7]. 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 e denote one of these sequences. Each element $\mathbf{e}_i$, $i \geq 0$, of 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 val-

ues of this predicate are derived from the query and action parts of the transaction body. The predicate $\text{step}(d, S, d')$ 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 [7].)

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

- Ds.e_0 is a valid initial dataspace of the program.
- For $i \geq 0$, if $\text{Tr.Ds.e}_i \neq \emptyset$ then $\text{step}(\text{Ds.e}_i, \text{Sg.e}_i, \text{Ds.e}_{i+1})$; otherwise $\text{Ds.e}_i = \text{Ds.e}_{i+1}$.
- e is *fair*, i.e.,

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

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 [3]. 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 $(p \wedge \neg q)(\text{e}_i)$ to denote the evaluation of the predicate $p \wedge \neg q$ with respect to 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:

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

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:

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

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:

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

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:

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

From this we can deduce:

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

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:

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

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:

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

We use the ensures relation to state the most basic progress (liveness) properties of programs. UNITY programs consist of a static set of statements. In contrast, Swarm programs consist of a dynamically varying set of transactions. The dynamism of the Swarm transaction space requires a reformulation of the ensures relation. For a given program in the Swarm subset considered in this paper, the ensures relation is defined:

$$\begin{array}{l} p\ \textbf{ensures}\ q \equiv \\ \quad (p\ \textbf{unless}\ q)\ \wedge \\ \quad (\exists t : t \in \mathbf{TRS} : (p \wedge \neg q \Rightarrow [t]) \wedge \\ \qquad\qquad\qquad \{p \wedge \neg q\}\ t\ \{q\}) \end{array}$$

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:

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

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:

- $$\frac{p\ \textbf{ensures}\ q}{p \longmapsto q}$$
- $$\frac{p \longmapsto q,\ \ q \longmapsto r}{p \longmapsto r} \qquad \text{(transitivity)}$$
- *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:

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

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

UNITY makes extensive use of the fixed-point predicate *FP* which can be derived syntactically from the program text. Since *FP* predicates cannot be defined syntactically in Swarm, verifications of Swarm programs must formulate program postconditions differently—often in terms of other stable properties. However, unlike UNITY programs, Swarm programs can *terminate*; a termination predicate *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 [3] can be proved for Swarm as well. We use the Swarm analogues of various UNITY theorems in the proofs in the next section.

4 Region Labeling

This section applies the programming logic given in Section 3 to the verification of the Swarm program presented informally in Section 2, a program to label the equal-intensity regions of a digital image. In this section we "start from scratch." We formally define the problem and correctness criteria, elaborate the program data structures, and then state a program and argue that it satisfies the correctness criteria.

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:

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

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

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

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

Property 1 (Labeling Completion)

$$INIT \longmapsto POST$$

Property 2 (Labeling Stability)

stable *POST*

The Swarm program *RegionLabel* uses a static set of transactions to label the pixels of an image. 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.

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)

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

(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)

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

The solution to the region labeling problem exploits 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 *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,

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

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

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

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

where 0 denotes the coordinates (0,0).

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 Correctness Proof

In addition to transaction type *Label* and tuple types *has_intensity* and *has_label*, the Swarm program *RegionLabel* (shown in Figure 1) defines predicates *Pixel* and *R_neighbors*. 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 **initialization** section establishes the initial dataspace for execution of the program. Initially, for each pixel P in the image, the dataspace contains a $has\_label(P,P)$ tuple and a $Label(P)$ transaction. A *has_intensity* tuple also associates the proper intensity value with each pixel.

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

```
program RegionLabel(M, N, Lo, Hi, Intensity :
    1 ≤ M, 1 ≤ N, Lo ≤ Hi, Intensity(ρ : Pixel(ρ)),
    [∀ ρ : Pixel(ρ) : Lo ≤ Intensity(ρ) ≤ Hi] )
definitions
  [ P, Q, L ::
    Pixel(P) ≡
      [∃ x, y : P = (x, y) :: 1 ≤ x ≤ N, 1 ≤ y ≤ M];
    R_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];
      [∃ 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) ::
    Label(P) ≡
        ρ, λ1, λ2 :
          has_label(P, λ1)†, has_label(ρ, λ2),
          R_neighbors(P, ρ), λ1 > λ2
            → has_label(P, λ2)
      ‖ true→ Label(P)
  ]
initialization
  [ P : Pixel(P) ::
    has_label(P, P),
    has_intensity(P, Intensity(P)),
    Label(P)
  ]
end
```

Figure 1: Nonterminating Region Labeling

Proof (Intensity Invariant): Prove

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

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

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

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

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 5 (Pixel Label Stability)

$$\text{stable } p \in R(i) \ \wedge \ has\_label(p, w(i))$$

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:

$$\text{stable } POST$$

The stability of the assertion $excess(i) = 0$, for any region i, follows from the Pixel Label Stability property for each pixel in the region, the unless Conjunction Theorem from [3], 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 as a whole.

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

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

The predicate $BOUNDARY(i,p,q)$ is *true* if and only if p and q are neighboring pixels in region i such that p's label is greater than q's.

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

Property 6 (Incremental Labeling)

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

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

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

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 *Label* transaction "anchored" on every pixel in the image.

Property 7 (Static Transaction Space)

invariant $Label(p)$

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 $t \in \mathrm{TRS}$ such that

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

where PRE is

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

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

Thus the Incremental Labeling property holds for *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 8 (Regional Progress)

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

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

Property 9 (Boundary Invariant)

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

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

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

Proof (Regional Progress): Since

$$excess(i) = 0 \longmapsto excess(i) = 0$$

is obvious, only

$$excess(i) > 0 \longmapsto excess(i) = 0$$

remains to be proven.

From the Incremental Labeling progress property we know

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

Because of the Boundary Invariant, we also know

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

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

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

which can be rewritten as

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

$excess(i)$ is a well-founded metric. Thus, using the induction principle for leads-to [3], 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,

$$INIT \quad \Rightarrow \quad (\forall\, i :: excess(i) \geq 0).$$

Hence, for each region i,

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

From the Regional Progress property,

$$excess(i) \geq 0 \quad \longmapsto \quad excess(i) = 0.$$

The Labeling Stability property, the Completion Theorem for leads-to [3], and the transitivity of leads-to allow us to conclude $INIT \longmapsto POST$. ∎

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

5 Conclusions

Mission- and safety-critical software demands the systematic application of program verification methods. In this paper we have shown that the shared dataspace language Swarm has many key features in common with rule-based languages. Consequently, the axiomatic proof logic we have constructed for Swarm is a suitable foundation for the development of proof techniques specific to present and future rule-based languages. This could extend the applicability of rule-based programming to areas where strict validation is required.

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, and Wei Chen for their suggestions concerning the Swarm programming logic. We also thank Rose Fulcomer for her helpful comments on this paper.

References

[1] 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.

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

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

[4] H. C. Cunningham and G.-C. Roman. A UNITY-style programming logic for a shared dataspace language. Technical Report WUCS–89–5, Washington University, Department of Computer Science, St. Louis, Missouri, March 1989.

[5] C. L. Forgy. The OPS83 report. Technical Report CS-84-135, Carnegie-Mellon University, Department of Computer Science, May 1984.

[6] 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.

[7] 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.