WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

---

# THE SHARED DATASPACE APPROACH TO CONCURRENT COMPUTATION:

## The Swarm Programming Model, Notation, and Logic

by

H. Conrad Cunningham

Prepared under the direction of Professor Gruia-Catalin Roman

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

DOCTOR OF SCIENCE

August, 1989

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY

---

ABSTRACT

---

# THE SHARED DATASPACE APPROACH TO CONCURRENT COMPUTATION:

## The Swarm Programming Model, Notation, and Logic

by H. Conrad Cunningham

---

ADVISOR: Professor Gruia-Catalin Roman

---

August, 1989

Saint Louis, Missouri

---

In the shared dataspace approach to concurrent computation, the concurrent components of programs communicate by manipulating a content-addressable data structure called a dataspace. In this dissertation we study this paradigm by means of a simple model called Swarm.

The Swarm model unifies several computational paradigms into a single framework. It integrates the tuple space communication metaphor of Gelernter's Linda with a computational model inspired by Chandy and Misra's UNITY. Swarm reduces both communication and computation to a single mechanism—the atomic execution of statements called transactions. In a manner similar to production rules, Swarm transactions specify a group of tuple insertions and deletions. Unlike UNITY's static set of statements, Swarm supports a dynamically varying set of transactions. The synchrony relation construct adds further dynamism and flexibility to the Swarm programs. It enables a program to couple individual transactions dynamically into groups; each group is executed atomically as if it were a single transaction.

This dissertation informally specifies the Swarm notation, presents a formal operational model, and defines an axiomatic programming logic—the first such logic for a shared dataspace language. Using a sequence of solutions to the problem of labeling the equal-intensity regions of a digital image, the dissertation explores the impact of Swarm upon programming styles. The dissertation also illustrates use of the programming logic by verifying the correctness of three of the region labeling programs.

# TABLE OF CONTENTS

# LIST OF FIGURES

# THE SHARED DATASPACE APPROACH TO CONCURRENT COMPUTATION:

## The Swarm Programming Model, Notation, and Logic

# 1. INTRODUCTION

## 1.1. MOTIVATION

Although important for decades at the system level, concurrency is becoming an increasingly important concept for "applications" programming. Our society's appetite for computing power continues to grow unabatedly. However, the computational speeds of our traditional computing machines are pushing up toward the physical limits. Fortunately, both computer networking strategies and new parallel machine architectures offer promising paths for continuing growth in computational power. Unfortunately, our capabilities for harnessing this potential are still quite primitive. We are bumping up against what Peter Denning has termed the "software barrier." [1]*

New programming approaches are needed to exploit the capabilities of multiple processors operating in parallel. Although multicomputer implementations

*The numbers in brackets in the text indicate references in the Bibliography.

of traditional sequential programming languages can effectively exploit some parallelism in programs, in general, the conceptual frameworks underlying such languages limit the degree of parallelism that can be achieved on currently available multicomputer architectures. To overcome the weaknesses of sequential languages, a wide variety of approaches to concurrent programming have been advanced, e.g., Ada† [2], CSP [3], FP [4], Concurrent Prolog [5], and Actors [6]. Although many of these approaches have strong features and clusters of advocates, there is no broad agreement on which approaches are the "winners" for concurrent programming. No approach has achieved a good balance among conceptual elegance, support for program verification, programming convenience, and various pragmatic issues. Research is still needed to explore new paradigms and concurrent programming techniques.

Landmark developments are rare—their full significance often not appreciated until a significant amount of time has passed. Although Hoare's paper "Communicating Sequential Processes" [3] did receive considerable immediate attention, few would have expected the language fragment he proposed to dominate a whole decade of research in concurrency. CSP has had an enormous impact upon research and development in many areas: computational models [7, 8, 9], program verification [10, 11, 12, 13], programming languages (e.g., Occam [14, 15] and the Ada rendezvous [2]), hardware (e.g., the Transputer [16]), distributed algorithms [17], system design [18, 19], and implementation [20, 21] techniques. Although the CSP approach is still important, the concurrency research community seems to be turning in other directions.

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

What current work will enjoy the same success? We probably will not know until another decade passes. However, two relatively recent developments have the makings of emerging success stories—the kind of power and simplicity that can capture the imagination of both researchers and practitioners. These are Chandy and Misra's UNITY [22, 23, 24] and Gelernter's Linda [25, 26, 27].

**UNITY.** Chandy and Misra argue that the fragmentation of programming approaches along the lines of architectural structure, application area, and programming language features obscures the basic unity of the programming task. With UNITY, their work has not been directed toward the development of a new programming language, per se. Instead, their goal is to unify seemingly disparate areas of programming with a simple theory consisting of a model of computation and an associated proof system.

They build the UNITY computational model upon a traditional imperative foundation, a state-transition system with named (shared) variables to express the state and conditional multiple-assignment statements to express the state transitions. Above this foundation, however, UNITY follows a more radical design: all flow-of-control and communication constructs have been eliminated from the notation. A UNITY program consists of a set of variable declarations, a specification of their initial values, and a finite, static set of multiple-assignment statements. A program execution starts from any state satisfying the initial condition and continues infinitely; in each step an assignment statement is selected nondeterministically, but fairly, and executed atomically. A computation of a UNITY program may reach a fixed point, that is, a state in which further execution of the program does not change the values of the variables. (In [22] Chandy credits the popularity of spreadsheet programs with motivating their study of this simple program structure.)

The UNITY proof system uses an assertional programming logic built on the underlying computational model. By the use of logical assertions about program states, the programming logic frees the programmer from the necessity of reasoning about the execution sequences. Unlike most assertional proof systems which rely on the annotation of the program text with predicates, the UNITY logic seeks to extricate the proof from the text by relying upon proof of program-wide properties such as invariants and progress conditions.

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

**Linda.** David Gelernter argues that "machine-independent methods for parallel programming have been slow to emerge, and that consequently programmers have been forced to accommodate themselves to the machines rather than vice versa." [28] To remedy the situation, he and his colleagues have put forth the Linda communication model as a practical, machine-independent programming vehicle.

Linda's primary contribution to concurrent programming has been the notion that the communicating processes of a concurrent program can be spatially and temporally uncoupled from each other. In the Linda model, the processes communicate by inserting tuples into, deleting tuples from, and examining tuples in a common, content-addressable data structure called a *tuple space*—a process which Gelernter calls generative communication. The processes are uncoupled because

the process that creates the tuple does not "know" when or where or by which process the tuple will be used. Consequently, program changes concerning the use of the particular data do not affect the producer in any way. The model also treats processes as "live tuples" which come into existence by being inserted into the tuple space. The uncoupling and dynamic creation of processes facilitate programs which exhibit high degrees of concurrency.

Although somewhat outside of the mainstream of concurrent programming research, Linda has stimulated considerable interest [27]. Linda has been implemented as an extension to the Fortran and C programming languages by Gelernter's group; other researchers have used other base languages. Implementations have been done for shared memory multicomputers like the Encore Multimax, Sequent Balance and Symmetry, and Alliant FX/8; for distributed memory multicomputers like the Intel iPSC/2 and the S/Net [29]; and for a VAX-based local area network. A Linda Machine [30, 31] which supports the tuple space operations in hardware is under construction. In a commercial venture, Cogent Research is using the Linda communication model as the basis for the operating system in its transputer-based XTM multicomputer system [32]. As noted in [33], Linda has been used for a number of applications, e.g., DNA sequencing.

The Linda research also has shortcomings. Since pragmatic issues have apparently driven much of the recent work on Linda, theoretical issues are given little attention. To our knowledge, no formal model for the language has been published. Proof systems are dismissed as irrelevant. In addition, the one-tuple-at-a-time operations and the inability to handle synchronous computations considerably limit the power and flexibility of the language.

## 1.2. APPROACH

We believe that a model of concurrency which preserves the gains made by UNITY and Linda, while overcoming their limitations, can become a serious alternative to the two dominant concurrent programming paradigms, message-passing and shared variables. Our research has identified a concurrency model that can accomplish this. We call it the *shared dataspace paradigm*. This paradigm, first so named in [34], refers to a class of languages and models in which the primary means for communication among the concurrent components of a program is a common, content-addressable data structure called a *shared dataspace*. Elements of the dataspace may be examined, inserted, or deleted by programs. Gelernter's Linda, Rem's Associons [35, 36], Kimura's Transaction Networks [37], our own Swarm model [38], and production rule languages such as OPS5 [39] all follow the shared dataspace approach.

By choosing the name *Swarm* for our shared dataspace programming model, we evoke the image of a large, rapidly moving aggregation of small, independent agents cooperating to perform a task. In designing Swarm, we attempted to merge the philosophy of UNITY with the methods of Linda. Swarm has a UNITY-like program structure and computational model and Linda-like communication mechanisms. We partition the Swarm dataspace into three subsets: a tuple space (a finite set of data tuples), a transaction space (a finite set of transactions), and a synchrony relation (a symmetric relation on the set of valid transaction instances). We replace UNITY's fixed set of variables with a set of tuples, and the fixed set of conditional assignment statements with a set of transactions. A Swarm transaction denotes an atomic transformation of the dataspace. It is a set of concurrently executed query-action pairs. A query consists of a predicate over all three subsets of

the dataspace; an action consists of a group of deletions and insertions of dataspace elements. Instances of transactions may be created dynamically by an executing program.

A Swarm program begins execution from a specified initial dataspace. On each execution step, a transaction is chosen nondeterministically from the transaction space and executed atomically. This selection is fair in the sense that every transaction in the transaction space at any point in the computation will eventually be chosen. An executing transaction examines the dataspace and then, depending upon the results of the examination, can delete tuples (but not transactions) from the dataspace and insert new tuples and transactions into the dataspace. Unless a transaction explicitly reinserts itself into the dataspace, it is deleted as a by-product of its execution. Program execution continues until there are no transactions in the dataspace.

The synchrony relation feature adds even more dynamism and expressive power to Swarm programs. It is a relation over the set of possible transaction instances. This relation may be examined and modified by programs in the same way as the tuple and transaction spaces are. To accommodate the synchrony relation, we extend the program execution model in the following way: whenever a transaction is chosen for execution, all transactions in the transaction space which are related to the chosen transaction by (the closure of) the synchrony relation are also chosen; all of the transactions that make up this set are executed as if they composed a single transaction.

Although we may occasionally refer to Swarm as a "language," it is not really a programming language. We give little attention to the pragmatic issues necessary for a programming language, e.g., input/output facilities, syntactic sugar, and efficient implementation. Instead, we focus on Swarm as a computational model and

present a simple notation for specifying computations. We have based the Swarm model on a small number of concepts (e.g., the tuple space communication medium, atomic transactions, and the synchrony relation) which we believe are at the core of a large class of shared dataspace languages. By concentrating on the essential nature of the shared dataspace paradigm, our goal was to develop techniques applicable to many shared dataspace languages. For instance, the Swarm programming logic presented in this dissertation can perhaps be reformulated for Linda.

## 1.3. DISSERTATION ORGANIZATION

The starting point for this research is the desire to meld the programming flexibility of a Linda-like communication metaphor with the conceptual elegance of a UNITY-like computational model and programming logic. We want to preserve the strengths of both, while eliminating their weaknesses. We want to go further and explore the new opportunities created by the new concepts that arise out of this combination. As noted in the previous section, the Swarm programming notation is our vehicle for this investigation.

This dissertation lays the conceptual foundation for Swarm. We define the notation and specify a formal operational model and a UNITY-style programming logic. We also explore programming strategies and proof techniques fostered by the programming notation and its logic. Portions of the dissertation are based on earlier papers [38, 40, 41, 42]. In related efforts by others, a simple prototype for Swarm is being built for a hypercube multicomputer [43, 44] and "declarative" techniques for "visualizing" the dynamics of concurrent program execution are being studied in the context of the shared dataspace model [45, 46].

In this chapter we have looked at two models closely related to Swarm, UNITY and Linda. In Chapter 2 we survey some of the other related research in concurrent programming languages, models, and logics.

Chapter 3 informally presents the syntax and semantics of the Swarm notation. It first introduces the notation by analogy to a more familiar procedural language, and then discusses its features in more detail.

The Swarm model brings together a variety of programming styles (e.g., synchronous and asynchronous, static and dynamic) within a unified computational framework. Using the problem of labeling the equal-intensity regions of a digital image as an example, Chapter 4 explores some of the programming styles made possible by the notation. This chapter illustrates the different styles by means of a sequence of variant solutions to the region labeling problem.

To put the notation system on a firm formal foundation, in Chapter 5 we present an operational, state-transition model for Swarm. This model formalizes the linguistic concepts expressed informally in Chapter 3 and lays the foundation for our development of a Swarm programming logic in Chapter 6.

In Chapter 6 we present an assertional programming logic for a subset of Swarm; we do not consider the synchrony relation feature in this chapter. The Swarm logic is similar in style to the logic given for UNITY in [24]. To incorporate Swarm's distinctive features, we must 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.

Chapter 7 applies the programming logic given in Chapter 6 to the verification of two Swarm programs. The programs are two of the solutions to the region labeling

problem given in Chapter 4. This chapter formally defines the problem and the correctness criteria, elaborates the program data structures, and then presents the programs and argues that they satisfy the correctness criteria.

In Chapter 8, we extend the logic given in Chapter 6 to handle synchronic groups. We present a synchronic group rule and generalized definitions for the **unless** and **ensures** relations.

In Chapter 9 we specify a program for a variant of the region labeling problem in which the image extends infinitely in one direction and then verify the program's correctness using the generalized Swarm programming logic outlined in Chapter 8. This program uses the synchronic group feature of Swarm.

Chapter 10 discusses the contributions of this work to the field of concurrent programming. Chapter 11, the final chapter, outlines several directions for future research.

The proofs in Chapters 7 and 9 use the Swarm analogues of several theorems proved for the UNITY logic. These theorems, from Chapter 3 of Chandy and Misra's book [24], are listed in the Appendix.

# 2. BACKGROUND

In Chapter 1 we examined two programming models that have greatly affected the direction of our shared dataspace research, UNITY and Linda. In this chapter, we survey some of the other related research in concurrent programming languages, models, and logics on which we have drawn. Filman and Friedman's book [47] and Andrews and Schneider's survey article [48] provide good starting points for a survey of concurrent programming issues. Books by Manna [49], de Bakker [50], Gries [51], Loeckx and Sieber [52], and Barringer [53] survey various results of programming logic and verification research.

## 2.1. PROGRAMMING LANGUAGES AND MODELS

For purposes of this discussion we classify programming languages and models by the primary way in which data and control information is communicated among the concurrent components of a program. Our taxonomy is similar the ones used in [25] and [48]. We group programming languages into four categories:

- shared variables,
- message passing,
- remote operations,
- shared dataspace.

*Shared variables* (e.g., monitors) provide controlled access to (physically or logically) shared data entities. Components of concurrent programs communicate

data and control information by the serial manipulation of these shared variables. *Message-passing* systems communicate by passing data and control messages along communication channels among the concurrent components. In languages based on *remote operations* (e.g., remote procedure calls), subtasks are carried out for a component by other components. There is a master/slave control relationship between the components; data are communicated via argument/result linkages. In the *shared dataspace* paradigm, concurrent components of a program can communicate by manipulating the dataspace, e.g., by inserting data into or deleting them from the dataspace. Elements in the dataspace are addressed by content rather than by name or address. The dataspace is viewed as being directly accessible to many processes simultaneously.

### 2.1.1. Shared Variables

As noted above, shared variables provide controlled access to physically or logically shared data entities. The concurrent components of a program communicate data and control information by the manipulation of these shared variables. The access to the shared variables by the components must be coordinated in some fashion, usually by requiring that a component obtain mutually exclusive access before it can use or change the value of a variable. A number of different mechanisms have been developed to enable a programmer to achieve this coordination [48]—busy waiting (spin locks) [54], semaphores [55], conditional critical regions [56], monitors [57], and path expressions [58]. A number of shared variable languages handle synchronization implicitly by expressing computations in terms of, perhaps quite complex, atomic operations which are executed in some serial order.

**Monitors.** Probably the most common way of supporting shared variables in programming languages is by means of monitors. A monitor encapsulates a set of

shared variables with a set of procedures for manipulating these variables. The values of the shared variables are retained between activations of the procedures and can be accessed only from within the monitor. The monitor's procedures are invoked by concurrent components of the program using the usual procedure call semantics, but execution of monitor procedures is guaranteed to be mutually exclusive. The most important languages that use monitors to synchronize access to shared variables are probably Brinch Hansen's Concurrent Pascal [59, 60] and Wirth's Modula [61] (whose descendent language Modula-2 [62] sees considerable current usage). The monitor concept has had little direct influence on the design of the Swarm programming model.

**Serialized atomic actions.** Chandy and Misra's UNITY [24] model and notation were discussed in Chapter 1. The UNITY notation is a very simple. The coordination of access to single variables is subsumed into the computational model (i.e., serial, atomic execution of multiple assignment statements) and the language design (i.e., conflicting assignments of values to variables are not allowed within a multiple assignment statement). No higher-level synchronization primitives are provided; if needed, they must be implemented by the programmer.

Two other recently proposed programming notations follow an "atomic action" approach similar to UNITY's. Instead of using conditional assignment statements, Shankar and Lam's Event Predicates approach [63, 64] expresses the atomic transformations of the values of the program's variables (i.e., the state) as predicates over the variables. The predicates have two parts, an enabling condition and an action relation. The enabling condition is a predicate on the state before an occurrence (execution) of the event. The action relation is a predicate which expresses a relationship between the state before the event's occurrence and the state afterward. Upon occurrence of the event, if the enabling condition is satisfied, then the values

of the variables are changed so that the action predicate is satisfied; otherwise the state is left unchanged. The fairness criteria for selection of events for execution differ from those of UNITY. While UNITY requires weak fairness [65] uniformly over the set of assignment statements, the Event Predicates approach allows each individual event to be assigned a different fairness attribute, e.g., weak fairness or no fairness. (See Section 2.2 for a discussion of fairness.)

Back and Kurki-Suonio's Action Systems [66] are also similar to UNITY in approach. In this system, the behavior of processes is described in terms of the possible interactions (actions) that the processes can engage in. A *process* is a grouping of variables and an *action* is a conditional statement labeled with the processes that execute the statement. The actions thus provide a symmetric communication mechanism that permits an arbitrary number of processes to be synchronized by a common handshake—a generalization of the usual asymmetric, two-way mechanism in a language like CSP. In addition to the UNITY-like sequential execution model, a concurrent execution model has been developed. In this model, concurrently executing actions must compete for access to the processes. The sequential model is the more convenient context for formal reasoning, while the concurrent model provides a more accurate representation of how a distributed implementation might actually work. Since the fairness notions differ between the two models, properties proved with respect to one model may not hold with respect to the other. The developers have studied the relationship between the two models and defined a class of action systems in which properties proved with respect to the sequential model also hold with respect to the concurrent model.

Swarm's model represents a program execution as a sequence of atomic actions similar to UNITY, Event Predicates, and Action Systems. Accordingly, some of this work has been (or can be) adapted to Swarm. However, since Swarm has a

different type of operation (content-addressable transactions), a slightly different notion of fairness, and allows the dynamic creation of new instances of "actions," we had to reformulate and extend several of the concepts.

**Data parallel languages.** Also of interest are the data-parallel languages [67] such as those designed for execution on a SIMD (Single-Instruction, Multiple-Data stream) architecture like the Connection Machine (CM) [68, 69]. In data parallelism the parallelism comes from simultaneous operations across large sets of data rather than from multiple threads of control. In the CM, for instance, each data element is stored on a separate CM processor and one serial program on the host machine directs the simultaneous operations on all the processors. For example, under direction of a host program the processors can carry out an operation on all elements of an array simultaneously.

Several data parallel languages have been developed. Data parallel languages for the CM include C* [70] and *Lisp [71], extensions to C and Common Lisp respectively. A CM implementation of the proposed FORTRAN 8x standard, which contains array operations, has also been developed [72]. Data parallelism is not restricted to SIMD machines like the Connection Machine, but can execute on MIMD (Multiple-Instruction, Multiple Data stream) machines as well. For example, C* has been implemented for the NCUBE hypercube multicomputer [73].

Although the data-parallel languages have not had much direct influence on Swarm, the Connection Machine literature has been a valuable source of useful application algorithms.

### 2.1.2. Message Passing

Message-passing systems communicate by passing data and control messages along communication channels among the concurrent components. Such systems

come in many different flavors: synchronous or asynchronous, buffered or unbuffered, reliable or unreliable, point-to-point or broadcast, datagram or virtual circuit, and so forth. Here we focus on two language approaches, Hoare's CSP and Hewitt's Actors.

**CSP.** Hoare's Communicating Sequential Processes (CSP) [3] language fragment is based on a simple idea—input/output commands—carefully integrated with a few other simple mechanisms [48]. CSP uses a variant of Dijkstra's Guarded Commands [74, 75] language to specify the sequential processes. To this base language, Hoare adds input/output commands as the mechanism for synchronization and communication among the processes. The message passing among processes is synchronous and two-way—a process executing an output command synchronizes with a process executing a matching input command. Input and output commands match if the type of the source value being transmitted by the output command is compatible with the receiving target variable on the input command. The communication structure is static—an output command must directly name the destination process and vice versa. However, by allowing input commands to appear in guards of the alternative and iteration statements, CSP supports a limited amount of selectivity in communication. Since Hoare proposed the language in 1978, other researchers have generalized CSP somewhat—replacing the static naming with ports [76, 77] and allowing output commands on guards [20]. The CSP ideas form the basis for the language Occam [14, 15] which is supported on the INMOS transputer [16].

**Actors.** The Actor model [6, 78, 79], a message passing framework being developed by Carl Hewitt and his colleagues at MIT, "takes the theme of object-oriented programming seriously and to an extreme." [47] In a pure actor model all programming constructs (procedures and data) are represented as (active) objects

called *actors*. Actors have unique addresses; they communicate by sending messages to other actors and by creating new actors. Actors are implemented as serialized closures, i.e., functional code within a specialized environment waiting to be applied to (accept) a message matching a specific pattern. The specification of an actor enables considerable parallelism during execution. The model also promotes an *open systems* [6, 80] approach by providing for dynamic growth and reconfiguration of programs.

The Actors model has been an active area of research since the mid-1970's. Several Actor-based languages have been defined, including Plasma, Act-1, Act2, Act3, Ether, and the Omega description system [81]. An Apiary multicomputer architecture, consisting of a group of Lisp machines, has been implemented for executing actor languages. The MCC consortium is developing the Rosette [82] architecture and language, an actor-based system for dynamically structured, concurrent computations.

Neither CSP nor Actors have had a direct impact upon the Swarm research. CSP's impact is cultural; before undertaking the shared dataspace research, we investigated the use of a CSP-like notation for specification of distributed systems [18, 19]. The CSP paradigm carried over into our early shared dataspace research [34, 83]. The Actors model has indirectly influenced Swarm. The programming styles encouraged by Actors—a fine-grained, dynamic, message-passing model—and Swarm—a fine-grained, dynamic, shared dataspace model—are similar.

### 2.1.3. Remote Operations

In languages based on remote operations, subtasks are carried out for a concurrent component by subordinate components. There is a master/slave control relationship between the components; data are communicated via the argument/result

linkages of a remote procedure call. Examples of remote operation languages [48] include Ada [2, 84], Brinch Hansen's Distributed Processes [85], Andrews' Synchronizing Resources (SR) [86, 87], and Liskov's Argus [88, 89]. Here we look at the Ada and Argus languages.

**Ada.** The Ada language was developed by the U.S. Department of Defense for use in programming of embedded systems, i.e., real-time, process-control software deployed as a part of a computer system embedded within a weapon or other product [48]. Ada processes are called *tasks*; the intertask communication mechanism is the *rendezvous*, a type of remote procedure call. The remote procedures, called *entries*, are ports into a server process; entries are specified by **accept** statements and are invoked by remote **call** statements. Interrupts can also be treated as remote calls of entries. The **select** statement, a construct similar to CSP's alternative command, allows a server to select among several competing entry calls. Remote procedure calls normally block unconditionally until the accept is processed, but several other alternatives are supported.

**Argus.** In the Argus language, being developed by Barbara Liskov's group at MIT, processes also interact by means of remote procedure calls. Argus has a *guardian* construct which encapsulates a computing resource; a guardian consists of data objects and handlers which manipulate the data. Handler calls result in the creation of concurrent processes within the guardian to carry out the operations. Argus integrates the data abstraction concepts of the programming language CLU [90] with ideas from the realm of database systems, e.g., atomic actions, recovery, and two-phase commit protocols, to form an elegant and practical language for distributed processing.

No language based on remote operations has had much direct influence on Swarm. However, by successfully integrating database concepts (e.g., atomic

actions) into a programming language for distributed computation, the results of the Argus research have encouraged our investigation of the shared dataspace programming paradigm.

### 2.1.4. Shared Dataspace

In shared dataspace programs, concurrent components communicate by manipulating a content-addressable data structure called a dataspace, e.g., by inserting data into or deleting them from the dataspace. Unlike shared variables, the dataspace is viewed as being directly accessible to many concurrent components simultaneously. Unlike message-passing schemes, the communication medium is viewed as a passive structure rather than an active agent. Although Roman coined the term shared dataspace relatively recently [34], a number of older languages fit into this paradigm. One such language, Linda [25, 26, 27], has had considerable influence on Swarm—as discussed in Chapter 1. Other shared dataspace languages include production rule languages, Associons, Transaction Networks, distributed versions of Prolog, and the data language for the Abstract Database System (ADS) [91]. Here we look at production rule languages, Associons, and Transaction Networks in more detail.

**Production rules.** In a rule-based language such as OPS5 [39], a program consists of an unordered set of production rules (stored in a production memory) and a set of data items (stored in a working memory). A rule is a guarded command in which the guard (left-hand-side) is a predicate and the command (right-hand-side) is a sequence of actions that create new data items, create new rules, or change the values of existing items. Data items are normally record-like structures (tuples); a rule's predicate matches data items based on the contents of the various fields. The execution of a program follows a three-step cycle. In the first step, the

execution mechanism finds all rules whose predicates are satisfied by the current contents of memory. These pairings of rules with data items that satisfy the rules' predicates are candidates for execution. (Here we describe what is called a forward-chaining matching strategy. Some rule-based systems use a backward-chaining strategy where the right-hand-sides are matched.) In the second step, using a complex selection (conflict resolution) strategy, the execution mechanism selects a set of pairings for execution. In the third step, the actions of the selected rule-data pairs are taken.

Swarm is, in essence, a rule-based model. However, it differs from most such systems in that its rule selection strategy is quite simple—nondeterministic, but fair selection—and it has an explicit notion of actions being taken in parallel.

**Associons.** Martin Rem's Associons model [35, 36] proposes a programming notation where the state of a computation is recorded as a set of *associons* (tuples). The state of a computation is changed by the creation of new associons deducible from those already recorded. The basic mechanism for these deductions is the *closure statement*, whose execution can involve a high degree of concurrency. A closure statement can be considered as implicitly specifying a community of anonymous processes that collectively compute the transitive closure of a relation. These processes communicate via an associon "database" (dataspace). In the development of the Associon notation, Rem emphasized the careful definition of the mathematical properties and correctness proving techniques.

The Associon model provides an elegant and appealing notation for fine-grained concurrent computation. The use of anonymous, implicit processes eliminates the need for a complex name management scheme (at least one that is visible to the programmer). It also aids in the scaling of the computation to fit the available computing resources; the number of processes implementing a closure statement

can be increased or decreased without requiring a modification of the program. The model does not support explicitly concurrent processes or allow separately specified modules to be composed into a new program in a straightforward manner.

**Transaction Networks.** Kimura's Transaction Network model [37] "demotes the notion of process as the key concept in organizing large scale parallel computation." It "promotes, instead, the notion of transaction, an anonymous atomic action void of internal state, as the basic element of computation." These transactions are organized into a network structure similar to a Petri net [92] where the transitions are replaced by transactions, the places by databases, and the tokens by tuples. The transaction net "fires" according to rules based on a *consume/produce* paradigm, consuming tuples from a transaction's input databases and producing new tuples in the output databases. A "two-dimensional graphic structure" is the means for expressing computations as transaction networks.

In some ways Transaction Networks and Swarm are similar—both lines of research arising from a period of collaboration in 1987. The transaction concept is similar in the two models. However, while Transaction Networks use a two-dimensional, graphic representation for its static program structures, Swarm uses a textual notation to represent concurrent programs that are highly dynamic in structure and degree of concurrency.

### 2.1.5. Other Paradigms

The preceding discussion is not a comprehensive survey of all concurrent language work related to this research effort, but it does briefly discuss several models that have had some influence on our approach. Most of the languages discussed above follow an imperative style and have an explicit notion of concurrency. Of

course, researchers from other programming language communities are also interested in parallelism.

The field of logic programming is quite interested in parallelism—encouraged in their research efforts by the Japanese Fifth Generation Project. The best known "concurrent" logic programming languages include Shapiro's Concurrent Prolog [5, 93], Clark and Gregory's Parlog [94, 95, 96], and Ueda's Guarded Horn Clauses [97]. Implementation of the transaction queries in Swarm is similar to implementation of *unification* in such languages.

The field of functional languages has also seen considerable activity related to parallelism. In addition to *Lisp for the Connection Machine (discussed in Section 2.1.1), various Lisp-like concurrent languages have been proposed and implemented, e.g., Multilisp [98, 99], Qlisp [100], and Spur Lisp [101]. Other approaches from the functional language realm are also interesting. In parafunctional programming [102, 103] the functional behavior (e.g., the application algorithm) is specified separately from the parafunctional behavior (e.g., the mapping of the program structures to a particular set of processors and memories). Combinator-based languages such as Backus' FP [4] have also stimulated considerable research. The influence of these functional languages upon Swarm has not been extensive. However, the powerful operators in FP and the ZF (generator) expressions in the functional language Miranda [104, 105] did provide some of the stimulus for making the constructor notation a prominent feature of the Swarm notation.

Before moving on to a discussion of programming logics, one other approach to parallelism in programming languages should be mentioned briefly, that of compiling standard sequential languages to parallel machines [106]. Although this is an important area of research that has been quite beneficial, we believe that there is a limit to what can be accomplished by parallelizing compilers. For example,

compilers cannot find parallelism that is not there; the algorithms that work best on parallel machine are often different from the ones that work well in sequential programs [27]. We believe that, with the "right" programming models, the "right" formal and technological tools, and the "right" programming culture, concurrent programming is not necessarily an onerous task. A goal of the shared dataspace research is to help define what "right" means.

## 2.2. PROGRAMMING LOGICS

In this section we examine some of the historical development of programming logics from the perspective of their influences on the development of the Swarm logic given in Chapters 6 and 8. We first discuss early work on logics for sequential programming languages and then later research on logics for concurrent programming languages.

There are two general classes of program properties—safety and progress (often called liveness) properties. The work of Alpern and Schneider [107, 108, 109] addresses these classes of properties in a formal manner.

Informally, a safety property states that nothing "bad" ever happens, i.e., the program never enters an unacceptable state [110]. Safety properties of interest include:

- partial correctness—if the program begins in a valid initial state, then it never terminates in an unacceptable state,
- absence of deadlock—the program never enters a state in which no further progress is possible,

- mutual exclusion—two different processes are never in their critical sections at the same time.

A progress property states that something "good" eventually does happen, i.e., the program eventually enters a desirable state. Progress properties of interest include program termination, the attainment of some stable state, and guaranteed delivery of messages. Total correctness is partial correctness plus termination—if a program begins in a valid initial state, then it always terminates in an acceptable state.

An important issue for progress properties is a concept we mentioned in the previous section—*fairness*. To illustrate the meaning of fairness, consider a repetitive choice among alternatives. In this context, fairness means that, in repeated choices among a set of alternatives, no choice is postponed forever [65]. A number of different notions of fairness have been defined—of primary significance are weak and strong fairness. If an event is *continuously enabled*, then weak fairness guarantees that the event will eventually occur. Strong fairness, on the other hand, guarantees the event will eventually occur if the event is *infinitely often enabled.* For example, the assignment statements in UNITY programs, which are always enabled for execution, are scheduled in a weakly fair manner; the actions in Action Systems are scheduled in a strongly fair manner. In CSP both the selection of an enabled guard and the selection of matching I/O commands for execution are considered anti-fair, i.e., no fairness is assumed. As illustrated by a recent exchange of position papers [111, 112, 113], the inclusion of fair constructs in programming languages is still controversial.

### 2.2.1. Sequential Programming

The notion of formal proofs of correctness for computer programs began to be discussed seriously in the mid-1960's. The earliest significant work was that of Floyd [114] in 1967, later extended by Manna [49] and others. Floyd's method concerned verification of flowchart programs. In his method an input predicate, a logical assertion describing the initial state of the program, is attached to the edge coming from the START node, an output predicate, an assertion describing the valid final state of the program, is attached to the edges going into the HALT nodes, and appropriate assertions describing the intermediate program states are attached to the other edges. To verify the program, one must show that each execution path from START to HALT maintains the truth of the assertions encountered. Floyd also introduced what has come to be known as a loop invariant. He attached an assertion at some edge in a cycle, called a cutpoint. Proof of this assertion requires that, for any execution beginning at the cutpoint with the assertion true, the assertion will be true whenever execution returns to the cutpoint.

Building on the work of Floyd, in 1969 Hoare defined the partial correctness semantics of a simple programming language by means of a system of logical axioms and inference rules. Hoare's system, an extension of the predicate calculus, included rules for assignment, **if-then-else**, and **while** statements and sequences thereof. Hoare's expressions are usually written in the form

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

where $p$ and $q$ are predicates over the values of the program's variables and $S$ is some program construct. The Hoare "triple" means that, if $p$ is true immediately prior to execution of $S$, then $q$ is true immediately after execution. To verify the

partial correctness of a program *Prog* for input condition *in* and output condition *out*, the Hoare method requires that we prove

$$\{in\}\ Prog\ \{out\}.$$

Typically a proof of this assertion proceeds as follows: the program text is first annotated with appropriate intermediate assertions between each statement of the program, then, by application of the axioms and rules of inference with respect to the program's text, the annotation is shown to be correct.

Much of the research on program verification during the past two decades has followed Hoare's axiomatic approach to semantics. The method has been extended to support total correctness proofs and additional language features.

By the mid-1970's, interest began to develop in using program correctness notions to derive programs. Building on Hoare's approach, Dijkstra introduced a "calculus of weakest preconditions" (i.e., *wp*-calculus) for the simple Guarded Commands language [74, 75]. In addition to proofs of correctness, he showed how the calculus could be used to derive programs from their input/output specifications. This seminal work laid the foundation for the active study of derivation of sequential programs that has developed during the past 15 years. The concept of developing a program hand-in-hand with its proof has become widely accepted, but not, however, as yet widely used by practitioners.

### 2.2.2. Concurrent Programming

By the mid-1970's attention was also turning to concurrent programming. In some significant early work, Owicki and Gries [115, 116] extended Hoare's partial correctness logic for sequential programs to shared-variable concurrent programs. The Owicki-Gries method requires proofs of "interference freedom" among

the statements of the concurrent processes. The method allows the addition of auxiliary variables to capture information about the control state of the program. Further work on shared variables languages has been done by Keller [117], Lamport [118], and others.

Lamport [11, 119] advocates a view of programs as "invariance maintainers." Instead of using an annotated program with auxiliary variables to implicitly encode the control state, Lamport's approach uses assertions which explicitly mention the control state by means of control point predicates. In this Control Point Hoare Logic, the safety properties of a program are stated as sets of global program invariants and the use of auxiliary variables is avoided.

Hoare's CSP notation has also been the subject of considerable research in program verification. The first known partial correctness proof system for CSP is that of Apt [10]. In Apt's system, the properties of the individual sequential processes are proved using assumptions about the behavior of the remaining processes in the program. Then, to prove the processes compose correctly into a program, these assumptions must be justified. This is done by means of a "cooperation" proof.

In later work, Soundararajan presents a proof system for CSP in which the individual processes can be proved in isolation—without assumptions about the behavior of other processes [12]. He introduces a "communication sequence" to record the communication activity for each process. The content of this sequence is used in reasoning about the process—the proofs do not use auxiliary variables. At the termination of execution the communication sequences will be mutually compatible, hence complex cooperation arguments are avoided. This approach has been extended to deal with the total correctness of CSP programs [13].

Another approach to the verification of concurrent programs, particularly of progress properties, is the use of temporal logic [120, 121, 122], first applied to

concurrent program verification by Pnueli [123]. Temporal logic is an extension of predicate calculus to include certain kinds of assertions about time. A common form of temporal logic is linear-time logic. Using this logic, the execution of programs is typically represented as infinite sequences of atomic events—often the atomic events from the processes of a program interleaved in some fashion. The primary temporal operators in this language are $\Box$, meaning "now and forever," and $\Diamond$, meaning "now or sometime in the future." In its basic form, proof of the temporal properties requires arguments over the set of execution sequences. In [110] Owicki and Lamport structure temporal logic proofs of liveness properties of shared-variable programs by means of "proof lattices." (Their logic assumes weak fairness of statement execution.)

This brings us now to Chandy and Misra's UNITY. Their proof theory utilizes a subset of the linear-time temporal logic, but, instead of using execution sequences from an operational model directly, it uses a Hoare-style axiomatic approach [24]. Thus reasoning at a low level—in terms of execution sequences—is unnecessary. The key elements of the logic are the standard multiple assignment axiom, the **unless** temporal relation (between two predicates) for stating basic safety properties, and the **ensures** temporal relation for stating elementary progress properties. Invariant properties are defined in terms of the **unless** relation. More complex progress properties are stated with the leads-to relation $\longmapsto$, which is defined in terms of **ensures** properties. The assumption of weakly fair scheduling of statement executions is essential to the logic—without fairness, the **ensures** property does not guarantee that the computation would ever progress. In an effort to put UNITY on a firmer formal foundation, Gerth and Pnueli [124] have defined the UNITY logic in terms of Manna and Pnueli's temporal logic [125] and Gerth's transition logic [126].

The Swarm logics given in Chapters 6 and 8 follow the approach advocated by UNITY. Like UNITY assignments, Swarm transactions are executed in weakly fair, nondeterministic order. The Swarm programming logics are based on the same logical relations as UNITY—**unless** and **ensures**.

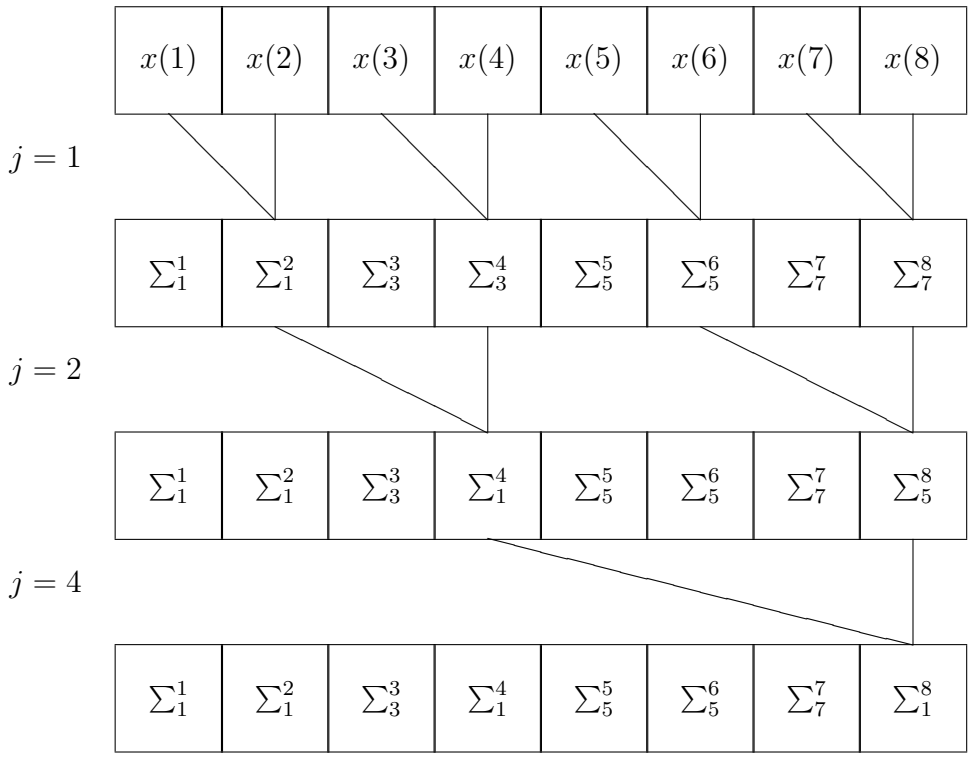
# 3. THE SWARM NOTATION

This chapter informally presents the syntax and semantics of the Swarm concurrent programming notation. The first section introduces the features of the programming notation by analogy to a more familiar procedural language. The remaining sections, following our presentation in [38], discuss the Swarm features in more detail.

## 3.1. AN INFORMAL INTRODUCTION

As noted in Chapter 1, our choice of the name *Swarm* evokes the image of a large, rapidly moving aggregation of small, independent agents cooperating to perform a task. In this section we introduce a notation for programming such computations. We first present an algorithm expressed in a familiar imperative notation—a parallel dialect of Dijkstra's Guarded Commands [75] language. We then construct a Swarm program with similar semantics.

The algorithm given in Figure 3.1 (adapted from the one given in [67]) sums an array of $N$ integers. For simplicity of presentation, we assume that $N$ is a power of 2. In the program fragment, $A$ is the "input" array of integers to be summed and $x$ is an array of partial sums used by the algorithm. Both arrays are indexed by the integers 1 through $N$. At the termination of the algorithm, $x(N)$ is the sum of the values in the array $A$. The loop computes the sum in a tree-like fashion as shown in the diagram: adjacent elements of the array are added in parallel, then the same is done for the resulting values, and so forth until a single value remains.




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

$j := 1 \,;\, \langle\, k : 1 \leq k \leq N :: x(k) := A(k) \rangle \,;$
**do** $j < N \longrightarrow$
$\quad \langle \| \;\; k : 1 \leq k \leq N \wedge k \,\mathbf{mod}\, (j * 2) = 0 ::$
$\quad\quad x(k) := x(k - j) + x(k) \rangle \,;$
$\quad j := j * 2$
**od**

Figure 3.1: A Parallel Array Summation Algorithm Using Guarded Commands

The construct

$$\langle \| \ k : predicate :: assignment \rangle$$

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

Swarm is a shared dataspace programming model. Instead of expressing a computation in terms of a group of named variables, Swarm uses a set of *tuples* called a *dataspace*. Each tuple is a pairing of a type name with a finite sequence of values; a program accesses a tuple by its content—type name and values—rather than by a specific name or address.

Swarm programs execute by deleting existing tuples from and inserting new tuples into the dataspace. The *transactions* which specify these atomic dataspace transformations consist of a set of *query-action* pairs executed in parallel. Each query-action pair is similar to a production rule in a language like OPS5 [39].

How can we express the array summation algorithm in Swarm? To represent the array $x$, we introduce tuples of type $x$ in which the first component is an integer in the range 1 through $N$, the second a partial sum. Assuming that $j$ and $k$ are constants, we can express an instance of the array assignment in the **do** loop as a Swarm transaction in the following way:

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

In the above notation, the part to the left of the "$\longrightarrow$" is the query; the part to the right is the action. The identifiers $v1$ and $v2$ designate variables that are local to the query-action pair.

The execution of a Swarm query is similar to the evaluation of a clause in Prolog [127]. The query in the paragraph above causes a search of the dataspace

for two tuples of type $x$ whose component values have the specified relationship—the comma separating the two tuple predicates is interpreted as a conjunction. If one or more solutions are found, then one of the solutions is chosen nondeterministically and the matched values are bound to the local variables $v1$ and $v2$ and the action is performed with this binding. If no solution is found, then the transaction is said to fail and none of the specified actions are taken.

The action of the above transaction consists of the deletion of one tuple and the insertion of another. The † operator indicates that the tuple $x(k, v2)$, where $v2$ has the value bound by the query, is to be deleted from the dataspace. The unmarked tuple form $x(k, v1 + v2)$ indicates that the corresponding tuple is to be inserted. Although the execution of a transaction is atomic, the effect of an action is as if all deletions are performed first, then all insertions.

The parallel assignment command of the algorithm can be expressed similarly in Swarm:

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

We call each individual query-action pair a *subtransaction* and the the overall parallel construct a *transaction*. As with the parallel assignment, the entire transaction is executed atomically. The cumulative effect of executing a transaction is as if the subtransactions are executed synchronously: all queries are evaluated first, then the indicated tuples are deleted, and finally the indicated tuples are inserted.

Like data tuples, transactions are represented as tuple-like entities in the dataspace. A transaction instance has an associated type name and a finite sequence of values called parameters. A subtransaction can query and insert transaction instances in the same way that data tuples are inserted, but transactions cannot

be explicitly deleted. Implicitly, a transaction is deleted as a by-product of its own execution—regardless of the success or failure of its component queries.

Two aspects of the **do** command in Figure 3.1 have not been translated into Swarm—the doubling of $j$ and the conditional repetition of the loop body. Both of these can be can be incorporated into a transaction. We define a transaction type called *Sum* as follows:

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

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

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

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

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

Since each $x$ tuple is only referenced once during a computation, we can modify the *Sum* subtransactions to delete both $x$ tuples that are referenced. A complete *ArraySum* program with this modification is given in Figure 3.2. The **program**, **tuple types**, and **transaction types** portions of the program declare program structures. The **initialization** section defines the contents of the initial dataspace.

In the remainder of this chapter, we discuss the syntax and semantics of Swarm programs in more detail.

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

Figure 3.2: A Parallel Array Summation Program in Swarm

**initialize the dataspace;**
**while the transaction space is not empty**
  **select a transaction fairly;**
  **evaluate the transaction's query;**
  **if the query succeeds**
    **delete tuples & the transaction;**
    **insert tuples & transactions;**
  **else**
    **delete the transaction;**

$$\boxed{D_0} \xrightarrow{t_0} \boxed{D_1} \xrightarrow{t_1} \boxed{D_2} \xrightarrow{t_2} \boxed{D_3} \xrightarrow{t_3} \boxed{D_4} \xrightarrow{t_4} \boxed{D_5} \xrightarrow{t_5} \cdots$$

Figure 3.3: Swarm Execution Model

## 3.2. BASIC CONCEPTS

**Execution Model.** 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 three subsets: the *tuple space*, a finite set of data *tuples*; the *transaction space*, a finite set of *transactions*; and the *synchrony relation*, which is discussed at the end of the chapter. 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 program execution model shown in Figure 3.3 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. (In Chapter 5 we discuss the operational model more formally.)

Before further 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 $\exists$, $\forall$, $\Leftrightarrow$, $\Sigma$, $\Pi$, **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 and the preceding colon is omitted.) 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 $\Sigma$, 1 for $\Pi$, **false** for $\exists$, and **true** for $\forall$.

**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 denote tests of the dataspace for membership by tuples and transactions. For example, an execution of the predicate $has\_label(17, \lambda)$, where $\lambda$ 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 $\lambda$.

**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 of a $has\_label(P, L)$ tuple for all values of $P$ and $L$ that satisfy the predicate $Pixel$. We do not allow the domain predicate of a generator to contain dataspace membership tests.

## 3.3. 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 3.4 shows a Swarm program to label each pixel in equal-intensity regions of a digital image with the smallest *xy*-coordinate pair in the region. (We order the pixel coordinates lexicographically, i.e., $(x, y) < (a, b)$ if and only if $x < a \vee (x = a \wedge y < b)$.) Various versions of this program are used as examples in the chapters that follow.

**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 3.4 is named *RegionLabel*; 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 *RegionLabel* program in Figure 3.4 defines predicates $Pixel(P)$, $neighbors(P, Q)$, and $R\_neighbors(P, Q)$. These predicates allow the other sections to be expressed in a more concise and readable fashion.

**program** $RegionLabel(M, N, Lo, Hi, Intensity :$
$\quad 1 \leq M, 1 \leq N, Lo \leq Hi, Intensity(\rho : Pixel(\rho)),$
$\quad [\forall\, \rho : Pixel(\rho) :: Lo \leq Intensity(\rho) \leq Hi]\ )$
**definitions**
$[\, P, Q, L ::$
$\quad Pixel(P)\ \equiv\ [\exists\, x, y : P = (x, y) :: 1 \leq x \leq N, 1 \leq y \leq M];$
$\quad neighbors(P, Q)\ \equiv$
$\quad\quad Pixel(P), Pixel(Q), P \neq Q,$
$\quad\quad [\exists\, x, y, a, b : P = (x, y), Q = (a, b) :: a - 1 \leq x \leq a + 1, b - 1 \leq y \leq b + 1];$
$\quad R\_neighbors(P, Q)\ \equiv$
$\quad\quad neighbors(P, Q), [\exists\, \iota :: has\_intensity(P, \iota), has\_intensity(Q, \iota)]$
$]$
**tuple types**
$[\, P, L, I : Pixel(P), Pixel(L), Lo \leq I \leq Hi ::$
$\quad has\_label(P, L);$
$\quad has\_intensity(P, I)$
$]$
**transaction types**
$[\, P : Pixel(P) ::$
$\quad Label(P)\ \equiv$
$\quad\quad \rho, \lambda 1, \lambda 2 :$
$\quad\quad\quad has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2),\ R\_neighbors(P, \rho), \lambda 1 > \lambda 2$
$\quad\quad\quad\quad \rightarrow\ has\_label(P, \lambda 2), Label(P)$
$\quad\quad \|\quad$ **NOR** $\rightarrow\ Label(P)$
$]$
**initialization**
$[\, P : Pixel(P) ::$
$\quad has\_label(P, P),$
$\quad has\_intensity(P, Intensity(P)),$
$\quad Label(P)$
$]$
**end**

Figure 3.4: A Nonterminating Region Labeling Program in Swarm

**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 3.4 tuple type *has_label* pairs a pixel with a label; *has_intensity* pairs a pixel with its intensity value.

**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 3.4 declares a transaction type $Label(P)$.

The body of a transaction instance consists of a sequence of subtransactions connected by the $\|$ operator:

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

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: the queries are evaluated simultaneously, then the indicated deletions are performed for all subtransactions, and finally the indicated tuple and transaction insertions are done.

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

**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.

## 3.4. SYNCHRONY RELATION

In our discussion so far we have ignored the third component of a Swarm program's state—the *synchrony relation*. The interaction of the synchrony relation with the execution mechanism provides a dynamic form of the $\|$ operator. The synchrony relation is a symmetric, irreflexive relation on the set of valid transaction instances. (Picture an undirected graph with transaction instances represented as nodes and a synchrony relationship between transaction instances as an edge between the corresponding nodes.) The reflexive transitive closure of the synchrony relation is thus an equivalence relation. (The equivalence classes are the connected components of the undirected graph mentioned above.) When one of the transactions in an equivalence class is chosen for execution, then all members of the class which exist in the transaction space at that point in the computation are also chosen. This group of related transactions is called a *synchronic group*. The subtransactions making up the transactions of a synchronic group are executed as if they were part of the same transaction. The scope of the global predicates, e.g., **AND**, extends to all local subtransactions in the synchronic group.

The synchrony relation can be examined and modified in much the same way as the tuple and transaction spaces can. For example, the predicate

$$Label(p) \sim Label(Q)$$

(where parameter $p$ is a variable and parameter $Q$ is a constant) in the query of a subtransaction examines the synchrony relation for a transaction instance $Label(p)$ that is directly related to an instance $Label(Q)$. Neither transaction instance is required to exist in the transaction space. The operator $\approx$ can be used in a predicate to examine whether transaction instances are related by the closure of the synchrony relation.

Synchrony relationships between transaction instances can be inserted into and deleted from the relation. For example, the operation

$$Label(p) \sim Label(q)$$

in the action of a subtransaction creates a dynamic coupling between transaction instances $Label(p)$ and $Label(q)$ (where $p$ and $q$ must have bound values). If two transaction instances $Label(p)$ and $Label(q)$ are related by the synchrony relation, then

$$(Label(p) \sim Label(q))\dagger$$

deletes the relationship. Note that the closure relation can be examined, but that only the base synchrony relation can be modified. (The dynamic creation of a synchrony relationship between two transactions can be pictured as the insertion of an edge in the undirected graph described earlier in the section, and the deletion of a relationship as the removal of an edge.)

By default the synchrony relation is empty. Initial couplings can be specified by putting insertion operations into the initialization section. For the purposes here, we assume that any two transaction instances can be related by the synchrony relation.

# 4. SWARM PROGRAMMING

In Chapter 3 we introduced a mechanism for examining and modifying the dataspace—the transaction. A transaction consists of a fixed set of subtransactions connected by the $\|$ operator. The subtransactions of a transaction are executed synchronously. Transactions may be coupled by means of the synchrony relation into synchronic groups which are executed asynchronously with respect to each other. Of course, a synchronic group may contain only one transaction, having, in turn, a single subtransaction. In this chapter we provide some of the motivation for these particular choices. We do this by identifying the kinds of programming strategies made possible by these constructs.

In this chapter we discuss a series of solutions to the region-labeling problem introduced in Chapter 3. Most of the programs in this chapter are variations of the *RegionLabel* program given in Figure 3.4. To distinguish among similar transactions in the various solutions, we append unique numbers to the base transaction names.

Given a digital image in which each pixel has an intensity attribute, a region-labeling program must assign unique labels to each connected, equal-intensity region of the image. For the region's label, the programs in this chapter use the "minimum" of the set of $xy$-coordinates for pixels in the region. Comparisons of pixel coordinates are in terms of the lexicographic ordering where, for example,

$$(x, y) < (a, b) \equiv x < a \vee (x = a \wedge y < b).$$

Reasoning about concurrent computations is generally done in terms of progress (liveness) and safety properties (e.g., stability). Progress is achieved by effecting

changes in the computation's state; stable properties are useful in detecting the completion of a particular phase of the computation. For these reasons our discussion is logically divided into two parts: computational progress and stable state detection.

## 4.1. COMPUTATIONAL PROGRESS

The manner in which progress is accomplished depends upon the computational style supported by the underlying model. Swarm supports both asynchronous and synchronous computation in the context of either a static or dynamic transaction space. These capabilities are illustrated below by considering the region-labeling problem. Throughout this section we will ignore the issue of termination detection and assume that any transaction which cannot change the labeling result is harmless. We could inhibit the creation of such transactions, but we prefer to keep the presentation simple.

**Static asynchronous computation.** First, we consider the case of an asynchronous computation with a static transaction space. In a manner similar to Figure 3.4, the transaction space consists of one transaction per pixel:

$$
\begin{array}{l}
[\, P : Pixel(P) :: \\
\quad has\_label(P, P), has\_intensity(P, Intensity(P)), \\
\quad Label1(P) \\
]
\end{array}
$$

Each $Label1(P)$ transaction reinserts itself and thus leaves the transaction space unchanged:

$$
\begin{array}{l}
[\, P : Pixel(P) :: \\
\quad Label1(P) \;\equiv \\
\qquad\quad \rho, \lambda 1, \lambda 2 : \\
\qquad\qquad\quad has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2), R\_neighbors(P, \rho), \lambda 1 > \lambda 2 \\
\qquad\qquad\qquad\quad \rightarrow \; has\_label(P, \lambda 2) \\
\qquad\quad \| \qquad \mathbf{true} \;\; \rightarrow \; Label1(P) \\
]
\end{array}
$$

Each $Label1$ transaction is anchored at a pixel; the transaction repeatedly relabels its pixel to smaller labels held by neighbor pixels. Eventually the winning label propagates throughout the entire region.

**Dynamic asynchronous computation.** A very different kind of solution may be obtained if we allow a dynamic transaction space. As before, we can start with one transaction associated with each pixel in the image:

$$[\, P : Pixel(P) :: \; \cdots, Label2(P, P)]$$

Each transaction, however, has two arguments. The first argument is the pixel the transaction is attempting to label; the second is the label it is attempting to place on that pixel:

$$
\begin{array}{l}
[\, P, L : Pixel(P), Pixel(L) :: \\
\quad Label2(P, L) \;\equiv \\
\qquad\quad [\| \; \delta : P = L, neighbors(P, \delta) :: \\
\qquad\qquad\quad \iota : has\_intensity(P, \iota), has\_intensity(\delta, \iota) \\
\qquad\qquad\qquad\quad \rightarrow \; Label2(\delta, P) \\
\qquad\quad ] \\
\qquad \| \\
\qquad\quad \lambda : has\_label(P, \lambda)\dagger, \lambda > L \\
\qquad\qquad\quad \rightarrow \; has\_label(P, L) \\
\qquad \| \\
\qquad\quad [\| \; \delta : \delta \neq L, neighbors(P, \delta) :: \\
\qquad\qquad\quad \lambda, \iota : has\_label(P, \lambda), \lambda > L, \\
\qquad\qquad\qquad\quad has\_intensity(P, \iota), has\_intensity(\delta, \iota) \\
\qquad\qquad\qquad\quad \rightarrow \; Label2(\delta, L) \\
\qquad\quad ] \\
]
\end{array}
$$

Each $Label2(P, L)$ transaction consists of three groups of subtransactions. In the first and third groups we use the subtransaction generator feature. For $P = L$, the first group includes a subtransaction for each pixel $\delta$ such that $neighbors(P, \delta)$; otherwise, the group is null. This group of subtransactions starts the propagation of a pixel's label to its neighbors. The second subtransaction group is a single subtransaction which relabels pixel $P$ when it has a label larger than $L$. When a label is changed, the third subtransaction group propagates the relabeling activity to the pixel's neighbors. A wavefront of transactions working on behalf of the pixel having the smallest coordinate in the region, i.e., the winning pixel, will expand until it reaches the region boundaries where, having completed the region labeling, it dissipates.

**Static synchronous computation.** The synchronous version of the static transaction space is a highly unpleasing one. It demands the creation of a super-transaction that covers the entire image:

$$
\begin{array}{l}
[\ ::\\
\quad Label3(\,) \equiv\\
\qquad [\!|\ \rho : Pixel(\rho) ::\\
\qquad\quad \delta, \lambda1, \lambda2 : has\_label(\rho, \lambda1)\dagger, has\_label(\delta, \lambda2),\\
\qquad\qquad R\_neighbors(\rho, \delta), \lambda1 > \lambda2\\
\qquad\qquad\qquad \rightarrow\ has\_label(\rho, \lambda2)\\
\qquad ]\\
\qquad\ \|\qquad\quad \textbf{true}\ \ \rightarrow\ Label3(\,)\\
]
\end{array}
$$

This kind of solution, typical for many SIMD machines, creates an unnecessary coupling between independent regions of the image. Because the structure of the image varies, one cannot restrict a transaction to processing a single region.

For this reason Swarm includes the synchrony relation $\sim$. For static data, the synchrony relation may be used to create an initial configuration of the transaction space which is tailored to the initial structure of the tuple space. Using the earlier

definition of the $Label1(P)$ transaction type, we can redefine the initial configuration to be as follows:

$$
\begin{array}{l}
[\, P : Pixel(P) :: \\
\quad has\_label(P, P), has\_intensity(P, Intensity(P)), \\
\quad Label1(P) \\
\,] \; ; \\
[\, P, Q : neighbors(P, Q), Intensity(P) = Intensity(Q) :: \\
\quad Label1(P) \sim Label1(Q) \\
\,]
\end{array}
$$

All the transactions working on the same region form a synchronic group which reinserts itself after each step.

**Dynamic synchronous computation.** Synchronic groups can also be formed during program execution in response to dynamically created data. This brings us to the case of a synchronous solution in a dynamic transaction space. This approach can be illustrated by altering the definition of $Label1(P)$ so that it couples itself to those transactions that are associated with its neighbors in the same region:

$$
\begin{array}{l}
[\, P : Pixel(P) :: \\
\quad Label4(P) \;\equiv \\
\qquad \rho, \lambda 1, \lambda 2 : \\
\qquad\quad has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2), R\_neighbors(\rho, P), \lambda 1 > \lambda 2 \\
\qquad\qquad\qquad \rightarrow \; has\_label(P, \lambda 2) \\
\qquad \| \quad \rho : R\_neighbors(\rho, P), \neg(Label4(\rho) \sim Label4(P)) \\
\qquad\qquad\qquad \rightarrow \; Label4(\rho) \sim Label4(P) \\
\qquad \| \qquad \mathbf{true} \;\; \rightarrow \; Label4(P) \\
\,]
\end{array}
$$

Gradually, the $Label4(P)$ transactions associated with the same region are brought into synchrony with each other.

## 4.2. STABLE STATE DETECTION

Having considered four alternative ways of accomplishing the labeling, we turn now to the issue of detecting the completion of the process on a region-by-region basis. We examine four distinct detection paradigms and relate them to the different computing strategies discussed above.

**Coordinated detection.** The first paradigm could be called coordinated detection, a computation which executes a special protocol to detect the desired condition. Termination [128], quiescence [23], and global snapshot algorithms [129] are representative of this paradigm. Algorithms for detecting the termination of a diffusing computation may be adapted to detecting the completion of the region-labeling process. To do this we modify the program given in Figure 3.4 to form the program shown in Figure 4.1. The key modification is the introduction of a tuple $is\_a\_child\_of(\rho, \delta)$ which is used to construct a spanning tree of pixels—a pixel becomes a child of that neighbor whose label it acquired last. During labeling the tree grows from the winning pixel and gradually attaches all pixels in the region to the winning pixel. Trees rooted at losing pixels are eventually destroyed. The growth is coded as part of the $Label5(P)$ transaction. Once the labeling is complete, the tree shrinks to its root which is declared to be the winner. This is carried out by the $Track1(P)$ transaction.

Note that the additional code required to perform the detection involved the modification of the $Label1$ transaction type to form the $Label5$ type. It was not sufficient to merge two separate programs, a labeling and a detection program. We had to introduce some coupling between the two computations. In Swarm such coupling may be easily avoided because of the kinds of queries one can perform against the tuple and transaction spaces.

**program** $RegionLabel5(M, N, Lo, Hi, Intensity :$
$\quad 1 \leq M, 1 \leq N, Lo \leq Hi, Intensity(\rho : Pixel(\rho)),$
$\quad [\forall\, \rho : Pixel(\rho) :: Lo \leq Intensity(\rho) \leq Hi]\,)$
**definitions**
$[\, P, Q, L ::$
$\quad Pixel(P) \;\equiv\; [\exists\, x, y : P = (x, y) :: 1 \leq x \leq N, 1 \leq y \leq M]\,;$
$\quad neighbors(P, Q) \;\equiv$
$\quad\quad Pixel(P), Pixel(Q), P \neq Q,$
$\quad\quad [\exists\, x, y, a, b : P = (x, y), Q = (a, b) :: a - 1 \leq x \leq a + 1, b - 1 \leq y \leq b + 1]\,;$
$\quad R\_neighbors(P, Q) \;\equiv$
$\quad\quad neighbors(P, Q), [\exists\, \iota :: has\_intensity(P, \iota), has\_intensity(Q, \iota)]$
$]$
**tuple types**
$[\, P, Q, L, I : Pixel(P), Pixel(Q) \vee Q = \mathbf{nil}, Pixel(L), Lo \leq I \leq Hi ::$
$\quad has\_label(P, L)\,;\; has\_intensity(P, I)\,;\; is\_a\_child\_of(P, Q)\,;\; wins(P)$
$]$
**transaction types**
$[P : Pixel(P) ::$
$\quad Label5(P) \;\equiv$
$\quad\quad \rho, \lambda 1, \lambda 2 :$
$\quad\quad\quad has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2), R\_neighbors(P, \rho), \lambda 1 > \lambda 2$
$\quad\quad\quad\quad \rightarrow\; has\_label(P, \lambda 2)$
$\quad\quad \|\quad \rho, \delta, \lambda 1, \lambda 2 :$
$\quad\quad\quad has\_label(P, \lambda 1), is\_a\_child\_of(P, \delta)\dagger,$
$\quad\quad\quad has\_label(\rho, \lambda 2), R\_neighbors(P, \rho), \lambda 1 > \lambda 2$
$\quad\quad\quad\quad \rightarrow\; is\_a\_child\_of(P, \rho)$
$\quad\quad \|\quad\quad \mathbf{true} \;\rightarrow\; Label5(P)\,;$
$\quad Track1(P) \;\equiv$
$\quad\quad \lambda, \delta : has\_label(P, \lambda), is\_a\_child\_of(P, \delta)\dagger,$
$\quad\quad\quad [\forall\, \rho : R\_neighbors(P, \rho) :: has\_label(\rho, \lambda), \neg is\_a\_child\_of(\rho, P)\,]$
$\quad\quad\quad\quad \rightarrow\; is\_a\_child\_of(P, \mathbf{nil})$
$\quad\quad \|\quad\quad has\_label(P, P), is\_a\_child\_of(P, \mathbf{nil}) \;\rightarrow\; wins(P)$
$\quad\quad \|\quad \lambda : has\_label(P, \lambda), \neg wins(\lambda) \;\rightarrow\; Track1(P)$
$]$
**initially**
$[P : Pixel(P) ::$
$\quad\quad has\_intensity(P, Intensity(P)), has\_label(P, P), is\_a\_child\_of(P, P),$
$\quad\quad Label5(P), Track1(P)$
$]$
**end**

Figure 4.1: A Region Labeling Program with Termination Detection
*Using a Classic Algorithm to Detect the Termination of a Diffusing Computation*

**Absence of activity.** The three remaining detection paradigms show the different ways decoupling may be accomplished. One strategy we can pursue is to detect the absence of computational activity which may occur once a stable state is established. Of course, this is possible only if the transaction space is dynamic. In the dynamic asynchronous solution presented earlier ($Label2$), when the labeling activity is completed, the winning pixel $P$ is still labeled with its own coordinate and no transactions are attempting to place the label $P$ on any other pixels. Unfortunately this property can also be satisfied by losing pixels. However, a trivial change to $Label2$ allows us to come up with the following elegant solution:

$$
\begin{array}{l}
[\, P : Pixel(P) :: \\
\quad Track2(P) \;\equiv \\
\qquad\qquad alive(P), [\exists\, \rho :: Label2(\rho, P)] \quad \rightarrow\; Track2(P) \\
\qquad \| \quad\; alive(P), [\forall\, \rho :: \neg Label2(\rho, P)] \quad \rightarrow\; wins(P) \\
]
\end{array}
$$

We initially associate a $Track2(P)$ transaction with each pixel $P$ in the image.

The $Track2$ transaction requires that we modify the $RegionLabel2$ program in two ways. Initially an $alive(P)$ tuple exists for each pixel $P$. Transaction $Label2'$ then deletes the tuple $alive(\lambda)$ whenever it relabels any pixel labeled $\lambda$ to a smaller value.

$$
\begin{array}{l}
[\, P, L : Pixel(P), Pixel(L) :: \\
\quad Label2'(P, L) \;\equiv \\
\qquad \cdots \\
\qquad \| \\
\qquad\quad \lambda : has\_label(P, \lambda)\dagger, \lambda > L \\
\qquad\qquad\qquad \rightarrow\; has\_label(P, L), alive(\lambda)\dagger \\
\qquad \| \\
\qquad \cdots \\
]
\end{array}
$$

**Global coordination.** In the next stable state detection mechanism we exploit the global coordination capabilities available in the definition of a synchronic

group. For each region we grow a synchronic group of $Track3$ detectors, one per pixel. The $Track3$ detector transaction for each pixel includes (1) a local subtransaction which fails if the pixel is properly labeled with respect to its neighbors, (2) a second local subtransaction which fails if the $Track3$ transaction for the pixel is in synchrony with the $Track3$ transactions for all neighbors, and (3) a global check which succeeds if all local subtransactions fail and this detector transaction is associated with the winning pixel:

$$
\begin{array}{l}
[P : Pixel(P) :: \\
\quad Track3(P) \ \equiv \\
\qquad\quad \rho, \lambda 1, \lambda 2 : \\
\qquad\qquad has\_label(P, \lambda 1), has\_label(\rho, \lambda 2), R\_neighbors(P, \rho), \lambda 1 > \lambda 2 \\
\qquad\qquad\qquad\quad \rightarrow \ \textbf{skip} \\
\qquad \| \quad \rho : R\_neighbors(P, \rho), \neg(Track3(P) \sim Track3(\rho)) \\
\qquad\qquad\qquad\quad \rightarrow \ Track3(P) \sim Track3(\rho) \\
\qquad \| \qquad \textbf{OR} \quad \rightarrow \ Track3(P) \\
\qquad \| \qquad \textbf{NOR}, has\_label(P, P) \ \rightarrow \ wins(P) \\
]
\end{array}
$$

(In the example above, **skip** is the "no-operation" action.) This approach works by incrementally constructing a synchronic group of $Track3$ transactions for each region of the image; a region's synchronic group encompasses all of the $Track3$ transactions associated with the pixels in the region. When the construction of this group is complete and all pixels in the region are labeled identically, the detector can declare the pixel which is labeled with its own coordinates to be the winner. This approach is compatible with all labeling solutions presented earlier. It does not require that $alive(P)$ tuples be introduced into the $Label2$ computation.

**Global query.** Finally, the most direct solution one can construct is by actually specifying a global query to determine whether the region is or is not labeled:

$$
\begin{array}{l}
[P : Pixel(P) :: \\
\quad Track4(P) \equiv \\
\qquad\qquad has\_label(P,P), \neg wins(P) \rightarrow Track4(P) \\
\qquad \| \qquad has\_label(P,P), \\
\qquad\qquad [\forall\, \rho, \delta : has\_label(\rho,P), R\_neighbors(\rho,\delta) :: has\_label(\delta,P)] \\
\qquad\qquad\qquad \rightarrow wins(P) \\
]
\end{array}
$$

This solution allows labeling and detection to be totally decoupled; it is a direct encoding of the problem statement. To this extent, it represents the ideal programming solution.

## 4.3. PROGRAMMING IMPLICATIONS

The desire to assist programmers in the development and analysis of concurrent computations motivates the shared dataspace model. The model brings together a variety of computing styles within a single unified framework. Programming convenience has been achieved by the provision of powerful queries over both the tuple and transaction spaces and by the ease with which unstructured and unbounded problems may be approached. The opportunities for temporal and spatial decoupling of computations characteristic of shared dataspace languages have been strengthened and enhanced. Development of a proof system for shared dataspace languages (Chapters 6 and 8) and on declarative visualization techniques [45, 46] promise to contribute to increased analyzability of shared dataspace programs.

In Swarm, the *replicated worker* [26] metaphor proposed by Gelernter and his colleagues is refined, acquiring new forms and nuances. First of all, motivated by the fact that reasoning about concurrent computations is done in terms of progress and safety properties, we have been pursuing a programming methodology in which computations are partitioned between progress and detection activities. Progress

and detection programs can be composed either by merging or by introducing some form of coupling (static or dynamic). As made evident in the previous section, the simple merging of independent programs is the preferred method of composition because it enhances program modularity and simplifies reasoning about the

composite program. The use of dynamic coupling as a program composition mechanism remains to be investigated.

The degree of decoupling achievable in Swarm encourages modular programming. As shown in the previous section, labeling and detection activities can be easily composed if we avoid the traditional programming approach illustrated in Figure 4.1. We can actually go one step further: we can construct a speed-up program which can also be merged with the $Label1$, $Label3$, and $Label4$ solutions presented earlier. The speed-up can be carried out by transactions of the following type:

$$
\begin{array}{l}
[ : Pixel(P) :: \\
\quad SpeedUp(P) \equiv \\
\qquad \rho, \lambda : has\_label(P, \rho)\dagger, has\_label(\rho, \lambda) \\
\qquad\qquad\qquad \rightarrow has\_label(P, \lambda), SpeedUp(P) \\
]
\end{array}
$$

This transaction does not work with the $Label2$ program because $SpeedUp$ allows the labeling computation to halt prematurely. This problem can be remedied by modifying $SpeedUp(P)$ to create $Label2(\delta, \lambda)$ transactions for each neighbor $\delta$ of $P$ whenever $P$ is relabeled with $\lambda$.

Transactions participating in progress activities could be called *workers*, while those involved in detection could be called *detectives*. In Swarm, however, workers may be categorized by the way they function and by their level and style of cooperation. Cultivating Gelernter's metaphor, workers in Linda could be called migrant workers because they exist solely to seek out work assignments encoded as tuples in the dataspace. The transactions of the type $Label2(P, L)$ are migrant workers. In contrast, the transactions of type $Label1(P)$ are anchored to a particular pixel serving its labeling needs as a waiter might service a particular table. Through the use of the synchrony relation, a group of workers can be organized

into a community (i.e., a locally synchronous computation) which can evolve and ultimately dissolve on its own. Finally, detectives may be monitoring either the tuple or the transaction spaces seeking to determine the end of a particular phase in the computation.

# 5. A FORMAL MODEL

In this chapter we present an operational, state-transition model for Swarm. This model formalizes the concepts expressed informally in Chapter 3 and lays the foundation for our development of Swarm programming logics in Chapters 6 and 8. The model presented here is similar to the one we presented in [38].

The model represents the execution of a Swarm program as an *infinite* sequence of dataspaces (program states). Terminating computations are modeled as infinite sequences by replicating the final dataspace. The first dataspace in each program execution sequence is one of the valid initial dataspaces of the program. Each successive element consists of the transformed dataspace resulting from the execution of a synchronic group from the preceding element's transaction space. Allowed transitions between dataspaces are specified with a *transition relation*. The choice of the transactions to execute is assumed to satisfy a *fairness* property.

The Swarm model is stated in terms of relationships among several sets of basic entities. **Val** denotes the set of constant *values* used in Swarm programs. In this dissertation we restrict ourselves to integer (the set **Int**) and boolean (the set **Bool**) values. **Nam** is the set from which names of tuple and transaction types are drawn ($\mathbf{Nam} \cap \mathbf{Val} = \emptyset$). In the definition of functions, the domain operator $\rightarrow$ implicitly associates to the right, i.e, $A \rightarrow B \rightarrow C$ means $A \rightarrow (B \rightarrow C)$.

The model also uses a number of operations on sets. For set $S$, $Pow(S)$ denotes the powerset and $Fs(S)$ denotes the set of all finite subsets. We use a three-part notation similar to Swarm's constructor to express set construction and quantified expressions, e.g., $\{n : n > 10 : n\}$ denotes the set of values greater than 10. If $R$

is a binary relation on some set, then $R^\tau$ is the reflexive, transitive closure of the relation. If $S$ is a set, then $S^*$ denotes the set of all finite-length sequences whose elements are drawn from $S$ and $S^\infty$ denotes the set of all infinite sequences. The symbol $\varepsilon$ signifies the empty (zero-length) sequence. Sequence elements are indexed with natural numbers beginning with 0. The notation $s_i$ designates the $i$th element of the sequence $s$; $\#s$ denotes the length of the sequence.

Ignoring the **program** and **definitions** sections (which are syntactic sugar), a Swarm program is modeled as a four-tuple $\langle \mathbf{TP}, \mathbf{TR}, \mathbf{SR}, \mathbf{ID} \rangle$ where:

**TP : Nam** $\rightarrow$ **Val**$^*$ $\rightarrow$ **Bool** is the characteristic function for data tuple types. $\mathbf{TP}(name, values) = true$ if and only if $name(values)$ is a tuple instance allowed by the tuple type declaration in the program's text. A tuple type is the nonempty set of all tuple instances corresponding to one tuple name. The number of tuple types in a program must be finite.

**TR : Nam** $\rightarrow$ **Val**$^*$ $\rightarrow$ **Beh** is the characteristic function for transaction types. $\mathbf{TR}(name, values) \neq \varepsilon$ if and only if $name(values)$ is a transaction instance allowed by the transaction type declaration in the program's text. A transaction type is the nonempty set of all transaction instances corresponding to one transaction name. The number of transaction types must be finite. The sets of names for tuple and transaction types must be disjoint. **Beh** is the set of transaction *behaviors* defined below.

**SR** is the set of valid synchrony relations. Each element of **SR** is a symmetric, irreflexive binary relation on the set of valid transaction instances.

**ID** is the set of valid initial dataspaces; one of these dataspaces is chosen nondeterministically as the first dataspace of an execution sequence.

The data type and transaction type characteristic functions define the sets of all valid instances of tuples ( **TPS** ) and transactions ( **TRS** ):

$$\begin{aligned} \mathbf{TPS} &= \{n, v : n \in \mathbf{Nam} \wedge v \in \mathbf{Val}^* \wedge \mathbf{TP}(n, v) = \mathbf{true} : (n, v)\} \\ \mathbf{TRS} &= \{n, v : n \in \mathbf{Nam} \wedge v \in \mathbf{Val}^* \wedge \mathbf{TR}(n, v) \neq \varepsilon : (n, v)\} \end{aligned}$$

**SR** is a subset of $Pow(\mathbf{TRS} \times \mathbf{TRS})$.

**DS**, the universe of dataspaces (program states), can now be defined as follows:

$$\mathbf{DS} = Fs(\mathbf{TPS}) \times Fs(\mathbf{TRS}) \times \mathbf{SR}$$

Each dataspace consists of a finite tuple space, a finite transaction space, and a synchrony relation. **ID** is a (normally singleton) subset of **DS**.

The set of transaction behaviors **Beh** is a subset of the set of sequences $(\mathbf{L} \cup \mathbf{G})^*$ where:

$\mathbf{L} \cap \mathbf{G} = \emptyset$**.**

$\mathbf{L} \subseteq [\ \mathbf{Bool}^* \rightarrow \mathbf{DS} \rightarrow \mathbf{Val}^* \rightarrow \mathbf{Bool} \times \mathbf{DS} \times \mathbf{DS}\ ]$ is a set of behaviors for subtransactions which involve only ordinary *local* predicates. Each element of **L** maps a dataspace and a set of bindings for subtransaction variables to a query result flag, a group of (tuple and synchrony relation) deletions, and a group of (tuple, transaction, and synchrony relation) insertions. Given a dataspace $d$ and a sequence of values for the subtransaction variables $v$:

$$(\forall\, b : b \in \mathbf{Bool}^* : \mathbf{L}(b, d, v) = \mathbf{L}(\varepsilon, d, v))$$

because the $\mathbf{Bool}^*$ argument is a "dummy" included for compatibility with the set **G**.

$\mathbf{G} \subseteq [\ \mathbf{Bool}^* \rightarrow \mathbf{DS} \rightarrow \mathbf{Val}^* \rightarrow \mathbf{Bool} \times \mathbf{DS} \times \mathbf{DS}\ ]$ is a set of behaviors for subtransactions involving the special *global* predicates **AND**, **OR**, **NAND**, and **NOR** as discussed in Chapter 3. The $\mathbf{Bool}^*$ arguments represent the success and failure results of all the local subtransactions executed in the same step. The function range is interpreted in the same way as in **L**. Given a dataspace $d$, a sequence of local query results $b$, and a sequence of values for the subtransaction variables $v$:

$$(\forall\, b' : b' \text{ is a permutation of } b : \mathbf{G}(b, d, v) = \mathbf{G}(b', d, v))$$

because the global predicates are commutative and associative.

Swarm subtransactions can be translated to **L** and **G** functions in a straightforward manner.

For convenience, we define a number of prefix operators. For any dataspace $d$ in **DS**, $\mathbf{Tp}.d$, $\mathbf{Tr}.d$, and $\mathbf{Sr}.d$ yield, respectively, the tuple space, transaction space, and synchrony relation components of $d$. For example, if $d = (a, b, c)$ is an element of **DS**, then $\mathbf{Tp}.d$ yields the tuple space $a$. For any subtransaction behavior $s$ in $\mathbf{L} \cup \mathbf{G}$, $\mathbf{Q}.s$, $\mathbf{D}.s$, and $\mathbf{I}.s$ are functions which yield the three components of $s$'s range when applied to the same arguments as $s$, i.e., the query result, the dataspace deletions, and the dataspace insertions.

For any dataspace $d$ in **DS**, $(\mathbf{Sr}.d)^\tau$ is an equivalence relation on **TRS**. An equivalence class of the closure is called a *synchrony class*. For a dataspace $d$ having a synchrony class $C$, if $C \cap \mathbf{TR}.d \neq \emptyset$, then $C \cap \mathbf{Tr}.d$, the set of transaction instances in the synchrony class which actually exist in the transaction space, is a *synchronic group* of $d$. To facilitate the modeling of terminating computations, we define $\emptyset$ to be the synchronic group of the empty transaction space.

So far we have modeled the program as a static entity. As noted at the beginning of the chapter, an execution of a program is denoted by an infinite sequence of dataspaces. To be more precise, we define the universe of execution sequences **ES** as follows:

$$\mathbf{ES} \quad = \quad (\mathbf{DS} \times Fs(\mathbf{TRS}))^{\infty}$$

For all $e \in$ **ES** and for all $i \geq 0$, $\mathbf{Ds}.e_i$ is the first component of $e_i$ (the "current" dataspace) and $\mathbf{Sg}.e_i$ is the second (the synchronic group to be executed next).

To define the allowed orders in which dataspaces may be sequenced in an execution of the program, we introduce the *transition relation* **step**. This relation is defined in Figure 5.1. The step relation states that a transition from a dataspace $d$ to a dataspace $d'$ can occur by the execution of a set of transactions $S$ if and only if $S$ is a synchronic group of $d$'s transaction space and $d'$ is a possible result of the synchronous execution of all the subtransactions in $S$ from dataspace $d$. Because there may be several sets of values for the bound variables in a subtransaction that allow the query to succeed on dataspace $d$, the execution of the subtransaction nondeterministically chooses one set. Given a set of values that satisfy the query, the deletion of entities from the tuple space, transaction space, and synchrony relation are "performed before" the insertions of new entities. The subtransactions involving global predicates depend upon the success or failure of the local subtransactions as well as directly upon the dataspace.

Some of the notation in Figure 5.1 needs further explanation. Note in lines 4 and 5 the definition of the functions $v$ and $b$. $v$ maps a subtransaction of $S$ into a sequence of value bindings for its variables, and $b$ maps a subtransaction into a boolean query success flag. In lines 10 and 11 the queries for the global transactions depend upon the elements of $b$ corresponding to local subtransactions.

$$
\begin{array}{l}
(\forall\, d, d', S : d \in \mathbf{DS} \wedge d' \in \mathbf{DS} \wedge S \subseteq \mathbf{TRS} : \\
\quad \mathbf{step}(d, S, d') \equiv \mathit{Synch}(S, d) \wedge \\
\qquad (\exists\, v, b : \\
\qquad\quad v \in [\{t, i : t \in S \wedge 0 \leq i < \#\mathbf{TR}(t) : (t, i)\} \to \mathbf{Val}^*] \wedge \\
\qquad\quad b \in [\{t, i : t \in S \wedge 0 \leq i < \#\mathbf{TR}(t) : (t, i)\} \to \mathbf{Bool}] : \\
\qquad\qquad (\forall\, t, i, \sigma : \mathit{subtrans}(S, t, i, \sigma) \wedge \sigma \in \mathbf{L} : \\
\qquad\qquad\qquad (\mathbf{Q}.\sigma(\varepsilon, d, v(t, i)) \wedge b(t, i)) \\
\qquad\qquad\qquad \vee ((\forall\, x :: \neg\mathbf{Q}.\sigma(\varepsilon, d, x)) \wedge \neg b(t, i))) \\
\qquad\qquad \wedge (\forall\, t, i, \sigma : \mathit{subtrans}(S, t, i, \sigma) \wedge \sigma \in \mathbf{G} : \\
\qquad\qquad\qquad (\mathbf{Q}.\sigma(\mathit{loc}(b, S), d, v(t, i)) \wedge b(t, i)) \\
\qquad\qquad\qquad \vee ((\forall\, x :: \neg\mathbf{Q}.\sigma(\mathit{loc}(b, S), d, x)) \wedge \neg b(t, i))) \\
\qquad\qquad \wedge \mathit{Update}(d, S, d', v, b) \\
\qquad ) \\
)
\end{array}
$$

where

$$
\begin{array}{l}
\quad \mathit{Synch}(S, d) \equiv \\
\qquad (S = \emptyset \wedge \mathbf{Tr}.d = \emptyset) \vee \\
\qquad (S \neq \emptyset \wedge S \subseteq \mathbf{Tr}.d \wedge (\forall\, t, t' : t \in S \wedge t' \in S : (t, t') \in (\mathbf{Sr}.d)^\tau) \wedge \\
\qquad\quad (\forall\, t, x : t \in S \wedge x \in \mathbf{Tr}.d \wedge x \notin S : (t, x) \notin (\mathbf{Sr}.d)^\tau))
\end{array}
$$

and

$$
\quad \mathit{subtrans}(S, t, i, \sigma) \equiv t \in S \ \wedge\ 0 \leq i < \#\mathbf{TR}(t) \ \wedge\ \sigma = (\mathbf{TR}(t))_i
$$

and

$$
\quad \mathit{loc}(b, S) \equiv (\mathrm{SEQ}\, t, i, \sigma : \mathit{subtrans}(S, t, i, \sigma) \wedge \sigma \in \mathbf{L} : b(t, i))
$$

and

$$
\begin{array}{l}
\quad \mathit{Update}(d, S, d', v, b) \equiv \\
\qquad \mathbf{Tp}.d' = (\mathbf{Tp}.d - (\cup\, t, i, \sigma : \mathit{subtrans}(S, t, i, \sigma) \wedge b(t, i) : \\
\qquad\qquad\qquad\qquad \mathbf{Tp}.\mathbf{D}.\sigma(\mathit{loc}(b, S), d, v(t, i)))) \\
\qquad\qquad\quad \cup\, (\cup\, t, i, \sigma : \mathit{subtrans}(S, t, i, \sigma) \wedge b(t, i) : \\
\qquad\qquad\qquad\qquad \mathbf{Tp}.\mathbf{I}.\sigma(\mathit{loc}(b, S), d, v(t, i))) \\
\qquad \wedge\, \mathbf{Tr}.d' = (\mathbf{Tr}.d - S) \\
\qquad\qquad\quad \cup\, (\cup\, t, i, \sigma : \mathit{subtrans}(S, t, i, \sigma) \wedge b(t, i) : \\
\qquad\qquad\qquad\qquad \mathbf{Tr}.\mathbf{I}.\sigma(\mathit{loc}(b, S), d, v(t, i))) \\
\qquad \wedge\, \mathbf{Sr}.d' = (\mathbf{Sr}.d - (\cup\, t, i, \sigma : \mathit{subtrans}(S, t, i, \sigma) \wedge b(t, i) : \\
\qquad\qquad\qquad\qquad \mathbf{Sr}.\mathbf{D}.\sigma(\mathit{loc}(b, S), d, v(t, i)))) \\
\qquad\qquad\quad \cup\, (\cup\, t, i, \sigma : \mathit{subtrans}(S, t, i, \sigma) \wedge b(t, i) : \\
\qquad\qquad\qquad\qquad \mathbf{Sr}.\mathbf{I}.\sigma(\mathit{loc}(b, S), d, v(t, i)))
\end{array}
$$

Figure 5.1: The Transition Relation **step**

In the definition of $loc(b, S)$ the operator SEQ means to concatenate the items in the range of the constructor into a sequence in an arbitrary order. In the definition of the *Update* predicate the subtraction symbol "−" is used to denote the set difference operation.

In Chapter 3 we stated the requirement that the selection of transactions for execution be fair. This fairness constraint can be stated in terms of the execution sequences of this model using the predicate **Fair** defined as follows:

$$
\begin{array}{l}
(\forall\, e : e \in \mathbf{ES} : \\
\quad \mathbf{Fair}(e) \equiv (\forall\, i, t : 0 \leq i \wedge t \in \mathbf{Tr}.\mathbf{Ds}.e_i : \\
\qquad\qquad (\exists\, j : j \geq i : t \in \mathbf{Sg}.e_j \wedge (\forall\, k : i \leq k \leq j : t \in \mathbf{Tr}.\mathbf{Ds}.e_k))))
\end{array}
$$

Informally, an execution sequence is fair if, once a transaction exists in the transaction space, it remains in the space until it is selected for execution and it will be selected for execution within a finite number of steps.

The set of program executions can now be formalized as follows:

$$
\begin{array}{lcl}
\mathbf{Exec} & = & \{e : e \in \mathbf{ES} \wedge \mathbf{Fair}(e) \wedge \mathbf{Ds}.e_0 \in \mathbf{ID} \wedge \\
& & \qquad (\forall\, i : 0 \leq i : \mathbf{step}(\mathbf{Ds}.e_i, \mathbf{Sg}.e_i, \mathbf{Ds}.e_{i+1})) \\
& & \qquad : e\}
\end{array}
$$

This is the set of all execution sequences which begin in a valid initial dataspace, execute a synchronic group of transactions at each computational step, and select transactions for execution in a fair manner.

Using this state-transition model to capture the desired notion of program execution, we have developed programming logics for Swarm (Chapters 6 and 8). These programming logics are similar in style to the logic for UNITY [24]. The above concept of fairness is a central assumption of the logics; it is essential to proofs of progress (liveness) properties of Swarm programs.

# 6. A PROGRAMMING LOGIC

This chapter presents an assertional programming logic for Swarm, building upon the formal model given in Chapter 5. (The logic presented here also appears in [41] and [42].) For simplicity in presentation, we do not consider the synchrony relation feature of Swarm, but we do keep the notation used here compatible with that needed for the full language. The model and logic presented here are generalized to incorporate synchronic groups in Chapter 8.

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

In Chapter 5 we modeled a Swarm program as a set of execution sequences, each of which is infinite and denotes one possible execution of the program. Let $\mathbf{e}$ denote one of these sequences. Each element $\mathbf{e}_i$, $i \geq 0$, of $\mathbf{e}$ is an ordered pair consisting of a program dataspace $\mathbf{Ds.e}_i$ and a set $\mathbf{Sg.e}_i$ containing a single transaction chosen from $\mathbf{Tr.Ds.e}_i$. (If $\mathbf{Tr.Ds.e}_i = \emptyset$, then $\mathbf{Sg.e}_i = \emptyset$.)

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

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

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

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

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 [24]; 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 dissertation we follow the notational conventions for UNITY in [24]. 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 $(p \wedge \neg q)(\mathbf{e}_i)$ to denote the evaluation of the predicate $p \wedge \neg q$ with respect to $\mathbf{Ds.e}_i$. Below we also use the notation $[t]$ to denote the predicate "transaction instance $t$ is in the transaction space."

UNITY assignment statements are deterministic; execution of a statement from a given state will always result in the same next state. This determinism, plus

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

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

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

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

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

Pictorially, we can represent the execution of transaction $t$ where $\{p\}\ t\ \{q\}$ is *true* with the following diagram:

$$\boxed{p \wedge [t]} \xrightarrow{\ t\ } \boxed{\ q\ }$$

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:

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

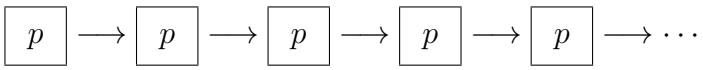
Informally, if $p$ is *true* at some point in the computation and $q$ is not, then, after the next step, $p$ remains *true* or $q$ becomes *true*. (Remember **TRS** is the set of all possible transactions, not a specific transaction space.) In terms of the sequences this rule implies

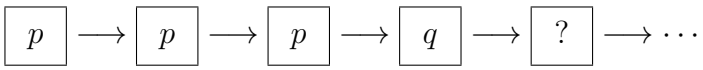
$$(\forall\, e, i : e \in \mathbf{Exec} \wedge 0 \leq i :\ (p \wedge \neg q)(e_i) \Rightarrow (p \vee q)(e_{i+1})\ ).$$

From this we can deduce

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

In other words, either (1) $p \wedge \neg q$ continues to hold indefinitely or (2) $q$ holds eventually and $p$ continues to hold at least until $q$ holds. Thus, where $p$ **unless** $q$ is *true*, we can picture possible execution sequences corresponding to the cases (1) and (2):

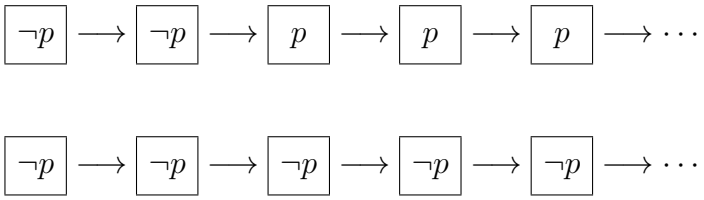



**Stable** and **invariant** properties are fundamental notions of our proof theory. Both can be defined easily as follows:

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

Above *INIT* is a predicate which characterizes the valid initial states of the program. A stable predicate remains *true* once it becomes *true*—although it may never become *true*. Thus, where **stable** $p$ is *true*, we can picture two possible execution sequences:




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

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

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 dissertation, the **ensures** relation is defined as follows:

$$\begin{aligned} p\ \mathbf{ensures}\ q \ \equiv\ & (p\ \mathbf{unless}\ q)\ \wedge \\ & (\exists\, t : t \in \mathbf{TRS} :\ (p \wedge \neg q \Rightarrow [t]) \wedge \{p \wedge \neg q\}\ t\ \{q\}\ ). \end{aligned}$$

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

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

Where $p$ **ensures** $q$ and $\{p \wedge \neg q\}t\{q\}$ are *true*, we can picture a possible execution sequence that establishes $q$ as *true*:

$$\boxed{p \wedge \neg q \wedge [t]} \longrightarrow \boxed{p \wedge \neg q \wedge [t]} \xrightarrow{t} \boxed{q} \longrightarrow \boxed{?} \longrightarrow \cdots$$

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

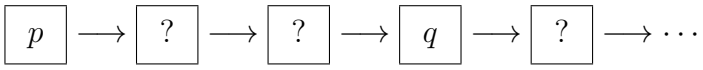
The **leads-to** property, denoted by the symbol $\longmapsto$, is commonly used in Swarm program proofs. The assertion $p \longmapsto q$ is *true* if and only if it can be derived by a finite number of applications of the following inference rules:

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

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

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

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*. Thus, where $p \longmapsto q$ is *true*, we can picture a possible execution sequence:




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 [24] can be proved for Swarm as well. We use the Swarm analogues of various UNITY theorems in the proofs in the next chapter. (The Appendix lists the theorems from [24] used in this dissertation.)

# 7. TWO REGION LABELING EXAMPLES

This chapter applies the programming logic given in Chapter 6 to the verification of two Swarm programs. The programs are two different solutions to the region labeling problem—the *Label*1 and *Label*2 solutions given in Chapter 4. In this chapter we "start from scratch." We formally define the problem and correctness criteria, elaborate the program data structures, and then state the programs and argue that they satisfy the correctness criteria. (The presentation in this chapter is based on [41] and [42].)

## 7.1. THE CORRECTNESS CRITERIA

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

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

neighbors. For convenience, we use the term region to mean a connected equal-intensity region.

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

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

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

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

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

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

We define the predicates $INIT$ and $POST$. $INIT$ characterizes the valid initial states of the computation, $POST$ the desired final state, i.e., the state in which each pixel is labeled with the smallest pixel coordinates in its region. More formally, we define $POST$ 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 and solution strategy are represented faithfully by the program structures,
2. the computation always reaches a state satisfying *POST*,
3. after reaching a state satisfying *POST*, subsequent states continue to satisfy *POST*.

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

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

**Property 2 (Labeling Stability)** **stable** $POST$

In this chapter we specify two different programs to solve the region labeling problem. The programs differ on how the required progress is achieved.

The first program, *RegionLabel*1, uses a static set of transactions. Each transaction is "anchored" to a pixel in the image; the transactions are reinserted 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, *RegionLabel*2, uses a dynamic set of transactions. Each transaction "carries" a pixel's label to a neighbor; the transaction does not reinsert itself. New transactions are created whenever a label changes. As before, a region's winning label eventually propagates throughout the region.

## 7.2. 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 $p$ 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. In this invariant the "#" operator denotes the operation of counting the number of elements satisfying the quantification predicate.

**Property 3 (Intensity Invariant)**

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

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

Only one label (*has_label* tuple) can be associated with each pixel. This label is the coordinates of some pixel within the same region. We also require a pixel's

label to be no larger than the pixel's own coordinates. These three requirements are captured in the Labeling Invariant stated below.

**Property 4 (Labeling Invariant)**

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

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:

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

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,

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

where the "$\Sigma$" and"$-$" operators denote component-wise summation and subtraction of the coordinates.

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

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

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

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

**Property 5 (Region Labeling Invariant)**

**invariant** $\mathbf{0} \leq excess(i) \leq (\Sigma\, p : p \in R(i) : p - w(i)\,)$

**¶ Proof of the Region Labeling Invariant.** This assertion follows from the Labeling Invariant and the definition of *excess*. ∎

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

## 7.3. A STATIC SOLUTION

Chapter 4 presents a region labeling program which expresses a static, asynchronous computation. This program, *RegionLabel*1, is identical to the Swarm program given in Figure 3.4 except that *Label*1 transactions are substituted for the *Label* transactions in the figure. The *Label*1 transaction type is defined as follows:

$$
\begin{array}{l}
[P : Pixel(P) :: \\
\quad Label1(P) \equiv \\
\qquad \rho, \lambda1, \lambda2 : \\
\qquad\quad has\_label(P, \lambda1)\dagger, has\_label(\rho, \lambda2), R\_neighbors(P, \rho), \lambda1 > \lambda2 \\
\qquad\qquad\quad \rightarrow\ has\_label(P, \lambda2) \\
\qquad \| \quad \mathbf{true} \ \rightarrow\ Label1(P) \\
]
\end{array}
$$

The *has_intensity* and *has_label* tuple types and the *Pixel* and *R_neighbors* predicates are defined in Figure 3.4. The predicate $Pixel(P)$ is *true* for every pixel $P$ in the image and *false* otherwise. The predicate $R\_neighbors(P, Q)$ is *true* if and only if pixel $P$ and pixel $Q$ are neighbors in the image (as described previously) and have *equal intensity attributes*.

The *RegionLabel*1 program is initialized as follows:

$$[P : Pixel(P) :: \\ has\_label(P, P), has\_intensity(P, Intensity(P)), Label1(P) \\ ]$$

The **initialization** section establishes the initial dataspace for the execution of the program. Initially, for each pixel $P$ in the image, the dataspace contains a $has\_label(P, P)$ tuple and a $Label1(P)$ transaction. A *has_intensity* tuple also associates the proper intensity value with each pixel.

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 of the Intensity Invariant.** Prove

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

is invariant. Clearly the assertion holds at initialization. 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 of the 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 assertion. No transaction deletes a $has\_label(p, *)$ tuple without inserting a $has\_label(p, *)$ tuple, and vice versa. Thus the number of tuples $has\_label(p, *)$ remains constant.

(2) Consider the second conjunct of the invariant. 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** $p \in R(i) \wedge has\_label(p, w(i))$

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

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

**¶ Proof of Labeling Stability.** We must prove the property **stable** $POST$. The stability of the assertion $excess(i) = \mathbf{0}$, for any region $i$, follows from the Pixel

Label Stability property for each pixel in the region, the **unless** Conjunction Theorem from [24], 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*1 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.

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

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

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

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

**Property 7 (Incremental Labeling)**

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

From the definition of **ensures** given in Chapter 6, we must prove:

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

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

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

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

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 $Label1$ transaction "anchored" on every pixel in the image.

**Property 8 (Static Transaction Space) invariant** $Label1(p)$

**¶ Proof of the Static Transaction Space Invariant.** Initially the property holds. Every transaction always reinserts 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 of Incremental Labeling (exists part).** We must show there is a $t \in$ **TRS** such that

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

where $PRE$ is

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

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

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)** $excess(i) \geq \mathbf{0} \longmapsto excess(i) = \mathbf{0}$

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

**Property 10 (Boundary Invariant)**

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

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

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

**¶ Proof of Regional Progress.** Since $excess(i) = \mathbf{0} \longmapsto excess(i) = \mathbf{0}$ is obvious, only $excess(i) > \mathbf{0} \longmapsto excess(i) = \mathbf{0}$ remains to be proven.

From the Incremental Labeling progress property we know

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

Because of the Boundary Invariant, we also know

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

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

$$\mathbf{0} < excess(i) = k \longmapsto excess(i) < k$$

which can be rewritten as

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

The function $excess(i)$ is a well-founded metric. Thus, using the induction principle for leads-to [24], we conclude the Regional Progress property. ∎

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

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

$$INIT \;\Rightarrow\; (\forall\, i :: excess(i) \geq \mathbf{0}).$$

Hence, for each region $i$,

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

From the Regional Progress property,

$$excess(i) \geq \mathbf{0} \;\longmapsto\; excess(i) = \mathbf{0}.$$

The Labeling Stability property, the Completion Theorem for leads-to [24], 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.

## 7.4. A DYNAMIC SOLUTION

This section states a dynamic solution to the region labeling problem and verifies its correctness. 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 transactions of type *Label*2 replace the transactions of type *Label*1. Transaction type *Label*2 is defined as follows:

$$
\begin{array}{l}
[\, P, L : Pixel(P), Pixel(L) :: \\
\quad Label2(P, L) \equiv \\
\qquad [\, \| \; \delta : P = L, neighbors(P, \delta) :: \\
\qquad\quad \iota : has\_intensity(P, \iota), has\_intensity(\delta, \iota) \\
\qquad\qquad\qquad \rightarrow \; Label2(\delta, P) \\
\qquad ] \\
\qquad \| \\
\qquad\quad \lambda : has\_label(P, \lambda)\dagger, \lambda > L \\
\qquad\qquad\qquad \rightarrow \; has\_label(P, L) \\
\qquad \| \\
\qquad\quad [\, \| \; \delta : \delta \neq L, neighbors(P, \delta) :: \\
\qquad\quad\quad \lambda, \iota : has\_label(P, \lambda), \lambda > L, \\
\qquad\quad\quad\quad has\_intensity(P, \iota), has\_intensity(\delta, \iota) \\
\qquad\quad\quad\quad\quad \rightarrow \; Label2(\delta, L) \\
\qquad\quad ] \\
]
\end{array}
$$

Each $Label2(P, L)$ transaction consists of three groups of subtransactions. In the first and third groups we use the Swarm subtransaction generator feature. 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 of the Intensity Invariant.** Similar to the proof of the Intensity Invariant 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)**

**invariant** $p \in R(i) \wedge Label2(p, l) \Rightarrow l \in R(i) \wedge w(i) \leq l$

**¶ Proof of the Transaction Label Invariant.** The only transactions existing initially are the $Label2(p, p)$ 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 $Label2(p, l)$ can only create transactions of the form $Label2(q, l)$ where $q$ is a neighbor of $p$. Thus the invariant is preserved. ∎

**¶ Proof of the Labeling Invariant.** The proof is similar to the Labeling Invariant proof for *RegionLabel*1 except the Transaction Label Invariant is used to prove the stability of the second and third parts of the invariant. ∎

**¶ Proof of Labeling Stability.** Similar to the proof of the Labeling Stability property for *RegionLabel*1. ∎

So far the proofs of the properties have been almost identical to the corresponding proofs for *RegionLabel*1. The remaining proof obligation is the Labeling Completion progress property. We follow the same methodology as with *RegionLabel*1.

To prove Labeling Completion, we first seek to prove

$$excess(i) \geq \mathbf{0} \;\longmapsto\; excess(i) = \mathbf{0}.$$

However, a stronger formulation of this property may be easier to prove. Initially there does not exist any transaction which can change a label anywhere in the region. The $Label2(p, p)$ transactions initiate the label propagation from each pixel $p$. However, once transaction $Label2(w(i), w(i))$ has executed for each region, there are transactions in the transaction space that decrease $excess(i)$. Moreover, $Label2(w(i), w(i))$ is never regenerated by the computation (because of the $\delta \neq P$ restriction in the transaction definition). Thus we seek to prove the property $\neg Label2(w(i), w(i)) \wedge excess(i) \geq \mathbf{0} \longmapsto excess(i) = \mathbf{0}$. 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 *Label*2 transactions labeling pixels with $w(i)$.

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

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

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

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

**Property 12 (D-Incremental Labeling)**

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

As with $RegionLabel1$, we divide the proof into an **unless**-part and an **exists**-part.

**¶ Proof of the D-Incremental Labeling Property (unless part).** All transactions either leave the labels unchanged or decrease one label by some amount. No transaction creates a $Label2(w(i), w(i))$ 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 $excess(i)$. We evoke the wavefront metaphor described above. The Transaction Wavefront invariant guarantees the existence of $Label2(*, w(i))$ transactions along the boundary of the wavefront.

**Property 13 (Transaction Wavefront)**

$$\textbf{invariant } \neg Label2(w(i), w(i)) \wedge BOUNDARY(i, p, q) \Rightarrow Label2(q, w(i))$$

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

**Property 14 (Startup)**

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

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

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

is *true* for all transactions $t \in$ **TRS**. ($LHS$ and $RHS$ are the left- and right-hand-sides of the **unless** assertion.) The precondition can only be *true* for $p = w(i)$ and $q$ a neighbor of $w(i)$ because of the Winning Label Initiation invariant (proved below). $Label2(w(i), w(i))$ creates $Label2(q, w(i))$, thus establishing the $RHS$ of the **unless** assertion. All other transactions leave $Label2(w(i), w(i))$ *true*. ∎

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

**Property 15 (Winning Label Initiation)**

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

**¶ Proof of Winning Label Initiation.** The invariant is trivially *true* for single pixel regions. Consider multi-pixel regions. Both the $LHS$ and $RHS$ are *true* initially. $Label2(w(i), w(i))$ 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 of Boundary Stability.** We need to prove $(\forall\, t : t \in \textbf{TRS} : \{I\}\ t\ \{I\})$ where $I$ is the implication in the property definition. We need only consider cases in which $I$ is *true* as the precondition.

For pixels $p$ and $q$ which are not equal-intensity neighbors or for single pixel regions, $BOUNDARY(i,p,q)$ 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 of the Transaction Wavefront Invariant.** We must show the assertion $\neg Label2(w(i), w(i)) \wedge BOUNDARY(i,p,q) \Rightarrow Label2(q, w(i))$ is invariant. The property holds initially because $INIT \Rightarrow Label2(w(i), w(i))$. From the Startup property, we know

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

From the Boundary Stability property we know

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

Using the Cancellation Theorem for **unless** [24], we conclude the invariant, i.e.,

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

∎

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

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

where $PRE$ is

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

Because of the Transaction Wavefront invariant, we know $Label2(q, w(i))$ is in the transaction space. Execution of this transaction establishes $excess(i) < k$. ∎

Thus the D-Incremental Labeling property holds for program $RegionLabel2$. 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)**

$$\neg Label2(w(i), w(i)) \wedge excess(i) \geq \mathbf{0} \ \ \longmapsto \ \ excess(i) = \mathbf{0}$$

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)**

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

**¶ Proof of the D-Boundary Invariant.** Similar to the proof of the Boundary Invariant for $RegionLabel1$. ∎

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

$$\neg Label2(w(i), w(i)) \wedge excess(i) > \mathbf{0} \ \ \longmapsto \ \ excess(i) = \mathbf{0}.$$

From the D-Incremental Labeling progress property we know

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

Because of the D-Boundary Invariant we also know

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

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

$$\neg Label2(w(i), w(i)) \ \wedge \ \mathbf{0} < excess(i) = k \ \ \longmapsto \ \ excess(i) < k.$$

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

$$\neg Label2(w(i), w(i)) \wedge excess(i) > \mathbf{0} \wedge excess(i) = k \ \longmapsto$$
$$(\neg Label2(w(i), w(i)) \wedge excess(i) > \mathbf{0} \wedge excess(i) < k) \vee excess(i) = \mathbf{0}.$$

The metric $excess(i)$ is well-founded. 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 of Labeling Completion.** Prove the assertion $INIT \ \longmapsto \ POST$. Clearly,

$$INIT \ \Rightarrow \ (\forall\, i :: excess(i) \geq \mathbf{0} \wedge Label2(w(i), w(i)) \,).$$

Hence, for each region $i$,

$$INIT \quad \textbf{ensures} \quad excess(i) \geq \mathbf{0} \wedge Label2(w(i), w(i)).$$

From the transaction definition, it is easy to see

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

Hence,

$$Label2(w(i), w(i)) \wedge excess(i) \geq \mathbf{0} \ \textbf{ensures}$$
$$excess(i) = \mathbf{0} \vee (\neg Label2(w(i), w(i)) \wedge excess(i) > \mathbf{0}).$$

From the D-Regional Progress property,

$$\neg Label2(w(i), w(i)) \wedge excess(i) \geq \mathbf{0} \quad \longmapsto \quad excess(i) = \mathbf{0}.$$

The Cancellation Theorem for leads-to [24]

$$Label2(w(i), w(i)) \wedge excess(i) \geq \mathbf{0} \quad \longmapsto \quad excess(i) = \mathbf{0}.$$

The Labeling Stability property, the Completion Theorem for leads-to [24], 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:

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

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)**

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

**¶ Proof of the Transaction Flushing Property.** 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

$$\begin{aligned} POST \wedge 0 < (\#\, q, m :: Label2(q, m)) = k \\ \longmapsto (\#\, q, m :: Label2(q, m)) < k. \end{aligned}$$

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

# 8. A GENERALIZED LOGIC

Chapter 6 presented a programming logic for the subset of Swarm without the synchrony relation feature. In this chapter we generalize the logic to handle the synchrony relation.

As with the subset Swarm logic, this logic is built upon the formal model given in Chapter 5, where we model a Swarm dataspace as a triple consisting of a finite tuple space, a finite transaction space, and a synchrony relation. The synchrony relation, which can be modified dynamically during program execution, is a symmetric, irreflexive relation on the set of possible transactions **TRS**. We let **SR** denote the set of all possible synchrony relations. At any point in a computation, a synchronic group is any nonempty intersection of the transaction space with an equivalence class of the reflexive transitive closure of the synchrony relation. The subtransactions making up the transactions of a synchronic group are executed as if they were part of the same transaction.

We define the set **SG** to be the set of all possible synchronic groups of a program. More formally,

$$\mathbf{SG} \;=\; (\cup R, \rho : R \in \mathbf{SR} \wedge \rho \in C(R^{\tau}) : Fs(\rho)\,)$$

where $R^{\tau}$ denotes the reflexive transitive closure of relation $R$, $C(R^{\tau})$ the set of equivalence classes of equivalence relation $R^{\tau}$, and $Fs(\rho)$ the set of all finite subsets of set $\rho$.

The transition relation predicate **step** expresses the semantics of the synchronic groups in **SG**; the values of this predicate are derived from the query and action

parts of the transaction body. The predicate **step**$(d, S, d')$ is *true* if and only if the set of transactions $S$ is a synchronic group of dataspace $d$ and the group's execution can transform dataspace $d$ to a dataspace $d'$.

We model the execution of a Swarm program as a set of execution sequences. Each element of an execution sequence is a pair consisting of a dataspace and a synchronic group of that dataspace. The first dataspace in a sequence is a valid initial dataspace; each successive dataspace is a result of the transformation of the previous element's dataspace by execution of the element's synchronic group. All execution sequences must be fair, i.e., a transaction in the transaction space at any point in the computation must eventually be executed. Terminating computations are extended to infinite sequences by replication of the final dataspace.

The generalized logic we define for Swarm programs with synchronic groups uses the same basic elements as the logic given in Chapter 6: a synchronic group rule; the **unless** relation; **stable**, **invariant**, and **constant** assertions defined in terms of **unless**; the **ensures** relation; and the leads-to relation $\longmapsto$ defined in terms of **ensures.** Here we define the synchronic group rule and the **unless** and **ensures** relations. The other elements of the logic are the same as given in Chapter 6.

We define a basic "Hoare triple" for synchronic groups in terms of the transition relation predicate **step**. For any $S \in \mathbf{SG}$,

$$\{p\}\ S\ \{q\} \quad \equiv \quad (\forall\, d, d' : \mathbf{step}(d, S, d') : p(d) \Rightarrow q(d')\ ).$$

Informally this means that, whenever the precondition $p$ is *true* and $S$ is a synchronic group of the dataspace, all dataspaces which can result from execution of group $S$ satisfy postcondition $q$.

A key difference between this logic and the previous logic is the set over which the properties must be proved. For example, the previous logic required that, in

proof of an **unless** property, an assertion be proved for all possible transactions, i.e., over the set **TRS**. On the other hand, this generalized logic requires the proof of an assertion for all possible synchronic groups, i.e., over the set **SG**.

For the synchronic group logic, we define the logical relation **unless** as follows:

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

If synchronic groups are restricted to single transactions, this definition is the same as the definition given for the earlier subset Swarm logic.

In UNITY-style logics the **ensures** relation is used to define the most basic progress properties of programs. We want to define the **ensures** relation such that $p$ **ensures** $q$ means that "if $p$ is *true* at some point in the computation, $p$ remains *true* as long as $q$ is *false*, and eventually $q$ becomes *true*." [24] The UNITY and subset Swarm logics require "the existence of *one* statement that establishes $q$ starting from a state in which $p \wedge \neg q$ holds." [24] The fairness assumption guarantees that this statement will eventually be executed.

Our first proposal for a definition of **ensures** is the same type of straightforward generalization we used with **unless**. We propose the definition

$$\begin{aligned} p \textbf{ ensures } q \quad \equiv \quad & (p \textbf{ unless } q) \ \wedge \\ & (\exists\, S : S \in \textbf{SG} :\ (p \wedge \neg q \Rightarrow [S]) \wedge \{p \wedge \neg q\}\ S\ \{q\}\ ) \end{aligned}$$

where we interpret $[S]$ to mean "$S$ is a synchronic group of the dataspace." With this definition we require that, when $p \wedge \neg q$ is *true*, there exists a synchronic group $S$ in the dataspace which will establish $q$ when executed from a state in which $p \wedge \neg q$ holds. Because of the fairness criterion, any transaction $t$ in $S$ must eventually be

chosen, and thus $S$ will eventually be executed. (Remember that fairness is stated in terms of transactions not in terms of synchronic groups.)

Because of the dynamic nature of synchronic group structures (i.e., synchronic groups can "go away" without being executed), the above definition seems unnecessarily restrictive. Many leads-to properties are difficult to prove in terms of **ensures** properties defined in this way. A less restrictive definition of **ensures**, which still captures the essence of the UNITY-like notion of basic progress, would be more useful. Thus we define the **ensures** relation in the following way:

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

With this definition we require that, when $p \wedge \neg q$ is *true*, there exists a transaction $t$ in the transaction space such that all synchronic groups which can contain $t$ will establish $q$ when executed from a state in which $p \wedge \neg q$ holds. Because of the fairness criterion, transaction $t$ will eventually be chosen for execution, and hence one of the synchronic groups containing $t$ will be executed. Instead of requiring that we find a single "statement" which will eventually be executed and establish the desired state, this rule requires that a group of "statements" (i.e, set of synchronic groups) be found such that each will establish the desired state and that one of them will eventually be executed. If synchronic groups are restricted to single transactions, this definition is the same as the definition for the subset Swarm logic.

Other than the cases pointed out above (i.e., synchronic group rule, **unless** and **ensures** definitions), this Swarm logic is the same as the subset Swarm logic given in Chapter 6 and as UNITY's logic (without *FP* predicates). The theorems (not

involving *FP*) developed in Chapter 3 of [24] can be proved for the generalized Swarm logic as well. We use the Swarm analogues of various UNITY theorems in the proofs in the next chapter.

# 9. ANOTHER REGION LABELING EXAMPLE

In this chapter we state an *unbounded* region labeling program and verify its correctness using the generalized Swarm programming logic outlined in Chapter 8. This program, a variant of one proposed by Fulcomer and Roman [130], uses the synchronic group feature of Swarm.

## 9.1. A PROGRAM

In this chapter we again address the problem of labeling the equal-intensity regions of a digital image, but we change the problem specification somewhat from that given in Chapters 4 and 7. Here we address an "unbounded" variant of the problem. The image to be processed is arranged on a grid with $M$ rows and an infinite number of columns. We identify the pixels by coordinates with $x$-values 1 or larger and $y$-values in the range 1 through $M$. Although the full image is assumed to extend to the right without bound, the length of each equal-intensity region, i.e., the number of columns intersected by the region, is assumed to be finite and bounded, but of unknown value. For convenience, we let the constant *MaxLen* designate this value for the image to be processed (but do not allow a program to use this constant directly).

We desire a program which labels the regions of unbounded images of this type. The program must not use an unbounded amount of space: the number of tuples and transactions existing at any point during the computation must be bounded above by some constant; the values of all integers used in the program must also be bounded above (and below).

To keep the number of tuples and transactions bounded, we adopt a sliding window metaphor for our solution to the problem. The window is a contiguous group of columns from the image. At any point in the computation, the window contains all pixels currently being processed. The program stores information about these pixels in the dataspace. The computation begins with the leftmost (smallest $x$-coordinate) column of the image in the window. As a computation proceeds, the window expands to the right—the column of the image immediately to the right of the window is inserted into the window when the pixels in that column are "needed." The program needs the new column when some region extends across all columns of the window. The window also contracts from the left—the leftmost column of the window is deleted when all pixels in the column have been "completed." A pixel is complete when all pixels in its region have been labeled with the region's label. (As before, we use the smallest coordinates of a pixel in the region as the region's label.) The window thus slides across the image from left to right; the maximum width of the window is $MaxLen + 1$.

For the size of the numbers used by the program to be bounded, the program cannot use the absolute coordinate system of the full image. Thus, for the pixels in the window, we adopt a new coordinate system—the program addresses pixels relative to the leftmost column of the window. When the program expands the window, all information inserted into the dataspace concerning the new pixels must use window-relative $x$-coordinates. When the program contracts the window, it must also modify all information concerning the pixels in the window to reflect the new coordinate system base.

Figure 9.1 shows a Swarm program, *Unbounded*, which uses this sliding window strategy for labeling the regions of an unbounded region. The program header, definitions, and tuple types are quite similar to those given for *RegionLabel* in

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

Figure 9.1: An Unbounded Region Labeling Program in Swarm

Figure 3.4. Since the number of columns is not fixed, we drop program parameter $N$ and change the definition of predicate $Pixel$ appropriately. We add predicates $on\_left$ and $on\_right$ to identify pixels on the left and right boundaries of the window, respectively. We also add tuple type $final$ to record the $x$-coordinate of the rightmost column of the window.

Program *Unbounded* uses three transaction types—*Label*, *Expand*, and *Contract*. The transactions of type *Label* carry out the labeling of the pixels of the image; transactions of type *Expand* and *Contract* implement the window expansion and contraction operations of the sliding window strategy. Note that the computation begins with the window positioned over a single column—the first column of the image. Figures 9.2, 9.3, and 9.4 show the details of these transaction definitions.

To organize the computation, we take advantage of the synchronic group feature of Swarm. For instance, we use a synchronic group to contract the window. The program creates a *Contract* transaction for each pixel in the window, either at initialization or when a new column is brought into the window by an *Expand* transaction, and links all of these transactions together into a synchronic group. When executed, this group simultaneously decrements the $x$-coordinates for all information recorded for each pixel in the window.

The program also uses synchronic groups of *Label* transactions to carry out the labeling of the regions and to detect when the labeling of a region is complete. The program creates a *Label* transaction for each pixel of the window, either at initialization or when a new column is brought into the window by an *Expand* transaction, and links the transactions for neighboring pixels of the same intensity into the same synchronic group. When one of these *Label* synchronic groups is executed, it either changes the labels of one or more pixels to a lower value or,

$$
\begin{array}{ll}
\multicolumn{2}{l}{Label(P) \equiv} \\
 & \rho, \lambda 1, \lambda 2 : has\_label(P, \lambda 1)\dagger, R\_neighbors(P, \rho), has\_label(\rho, \lambda 2), \lambda 2 < \lambda 1 \\
 & \qquad \longrightarrow\ has\_label(P, \lambda 2) \\
\| & \quad on\_right(P) \longrightarrow \textbf{skip} \\
\| & \quad \textbf{OR} \longrightarrow Label(P) \\
\| & \iota, \lambda : \textbf{NOR}, has\_intensity(P, \iota)\dagger, has\_label(P, \lambda)\dagger \longrightarrow \textbf{skip} \\
\| & \quad \textbf{NOR} \longrightarrow [\, \rho : neighbors(P, \rho) :: (Label(P) \sim Label(\rho))\dagger \,]
\end{array}
$$

Figure 9.2: Unbounded Region Labeling—*Label* Transaction

when it detects that labeling of the region is complete, deletes all information concerning the region from the dataspace.

Now we take a closer look at the details of the transaction definitions. A $Label(P)$ transaction consists of five subtransactions. The first two subtransactions involve local queries, i.e., they do not use the special global predicates **OR**, **NOR**, **AND**, or **NAND**. (See Chapter 3.) If pixel $P$ has a neighbor pixel (in the same region) which has a smaller label, then the first subtransaction relabels $P$ to the neighbor's label. The second subtransaction succeeds when pixel $P$ is on the right boundary of the window. This test is part of the detection strategy for labeling completion. A region is not yet complete if there can exist more pixels in the region that have not yet been input. The remaining three transactions use the special global predicates **OR** and **NOR**. The third subtransaction's query succeeds when *any* of the *local* subtransactions of any transaction *in the synchronic group* succeeds; it reinserts the $Label(P)$ transaction. Gradually the smallest label will propagate throughout the region during the successive executions of the *Label* transactions on a region. The fourth and fifth subtransaction queries can succeed when *none* of the local subtransactions in the synchronic group succeeds. In this

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

Figure 9.3: Unbounded Region Labeling—*Expand* Transaction

case the region has been completely labeled and all information about the region can be deleted. (In its current form this program does not generate any "output.")

Only one *Expand* transaction exists at any point in the computation; it is not in a synchronic group with any other transaction. The *Expand*(*Next*) transaction simulates an input operation—bringing the *Next* column of pixels from the "input" array *Intensity* into the dataspace—and builds appropriate synchronic groups of *Label* and *Contract* transactions. The input of a new column is enabled when there exists some region which spans the width of the window; *Expand* detects this situation by testing for a pixel on the right boundary of the window which is labeled by a pixel's coordinates from the left boundary of the window. (Note that the value of the *Next* argument of the *Expand* transaction grows without bound during a computation. In a sense, *Next* is an "auxiliary variable" needed because

$$
\begin{array}{lll}
\multicolumn{3}{l}{Contract(P) \equiv} \\
 & \iota : & on\_left(P), has\_intensity(P,\iota) \longrightarrow \textbf{skip} \\
\| & \textbf{OR} & \longrightarrow Contract(P) \\
\| & c : \textbf{NOR}, final(c)\dagger & \\
 & & \longrightarrow final(c-1), Contract(P) \\
\| & \iota : \textbf{NOR}, has\_intensity(P,\iota)\dagger & \\
 & & \longrightarrow has\_intensity(P - ONE, \iota) \\
\| & \lambda : \textbf{NOR}, has\_label(P,\lambda)\dagger & \\
 & & \longrightarrow has\_label(P - ONE, \lambda - ONE), Label(P - ONE) \\
\| & [\| \ \rho : neighbors(P,\rho) :: & \\
 & \textbf{NOR}, (Label(P) \sim Label(\rho))\dagger & \\
 & & \longrightarrow Label(P - ONE) \sim Label(\rho - ONE)\,]
\end{array}
$$

Figure 9.4: Unbounded Region Labeling—*Contract* Transaction

the Swarm notation, as defined in Chapter 3, does not have a true input operation. Thus we do not consider it to be a violation of our "bounded values" requirement.)

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

## 9.2. THE CORRECTNESS CRITERIA

In Chapter 7 we identified three key criteria for the correctness of a region labeling program:

- the characteristics of the problem and solution strategy are represented faithfully by the program structures,
- the computation always reaches a state in which every pixel $p$ is labeled with the coordinates of the smallest pixel in the region containing $p$,
- after reaching such a state for a pixel, its label does not change as the computation proceeds.

Because of the unbounded nature of the image for the current problem, we consider the above criteria with respect to finite prefixes of the image. We also impose an additional criterion:

- the program must not use an unbounded amount of resources.

Below we elaborate the first criterion as a set of integrity invariants. The remaining criteria are formalized as the Labeling Completion, Labeling Stability, and Bounded Window properties respectively.

Although at any point in the computation the program only has access to a narrow window imposed upon the image, we find the use of the full unbounded image to be convenient in reasoning about the program. In the statement of properties we use pixel coordinates with respect to the beginning of the full image and identify the regions of the image with integers beginning with 1. We define $R(i)$ to be the set of pixels in region $i$; $w(i)$ to be the "winning" pixel for region $i$—the pixel with smallest coordinates. For convenience, we also define $\mathit{left}(i)$ and $\mathit{right}(i)$ to be the

leftmost and rightmost column numbers of region $i$. Unless otherwise stated we assume that free variables occurring in property assertions are universally quantified implicitly over all valid values of the appropriate type, e.g., $p$ and $q$ over all pixels in the full image, $i$ and $j$ over all regions, $c$ and $d$ over all columns, and $b$ over all intensity values.

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

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

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

**Property 20 (Window Intensity)**

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

**Property 21 (Window Label)**

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

**Property 22 (Window Boundary)**

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

**Property 23 (Window Integrity)**

**invariant** $0 \leq base < last$

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

**Property 24 (Labeling Invariant)**

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

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

**Property 25 (Completion Invariant)**

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

In the above assertion, $col(p)$ denotes the column, or $x$-coordinate, of pixel $p$ and $reg\_gone(i)$ asserts that region $i$ has its final labeling and the associated tuples have been deleted. Formally,

$$\begin{aligned} reg\_gone(i) \;\equiv\; & (\forall\, p : p \in R(i) : pix\_gone(p) \;\wedge \\ & \qquad col(p) < last \wedge pix\_label(p) = w(i)\,) \end{aligned}$$

where $pix\_gone(p) \;\equiv\; (\forall\, b, l :: \neg has\_label(p', l) \wedge \neg has\_intensity(p', b)\,)$.

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

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

**Property 26 (Labeling Completion)**

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

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

**Property 27 (Labeling Stability)**

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

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

**Property 28 (Bounded Window)**

$$\textbf{invariant}\ \ last - base \leq \ MaxLen \ + 1$$

This property does not fully capture all of the "bounded resource" requirement given informally in the previous section, but it does capture the essence of the requirement and is, hence, sufficient for our purposes here.

## 9.3. INVARIANCE PROOFS

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

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

**Property 29 (Synchronic Group Integrity)**

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

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

**Property 30 (Expand Group)**

$$\textbf{invariant} \quad (\#\, c :: Expand(c)) = 1 \wedge Expand(last + 1)$$

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

**Property 31 (Contract Group)**

$$\begin{aligned}\textbf{invariant}\quad & (\forall\, p : base < col(p) \leq last : Contract(p')\,) \;\wedge \\ & (\forall\, p, q : Contract(p') \wedge Contract(q') : \\ & \qquad Contract(p') \approx Contract(q')\,)\end{aligned}$$

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

**Property 32 (Label Group)**

**invariant**

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

The predicate $neighbors(p, q)$ is *true* if and only if $p$ and $q$ are adjacent pixels in the full image.

There are two proof obligations in proof of an invariant: showing that the initial state satisfies the property and showing that all synchronic groups preserve the property. Once the synchronic group invariants above have been proven, they can be used as theorems in the proofs of other properties.

We do not prove all of the invariants here, but most of the proofs are not difficult. Below we sketch three of the more interesting proofs—for the Window Integrity, Completion, and Bounded Window invariants. For these proofs, we assume that

the Window Label, Window Intensity, Labeling, and synchronic group invariants have already been proven.

The invariant proofs below are simplified by the Region Data invariant. This invariant states that, for all unfinished regions, there exist data tuples for all pixels visible in the window.

**Property 33 (Region Data)**

$$\begin{aligned} \textbf{invariant} \quad & (\forall\, p : p \in R(i) \wedge col(p) \leq last : \\ & \qquad (\exists\, b, l :: has\_intensity(p', b) \wedge has\_label(p', l)\,)\,) \\ \equiv\ & \neg reg\_gone(i) \end{aligned}$$

**¶ Proof of the Region Data Invariant.** Let *LHS* and *RHS* refer to the left- and right-hand sides of the equivalence assertion. From the definition of $reg\_gone(i)$, $LHS \Rightarrow \neg reg\_gone(i)$. Thus we must show $\neg LHS \Rightarrow reg\_gone(i)$ is invariant.

The assertion holds initially. Because of the Window Label and Window Intensity invariants, *Contract* groups preserve the property. The *Expand* transaction creates new data tuples for each new pixel made visible in the window, thus preserving the property. Since *last* is nondecreasing, a finished region cannot revert to an unfinished state. A *Label* group which removes the data tuples for a region could violate the invariant. However, the tuples for the pixels in the region can only be removed when the entire region is in the window and all pixels are labeled with $w(i)$; in this case the group establishes $reg\_gone(i)$. ∎

In any computation of the program, the rightmost column of the window is never empty, i.e., there are tuples associated with its pixels. This observation is central to the proof of the Window Integrity invariant. For convenience, we define

$$col\_gone(c) \quad \equiv \quad (\forall\, p : col(p) = c : pix\_gone(p)\,).$$

**¶ Proof of the Window Integrity Invariant.** Prove $0 \leq base < last$ is invariant. Clearly the property holds initially. Since no group can decrease either *base* or *last*, the lower bound obviously holds. Now consider the stability of $base < last$. Note that $base < last$ **unless** $col\_gone(last)$.

$$
\begin{array}{ll}
& col\_gone(last) \\
\equiv & \{ \text{ definition of } col\_gone \ \} \\
& (\forall\, i, p : p \in R(i) \wedge col(p) = last : pix\_gone(p)\,) \\
\equiv & \{ \text{ Region Data invariant } \} \\
& (\forall\, i, p : p \in R(i) \wedge col(p) = last : reg\_gone(i)\,) \\
\Rightarrow & \{ \text{ definition of } reg\_gone \ \} \\
& (\forall\, i, p : p \in R(i) \wedge col(p) = last : col(p) < last\,) \\
\equiv & \textbf{false}.
\end{array}
$$

Thus $base < last$ is invariant. ∎

The *Contract* group increases *base* only when the next column is empty. This is the key observation needed for proving the Completion Invariant.

**¶ Proof of Completion Invariant.** Prove the assertion $p \in R(i) \wedge col(p) \leq base \Rightarrow reg\_gone(i)$ is invariant. Note that

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

is invariant because of the Region Data and Window Integrity invariants. Thus, to prove the Completion Invariant, it is sufficient to show that

$$p \in R(i) \wedge col(p) \leq base \Rightarrow pix\_gone(p)$$

is invariant.

The property holds initially. Since $base < last$ is invariant and $c \leq last \wedge col\_gone(c)$ is stable, only groups which increase *base*, i.e., the *Contract* group,

can violate the property. But *base* can only be incremented from states in which $col\_gone(base+1)$ is *true*. Thus the property is preserved. ∎

The Bounded Window property is a consequence of the bound on the length of individual regions and the enabling condition for expansion of the window, i.e., a label from the leftmost column of the window must have propagated to the rightmost column of the window. For convenience, we define

$$\begin{aligned} enabled(i,c,d) \;\equiv\; & (\exists\, p,q : p \in R(i) \wedge q \in R(i) \;\wedge \\ & \qquad col(p) = d \wedge col(q) = c+1 : has\_label(p',q')\,). \end{aligned}$$

**¶ Proof of Bounded Window Invariant.** Prove $last - base \leq MaxLen + 1$ is invariant. The property holds initially. Clearly

$$last \leq base + MaxLen + 1 \textbf{ unless } (\exists\, i :: enabled(i, base, base + Maxlen + 1)).$$

But

$$\begin{aligned} & enabled(i, base, base + Maxlen + 1) \\ & \quad \equiv \{\text{ definition of } enabled \,\} \\ & \qquad (\exists\, p,q : p \in R(i) \wedge q \in R(i) \wedge col(p) = base + Maxlen + 1 \;\wedge \\ & \qquad\quad col(q) = base + 1 : has\_label(p',q')\,) \\ & \quad \Rightarrow \{\text{ Window Label and Labeling Invariants,} \\ & \qquad\quad \text{maximum region length exceeded } \} \\ & \qquad \textbf{false}. \end{aligned}$$

Thus $last \leq base + MaxLen + 1$ is invariant. ∎

## 9.4. PROGRESS PROOF

The Labeling Completion property asserts that any execution of the unbounded region labeling program will actually label the regions. Specifically, the property

guarantees that any finite prefix of the columns of the full image will eventually be labeled and the associated data tuples deleted. *In terms of the sliding window metaphor, the window will eventually slide to the right of any arbitrary prefix of the full image.* Because of the Completion Invariant, we can conclude that the portion of the image to the left of the window has been labeled as desired. In our programming logic, we state the Label Completion property as

$$INIT \ \wedge C > 0 \longmapsto base \geq C.$$

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

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

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

If the region extends beyond the right boundary, then eventually a *Label* synchronic group will propagate a label from the left boundary across to the right boundary. This enables the input of the next column. Eventually a left-anchored region will

be completely within the window. The same label propagation mechanism will then eventually complete the labeling and remove the region from the dataspace.

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

Properties to prove: Labeling Completion.

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

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

PROPERTY 34. $base = c \longmapsto base = c + 1$

∎

Properties to prove: 34.

**¶ Proof of Property 34.** *For the left boundary of the window to advance past a column, all pixels in that column must be labeled with the winning pixel and deleted from the dataspace. All pixels in the column will eventually be labeled and deleted.*

Consider two cases for the leftmost column of the window: $col\_gone(c+1)$ and $\neg col\_gone(c+1)$. Note that $base = c$ **unless** $base = c + 1$.

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

PROPERTY 35. $base = c \wedge col\_gone(c+1)$ **ensures** $base = c + 1$

(2) Case $\neg col\_gone(c+1)$. First, using the leads-to property below and the **unless** property noted above, apply the Progress-Safety-Progress (PSP) Theorem [24], then apply the Cancellation Theorem for leads-to [24] using case (1).

PROPERTY 36. $base = c \wedge \neg col\_gone(c+1) \longmapsto col\_gone(c+1)$

∎

Properties to prove: 35, 36.

As an **ensures**, property 35 has two proof obligations. Let *LHS* and *RHS* refer to the left- and right-hand sides of the **ensures** assertion. (1) We must show *LHS* **unless** *RHS*. (2) We must also show that, whenever $LHS \wedge \neg RHS$ is *true*, there exists a transaction in the dataspace such that execution of any synchronic group containing that transaction will always establish *RHS* as *true*.

**¶ Proof of Property 35.** *The left boundary of the window will be advanced when the leftmost column has been processed and all pixels removed.*

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

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

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

using the Simple Conjunction Theorem for Unless [24].

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

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

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

Properties to prove: 36.

**¶ Proof of Property 36.** *The unfinished pixels in the leftmost column of the window are in regions which begin in that column and extend to the right. Labeling of these regions will eventually be completed and all pixels removed.*

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

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

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

The Implication Theorem for leads-to [24] allows the "⇒" to be replaced by a "$\longmapsto$".

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

PROPERTY 37. $base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \;\longmapsto\; reg\_gone(i)$

Because of property 37, the definition of $reg\_gone$, and Implication Theorem, we can deduce

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \;\longmapsto\; (\forall\, p : p \in R(i) \wedge col(p) = c+1 : pix\_gone(p)\,).$$

Since $col(p) \leq last \wedge pix\_gone(p)$ is stable and $base < last$ is invariant, we apply the Completion Theorem [24] over the regions intersecting column $c+1$ to deduce

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

Properties to prove: 37.

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

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

where the "$\Sigma$" and"$-$" operators denote component-wise summation and subtraction of the coordinates. We use *excess* as to measure the amount of labeling work remaining to be done on a region.

**¶ Proof of Property 37.** *To show that a left-anchored region eventually is finished, we must prove: (a) if the region extends beyond the right boundary of the window, then eventually all missing columns will be inserted into the window; (b) if the region is completely contained within the window, then it will eventually be labeled completely with the winning pixel and all pixels deleted.*

Prove the assertion

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

Consider three cases for the state of left-anchored regions:

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

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

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

PROPERTY 38.

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) < last \wedge excess(i) = \mathbf{0} \textbf{ ensures } reg\_gone(i)$$

(2) Case $right(i) < last \wedge excess(i) > \mathbf{0}$. First apply the PSP Theorem [24] using the leads-to property below and the conjunction of **unless** and **stable** properties noted above, then apply the Cancellation Theorem for leads-to [24] using case (1).

PROPERTY 39.

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) < last \wedge excess(i) > \mathbf{0} \longmapsto excess(i) = \mathbf{0}$$

(3) Case $right(i) \geq last$. First apply the PSP Theorem [24] using the leads-to property below and the **unless** property noted above, then apply the Cancellation Theorem for leads-to [24] using the disjunction of cases (1) and (2).

PROPERTY 40.

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

∎

Properties to prove: 38, 39, 40.

**¶ Proof of Property 38.** *If the region is labeled with the winning pixel, it will eventually be deleted.*

Prove the assertion

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

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

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

$$\begin{aligned} &\mathit{base} = c \wedge \neg \mathit{col\_gone}(c+1) \wedge \mathit{left}(i) = c+1 \wedge \mathit{right}(i) < \mathit{last} \\ &\qquad \Rightarrow\ \mathit{Label}(w(i)). \end{aligned}$$

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

Properties to prove: 39, 40.

**¶ Proof of Property 39.** *If the region is completely contained within the window, the winning pixel's label is gradually propagated throughout the region.*

Prove the assertion

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

Note that

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) < last \wedge excess(i) > \mathbf{0} \textbf{ unless } excess(i) = \mathbf{0}.$$

First, apply the PSP Theorem [24] using the **ensures** property below and the **unless** property noted above, then apply the Induction Principle for leads-to [24].

PROPERTY 41.

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) < last \wedge \mathbf{0} < excess(i) = k \textbf{ ensures } excess(i) < k$$

∎

Properties to prove: 41, 40.

**¶ Proof of Property 41.** *A Label synchronic group will propagate the label of the winning pixel to other pixels in the region.*

Prove the assertion

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) < last \wedge \mathbf{0} < excess(i) = k \textbf{ ensures } excess(i) < k.$$

Note that the following is invariant:

$$excess(i) > \mathbf{0} \Rightarrow (\exists\, p, q : p \in R(i) \wedge q \in R(i) \wedge neighbors(p,q) : pix\_label(p) < pix\_label(q)\,).$$

(1) **Unless** part. Consider the synchronic groups allowed by the synchronic group invariants. Clearly *Expand* and *Contract* groups and *Label* groups for regions other than $i$ preserve the *LHS* of the **ensures**. Because of the Region Data, Window Label, and "neighbor label" (noted above) invariants, a *Label* synchronic group on region $i$ will decrease $excess(i)$.

(2) Exists part. Because of the Label Group and Window Integrity invariants, $Label(w(i))$ is *true*. Any synchronic group containing this transaction, i.e., a *Label* group on region $i$, will decrease $excess(i)$ as argued above. ∎

Properties to prove: 40.

**¶ Proof of Property 40.** *If the region extends beyond the right boundary of the window, missing columns will gradually be inserted into the window.*

Prove the assertion

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

Note that the following is true:

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

First, apply the PSP Theorem [24] using the leads-to property below and the **unless** property noted above, then apply the Induction Principle for leads-to [24].

Property 42.

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) \geq last \wedge last = d \\ \longmapsto\ last = d+1$$

∎

Properties to prove: 42.

**¶ Proof of Property 42.** *If the region extends beyond the right boundary, then eventually a Label synchronic group will propagate a label from the left boundary across to the right boundary. This enables the input of the next column.*

Prove the assertion:

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

Note that

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

Consider two cases, whether or not insertion of a new column is enabled:

- $(\exists\, j :: enabled(j, c, d)\,)$
- $(\forall\, j :: \neg enabled(j, c, d)\,)$.

(1) Case $(\exists\, j :: enabled(j, c, d)\,)$. First, using the **ensures** below, apply the leads-to disjunction rule over the regions $j$ such that $left(j) = c + 1$. Next apply the Ensures Conjunction Theorem [24] using the **unless** noted above, then weaken the consequence to $last = d + 1$ to deduce this case.

PROPERTY 43.

$$base = c \wedge \neg col\_gone(c+1) \wedge last = d \wedge enabled(j, c, d) \\ \textbf{ensures } last = d+1$$

(2) Case $(\forall\, j \;::\; \neg enabled(j, c, d)\,)$. First apply the PSP Theorem [24] using the leads-to property below and the **unless** property noted above, then apply the Cancellation Theorem for leads-to [24] using case (1).

PROPERTY 44.

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) \geq last \wedge last = d \wedge (\forall\, j \;::\; \neg enabled(j, c, d)\,) \longmapsto (\exists\, j \;::\; enabled(j, c, d)\,)$$

∎

Properties to prove: 43, 44.

**¶ Proof of Property 43.** *If a pixel at the right boundary of the window is labeled with a pixel from the left boundary, then the next column is eventually inserted into the window.*

Prove the assertion:

$$base = c \wedge \neg col\_gone(c+1) \wedge last = d \wedge enabled(j, c, d) \textbf{ ensures } last = d+1.$$

(1) **Unless** part. Consider the synchronic groups allowed by the synchronic group invariants. Since $\neg col\_gone(c+1)$, the *Contract* groups preserve the *LHS* of the **ensures**. Since a *Label* synchronic groups can only decrease labels, pixels coordinates from the leftmost column of the window are less than pixels from all other columns, and regions can only be deleted when they are completely visible in the window, all *Label* groups on all regions preserve the *LHS*. The *Expand* transaction will establish the *RHS* when the *LHS* holds as a precondition.

(2) Exists part. Because of the Expand Group invariant, *Expand(last+1)* is in the dataspace. Because of the Synchronic Group Integrity and Expand Group

invariants, any synchronic group containing this transaction will establish the *RHS* as argued above. ∎

Properties to prove: 44.

The portion of an unfinished region visible in the window may be divided into one or more (maximal) connected subregions by the right boundary of the window. For convenience, we define $Wexcess(i, d)$ to be the total amount the labels on the visible portions of region $i$, when the right window boundary is at column $d$, exceed the desired labeling—all pixels in each visible subregion labeled with the minimum pixel for the subregion. More formally,

$$Wexcess(i, d) \;=\; (\Sigma\, p : p \in R(i) \wedge col(p) \leq d : pix\_label(p) - cw(i, d, p)\,)$$

where $cw(i, d, p)$ is the coordinates of minimum pixel in the subregion of region $i$ containing pixel $p$ when the all pixels to the right of column $d$ are ignored. We use *Wexcess* to measure the amount of labeling work that can be done on the visible subregions without inserting a new column on the right.

**¶ Proof of Property 44.** *Gradually a Label synchronic group will propagate a label from the left boundary across to the right boundary—thus enabling the input of the next column.*

Prove the assertion:

$$\begin{aligned} &base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge \\ &\quad right(i) \geq last \wedge last = d \wedge (\forall\, j :: \neg enabled(j, c, d)\,) \\ &\qquad \longmapsto\; (\exists\, j :: enabled(j, c, d)\,). \end{aligned}$$

Note that

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) \geq last \wedge last = d \wedge$$
$$(\forall\, j :: \neg enabled(j,c,d)\,) \wedge Wexcess(i,d) > \mathbf{0}$$
$$\textbf{unless } (\exists\, j :: enabled(j,c,d)\,).$$

First, apply the PSP Theorem [24] using the **ensures** property below and the **unless** property noted above, then apply the Induction Principle for leads-to [24].

PROPERTY 45.

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) \geq last \wedge last = d \wedge$$
$$(\forall\, j :: \neg enabled(j,c,d)\,) \wedge \mathbf{0} < Wexcess(i,d) = k$$
$$\textbf{ensures } Wexcess(i,d) < k \vee (\exists\, j :: enabled(j,c,d)\,)$$

∎

Properties to prove: 45.

**¶ Proof of Property 45.** *A Label synchronic group will propagate the label of the smallest pixel in a subregion to other pixels in the subregion.*

Prove the assertion:

$$base = c \wedge \neg col\_gone(c+1) \wedge left(i) = c+1 \wedge right(i) \geq last \wedge last = d \wedge$$
$$(\forall\, j :: \neg enabled(j,c,d)\,) \wedge \mathbf{0} < Wexcess(i,d) = k$$
$$\textbf{ensures } Wexcess(i,d) < k \vee (\exists\, j :: enabled(j,c,d)\,).$$

(1) **Unless** part. Consider the synchronic groups allowed by the synchronic group invariants. Since no region is enabled, the *Expand* transaction will preserve the *LHS* of the **ensures**. Since $\neg col\_gone(c+1)$, the *Contract* groups also preserve the *LHS*. Since region $i$ intersects the column $c+1$, no *Label* group on a region

other than $i$ can falsify the *LHS* without "enabling." Any *Label* group on region $i$ will decrease $Wexcess(i, d)$ when the *LHS* holds as a precondition.

(2) Exists part. Because of the Label Group and Window Integrity invariants, there is a *Label* transaction associated with a pixel on each subregion of region $i$. Any synchronic group containing this transaction, i.e., a *Label* group on a subregion of $i$, will establish the *RHS* as argued above. ∎

Properties to prove: none.

# 10. CONCLUSIONS

In this dissertation we introduced the Swarm approach. We informally specified the Swarm notation and presented a formal operational model and an axiomatic programming logic. We briefly explored the implications of Swarm on programming styles by presenting a series of variant solutions to a simple, but non-trivial, problem—labeling the equal intensity regions of a digital image. We also illustrated use of the programming logic by verifying the correctness of three of the region labeling programs.

What has this research contributed to the theory and practice of concurrent programming? In this chapter we address this question.

*The Swarm model reduces both communication and computation to a single mechanism—the atomic execution of transactions.* Most programming language models have separate mechanisms to specify communication and computation, e.g., message send and receive commands for communication and various control constructs for computation. In Swarm a single construct, the transaction, specifies both communication and computation activities. As a result, many of the low-level concerns about synchronization and mutual exclusion are subsumed into the model. Specification and verification of programs are thus simplified.

*The Swarm model unifies key aspects of several paradigms into a single framework.* Many different computational paradigms have been put forward—shared variables, production rules, static and dynamic networks of communicating processes, asynchronous (e.g., CSP) and synchronous (e.g., systolic array and data parallel programs) systems of processes, and so forth. The Swarm mechanisms

enable various approaches to computation to be combined readily within a single program. A Swarm program's general structure is very close to that of production rule systems. The subtransactions within a transaction (synchronic group) are executed synchronously, but the transactions (synchronic groups) themselves are executed asynchronously with respect to each other. Transactions can manipulate the tuple space in many different patterns—treating tuples as variables, as messages, or as parts of (Linda-like) distributed data structures. This multi-paradigmatic nature of Swarm gives a program designer more flexibility than does computational models with fixed paradigms; the designer can chose program and data structures appropriate to the problem and his chosen solution strategy.

*Swarm's synchrony relation is an elegant new language construct for dynamically organizing concurrency.* Sometimes programmers want to organize the concurrency in a program to match the structure of the program's "input" data. These data may be sparse, loosely structured, or unbounded in some manner. Sometimes programmers may also want to alter the structure of the concurrency as a result of a previous subcomputation. The dynamically modifiable synchrony relation, in conjunction with dynamically created transaction instances, provides a simple mechanism for achieving such program structures. For example, in Chapter 4 we show a program which uses the synchrony relation to organize a synchronic group of transactions on each equal-intensity region of an image. In Chapter 9 we use synchronic groups to advantage in structuring the computation for an image which is unbounded in one dimension.

*The Swarm programming logic is the first axiomatic proof system for a shared dataspace "language."* To our knowledge, no axiomatic-style proof systems have been published for Linda, production rule languages, or any other shared dataspace language. Taking advantage of the similarities between the Swarm and UNITY

computational models, we have developed a programming logic for Swarm which is similar in style to that of UNITY. The Swarm logic uses the same logical relations as UNITY, but the definitions of the relations have been generalized to handle the dynamic nature of Swarm, i.e., dynamically created transactions and the synchrony relation. The most challenging problem was the formulation of appropriate definitions for the **ensures** relation.

In our opinion, Swarm is a promising approach to computation. Regardless of Swarm's ultimate place in the pantheon of programming models and languages, we believe the results from this work can impact other research efforts in the programming language and artificial intelligence communities—Linda, UNITY, and rule-based languages in particular. However, more work—both theoretical and pragmatic—is needed before we can assess Swarm's long-term impact on concurrent programming.

# 11. FUTURE WORK

We believe that we have made significant progress toward our research goals, but, like most research efforts, new "questions" arise from the process of "answering" the old questions. In this chapter we examine possible future directions for research. We organize these ideas into four categories: exploration, extension, exploitation, and exportation.

## 11.1. EXPLORATION

Although we have developed several Swarm programs as a part of this and related research, our programming experience with the Swarm notation is still limited. We have found the region labeling problem examined in this dissertation to be an excellent motivating example for the development of Swarm, but we need to study and formulate programming solutions and correctness proofs for many additional problems. The Swarm approach enables many interesting programming styles, e.g., the combination of synchronous and asynchronous processing within the same program, the dynamic organization of concurrency, and the several variations of Gelernter's worker metaphor. These and other programming styles need to be further explored and elaborated. The Swarm programming logic was useful for verification of the region labeling programs; with new examples we can further study its usability and develop new techniques for verification of Swarm programs.

The foundations of the Swarm model and programming logic also need to be explored further. In this research our focus has been on programming. We wanted to define the Swarm notation and propose a means for reasoning about the correctness

of programs written therein. Because of the necessity to limit the scope of this research, we were not able to address the theoretical foundations of Swarm in great detail. The Swarm programming logic should be studied in relation to existing frameworks such as UNITY and temporal logic. The issues of soundness and completeness should be addressed more formally. Since, for the most part, our model and logic treats subtransactions as indivisible units, a finer-grained model, which specifies the semantics of subtransactions in terms of its constituent elements, would provide a better understanding of Swarm and, perhaps, give insight into methods for implementing transactions.

During the design of Swarm, we were faced with decisions concerning which constructs to include and how to formulate those we did include. Several of our choices were rather arbitrary. With the insight we have gained from this research, the exploration of other alternatives may now be profitable. Important alternatives include:

- allowing the explicit deletion of transactions,
- eliminating the automatic deletion of transactions after their execution—particularly in the case of unsuccessful queries,
- making the tuple space a bag (multiset) instead of a set,
- supporting infinite tuple spaces.

## 11.2. EXTENSION

The synchrony relation and the nature of the transaction space enable Swarm programs to organize computations dynamically. However, the notation presented

in this dissertation does not have built-in facilities for abstraction or program composition. It has a relatively flat structure. Perhaps Swarm should be extended to integrate concepts such as the multiple tuple spaces of recent versions of Linda. Or perhaps additional structuring constructs are needed, e.g., sets, subprograms, or arbitrary relations among dataspace elements.

The work of other programming logic researchers should also be adapted to Swarm. For example, the theory of program union and superposition in UNITY [24] and of refinement and projection in Event Predicates [64] should not be difficult to incorporate into a Swarm programming logic.

Another aspect of Swarm that can be extended is its formal model. The model given in Chapter 5 is sequential in the sense that synchronic groups are executed atomically in some sequence. Following the lead of the Action Systems [66] researchers, models incorporating more "realistic" notions of overlapped execution should be formulated and their impact upon proofs studied. These concurrent models can provide insight into various parallel implementation issues.

## 11.3. EXPLOITATION

Full exploitation of the Swarm model as a programming vehicle will, of course, require design and implementation of programming languages based on the model. Several Swarm-related implementation efforts are underway. A colleague has developed a simple transaction system package which executes on the NCUBE hypercube [43, 44]. A version of this package is also being developed for the Apple Macintosh personal computer. Using this package as a base, a subset of Swarm will be implemented. Our group is also investigating the design of a special architecture for execution of Swarm-like programs. These efforts need to be continued.

A more substantial implementation effort should be started. The design of a practical programming language and the engineering of an efficient parallel implementation will require a significant investment. The most important challenge seems to be the development of efficient and reliable algorithms for distributed implementations of the dataspace operations. This work can build on the results from current prototyping efforts, theoretical studies of the concurrent model, and the efforts of other researchers.

In the meantime, a sequential simulator for the Swarm notation would be useful. For example, an interpreter for a Swarm-like language should not be difficult to implement in Prolog. Although such an implementation would not exhibit parallelism, it would enable the example Swarm programs to be executed. Instrumentation of the interpreter could also gather useful statistics concerning execution of Swarm programs.

## 11.4. EXPORTATION

The ideas arising from the Swarm research can potentially be exported to other contexts. From the standpoint of its computational model and programming logic, the UNITY notation is closely related to Swarm. Currently UNITY programs consist of a static set of assignment statements. Perhaps the Swarm programming logic can provide some insight into the formulation of a programming logic for a version of UNITY with a dynamic statement set or with a feature similar to the synchrony relation. Also, as far as we know, axiomatic proof systems do not exist for either Linda or rule-based languages. Again the Swarm programming logic may provide insight into the development of proof systems for these languages, or,

perhaps, the Swarm research will suggest useful modifications of the semantics of such languages.

In research recently begun, the Swarm logic is serving as the basis for a new approach to the visualization of the dynamics of program execution. Roman and Cox are developing a proof-based declarative visualization paradigm [45, 46]. In contrast to the imperative approach to program visualization, in which the programmer must instrument his program with appropriate calls to a visualization package, the declarative approach uses an externally defined mapping from the program's state to a visual display. This research is being conducted in the context of shared dataspace programs. This work seems to have considerable potential.

The Swarm concepts, in conjunction with the declarative visualization techniques, may also be useful in development of the formal methods, algorithms, and methodologies for runtime monitoring of security violations in computing systems. A key part of this work would be the extension of the Swarm model and programming logic to include the semantics of information flow in the shared dataspace [131]. This provides a means for formally stating information flow policies and reasoning about them.

* * * * *

The line of research reported in this dissertation is interesting and, we believe, has made a useful contribution to the field of concurrent programming. However, neither Swarm nor any of the other current approaches solves all of the problems of concurrent programming. Much more remains to be done.

# 12. ACKNOWLEDGMENTS

The design of the Swarm model and notation has been a collaborative effort with my advisor and friend, Dr. Gruia-Catalin Roman. After working together for five years—two years on the shared dataspace paradigm—separating our individual contributions would be impossible. During the past fifteen months, we developed a working pattern and relationship that was quite productive. Catalin paints the sky blue; I find the black storm clouds on the horizon. I design beautiful towers to reach into his blue sky. In his perusal of my plans for a tower, he finds the flaws—maybe I left out the stairs or want to build upon ice which will melt in the sweltering St. Louis summer.

I thank Catalin for having patience with my idiosyncrasies, for encouraging my work, and for taking considerable time from his busy schedule to work closely with me. He has reviewed several drafts of this dissertation; his comments have resulted in many improvements in my presentation of the Swarm concepts. He has been a good "master." I hope that I have been a good "apprentice."

Others have contributed to the group of concepts which we call the shared dataspace paradigm—Mike Ehlers, Wei Chen, Ken Cox, Howard Lykins, and Rose Fulcomer. Mike helped explore the ideas that grew into Swarm. Long discussions with Wei, Ken, and Howard helped me clarify the Swarm programming logic. Rose helped us understand the relationship of Swarm to production rule languages.

I thank the members of my doctoral committee. Dr. Jerome R. Cox, Jr., encouraged my work and, by a favorable assistantship assignment, enabled me to concentrate much of my attention for the past year and a half on this research. By

asking tough questions, Dr. Takayuki Dan Kimura forced me to clarify the muddled statement of my research goals. Dr. Will D. Gillett helped me in my search for a research area and asked insightful questions as a committee member. Dr. Michael I. Miller enthusiastically participated on the committee during my proposal defense. I appreciate Dr. Michael G. Kahn for agreeing to join my committee for the dissertation defense. I thank Dr. Gillett and Dr. Kahn for their many suggestions concerning the text of this dissertation.

I thank Dr. Jan Tijmen Udding for uncovering the beauty of program verification and derivation for me to see. There is still some "hacker" left in me, but now I also appreciate the formal aspects of programming.

I thank Dr. Jay Misra for his review of some of our writing and for his encouragement of our study of UNITY-style programming logics.

My doctoral study would have been difficult without a graduate assistantship. I thank Professor Richard Dammkoehler and the Engineering Computer Laboratory for providing me an interesting place to work for my first four years. I thank Dr. Cox and the Department of Computer Science for the final two years of support. Both made good office and computing facilities available to support my research and writing.

I thank Kevin Fenster, Bill Ross, and the ECL staff for their professional support of the computing facilities I have used in preparing this dissertation and for toleration of a sometimes cranky user. There's probably no worse a "customer" than a former staff member.

I thank the administrative staff: Myrna Harbison, Jean Grothe, Sharon Matlock, Kay Komotos, Peggy Fuller, and Jennifer Sloane. The staff sets the tone for a friendly and productive working environment. They made sure I got my paycheck on time, rushed out my late manuscripts by overnight carrier, prepared my tech

reports, bailed me out when I've been befuddled by the copy machine, and didn't complain too much when I printed long papers on the laserprinter.

I thank my parents, Harold and Mary Cunningham, for creating a home environment that instilled in me the importance of learning, hard work, personal integrity, and religious faith. Their love and support remain very important to me.

I also thank my grandparents—Seibert and Lera Mae Cunningham, John and Pearl Gambill—for their love and nurture. I miss them dearly. I especially thank my Grandfather Cunningham. Even though he didn't finish high school, "Poppy" had a lifetime love of learning. Our wide-ranging discussions during my elementary school years stimulated my desire to know and understand as much as did any of my formal schooling.

Finally, I thank Diana (Glass) Cunningham—my wife of thirteen years. My doctoral program would not have been possible without her support. She loved me at times when I wasn't particularly lovable, lifted me up when I was down, and brought me down to earth when I soared too freely. She deferred progress toward many of her own goals so that I could accomplish a few of mine. I dedicate this work to her.

# 13. APPENDIX

# THEOREMS FROM THE UNITY LOGIC

This appendix lists several theorems for the UNITY programming logic from Chandy and Misra's book [24]. The Swarm analogues of many of these theorems are used in the proofs in this dissertation.

## Theorems about Unless

1. Reflexivity and Antireflexivity

$$p \textbf{ unless } p,$$

$$p \textbf{ unless } \neg p$$

2. Consequence Weakening

$$\frac{p \textbf{ unless } q\,,\ q \Rightarrow r}{p \textbf{ unless } r}$$

3. Conjunction and Disjunction

$$\frac{\begin{array}{l} \\ p \textbf{ unless } q\,, \\ p' \textbf{ unless } q' \end{array}}{\begin{array}{lll} (p \wedge p') \textbf{ unless } (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')\,, & \{\text{conjunction}\} \\ (p \vee p') \textbf{ unless } (\neg p \wedge q') \vee (\neg p' \wedge q) \vee (q \wedge q') & \{\text{disjunction}\} \end{array}}$$

4. Simple Conjunction and Simple Disjunction

$$\frac{\begin{array}{l} p \textbf{ unless } q\,, \\ p' \textbf{ unless } q' \end{array}}{\begin{array}{lll} p \wedge p' \textbf{ unless } q \vee q'\,, & \{\text{simple conjunction}\} \\ p \vee p' \textbf{ unless } q \vee q' & \{\text{simple disjunction}\} \end{array}}$$

5. Cancellation

$$\frac{\begin{array}{l} p \textbf{ unless } q\,, \\ q \textbf{ unless } r \end{array}}{p \vee q \textbf{ unless } r}$$

**Theorems about Ensures**

1. Reflexivity

$$p \textbf{ ensures } p$$

2. Consequence Weakening

$$\frac{p \textbf{ ensures } q \,,\; q \Rightarrow r}{p \;\; \textbf{ensures} \;\; r}$$

3. Impossibility

$$\frac{p \;\; \textbf{ensures} \;\; \textbf{false}}{\neg p}$$

4. Conjunction

$$\frac{\begin{array}{c} p \textbf{ unless } q \,, \\ p' \textbf{ ensures } q' \end{array}}{p \wedge p' \;\; \textbf{ensures} \;\; (p \wedge q') \;\vee\; (p' \wedge q) \;\vee\; (q \wedge q')}$$

5. Disjunction

$$\frac{p \;\; \textbf{ensures} \;\; q}{p \vee r \;\; \textbf{ensures} \;\; q \vee r}$$

**Theorems about Leads-To**

1. Implication Theorem

$$\frac{p \;\Rightarrow\; q}{p \;\longmapsto\; q}$$

2. Impossibility Theorem

$$\frac{p \longmapsto \textbf{ false}}{\neg p}$$

3. General Disjunction Theorem

   For any set $W$:

$$\frac{(\forall\, m : m \in W : p(m) \longmapsto q(m)\,)}{(\exists\, m : m \in W : p(m)) \;\longmapsto\; (\exists\, m : m \in W : q(m))}$$

4. Cancellation Theorem

$$\frac{p \longmapsto q \vee b\,,\; b \longmapsto r}{p \;\longmapsto\; q \vee r}$$

5. Progress-Safety-Progress (PSP) Theorem

$$\frac{p \longmapsto q\;,\; r \;\textbf{unless}\; b}{p \wedge r \;\longmapsto\; (q \wedge r) \vee b}$$

6. Completion Theorem

   For any finite set of predicates $p_i$, $q_i$, $0 \leq i < N$:

$$\frac{(\forall\, i :: p_i \longmapsto q_i \vee b\,)\,,\quad (\forall\, i :: q_i \;\textbf{unless}\; b\,)}{(\wedge\, i :: p_i) \;\longmapsto\; (\wedge\, i :: q_i) \vee b}$$

7. Induction Principle

   For $W$, a well-founded set under the relation $\prec$, and $M$, a function (called a *metric*) from the program states to $W$:

$$\frac{(\forall\, m : m \in W :\; p \wedge M = m \;\longmapsto\; (p \wedge M \prec m) \vee q}{p \;\longmapsto\; q}$$

   **Note:** In applying the induction rule, the set of program states on which the metric $M$ is defined may be limited to those satisfying $p \wedge \neg q$.

# 14. BIBLIOGRAPHY

[1] P. J. Denning. Editorial: Parallel computing and its evolution. *Communications of the ACM*, 29(12):1163–1167, December 1986.

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

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

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

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

[6] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[7] G. D. Plotkin. An operational semantics for CSP. In *Formal Description of Programming Concepts II*, pages 199–225. North-Holland, New York, 1983.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[9] T. Verhoeff. Nondeterminism and divergence created by concealment in CSP. Technical Report 86/06, Eindhoven University of Technology, Department of Mathematics and Computing Science, the Netherlands, September 1986.

[10] K. R. Apt, N. Francez, and W. P. DeRoever. A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.

[11] L. Lamport and F. B. Schneider. The Hoare logic of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984.

[12] N. Soundararajan. Axiomatic semantics of Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, October 1984.

[13] N. Soundararajan. Total correctness of CSP programs. *Acta Informatica*, 23(2), 1986.

[14] M. D. May. Occam. *SIGPLAN Notices*, 18(4):69–79, 1983.

[15] INMOS, Ltd. *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[16] INMOS, Ltd. *Transputer Reference Manual*. INMOS Limited, Bristol, United Kingdom, January 1987.

[17] J. Misra and K. M. Chandy. Termination detection of diffusing computations in Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 4(1):37–43, January 1982.

[18] G.-C. Roman. Specifying software/hardware interactions in distributed systems. In *Proceedings of the 9th International Conference on Software Engineering*, pages 126–139. IEEE, March 1987.

[19] G.-C. Roman, M. E. Ehlers, H. C. Cunningham, and R. H. Lykins. Toward comprehensive specification of distributed systems. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 282–289, September 1987.

[20] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.

[21] M. C. Hull and R. M. McKeag. Communicating Sequential Processes for centralized and distributed operating system design. *ACM Transactions on Programming Languages and Systems*, 6(2):175–191, April 1984.

[22] K. M. Chandy. Concurrent programming for the masses (1984 invited address). In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, pages 1–12, August 1985.

[23] K. M. Chandy and J. Misra. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM Transactions on Programming Languages and Systems*, 8(3):326–343, July 1986.

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

[25] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

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

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

[28] K. A. Frenkel. Introduction: Special issue on parallelism. *Communications of the ACM*, 29(12):1168–1169, December 1986.

[29] N. Carriero and D. Gelernter. The S/net's Linda kernal. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[30] V. Krishnaswamy, S. Ahuja, N. Carriero, and D. Gelernter. The Linda Machine. In S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture, and Technology*, pages 697–717. Plenum Press, 1988. Proceedings of the 1987 Princeton Workshop on Algorithms, Architecture, and Technology Issues for Models of Concurrent Computation.

[31] S. Ahuja, N. J. Carriero, D. H. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37(8):921–929, August 1988.

[32] IEEE. New products. *Computer*, 21(10):74, October 1988. Transputer-based, modular computer fits on a desk, runs at 45 Mflops and 150 VAX MIPS.

[33] N. Carriero and D. Gelernter. Applications experience with Linda. In *Proceedings of the ACM/SIGPLAN PPEALS 1989 (Parallel Programming: Experiences with Applications, Languages and Systems)*, pages 173–187, New

Haven, Connecticut, July 1988. ACM. Also *SIGPLAN Notices* 23(9), September 1988.

[34] 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.

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

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

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

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

[39] 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.

[40] H. C. Cunningham. Concurrent programming in the shared dataspace paradigm. In *Proceedings of the Computer Science Conference.* ACM, February 1989. Research Abstract.

[41] 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.

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

[43] K. C. Cox and G.-C. Roman. A transaction system for the NCUBE. Technical Report WUCS–88–31, Washington University, Department of Computer Science, St. Louis, Missouri, October 1988.

[44] 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*, pages 41–8. IEEE, May 1989.

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

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

[47] R. E. Filman and D. P. Friedman. *Coordinated Computing: Tools and Techniques for Distributed Software*. McGraw-Hill, New York, 1984.

[48] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–44, March 1983.

[49] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[50] J. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[51] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[52] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley, New York, second edition, 1987.

[53] H. Barringer. *A Survey of Verification Techniques for Parallel Programs*. Springer-Verlag, New York, 1985. Lecture Notes in Computer Science #191.

[54] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), June 1981.

[55] E. W. Dijkstra. Co-operating sequential processes. *Programming Languages*, pages 43–112, 1968.

[56] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, New York, 1972.

[57] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[58] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science #16*, pages 89–102. Springer-Verlag, New York, 1974.

[59] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–206, June 1975.

[60] P. Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[61] N. Wirth. Modula: A language for modular multiprogramming. *Software Experience and Practice*, 7:3–35, 1977.

[62] N. Wirth. *Programming in Modula-2*. Springer-Verlag, New York, 1983.

[63] A. U. Shankar and S. S. Lam. Time-dependent distributed systems: Proving safety, liveness, and real-time properties. *Distributed Computing*, 2(2):61–79, August 1987.

[64] 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.

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

[66] 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.

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

[68] W. Hillis. *The Connection Machine*. MIT Press, Cambridge, Massachusetts, 1985.

[69] L. W. Tucker and G. G. Robertson. Architecture and applications of the Connection Machine. *Computer*, 21(8):26–38, August 1988.

[70] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1986.

[71] D. L. Waltz. Applications of the Connection Machine. *Computer*, 20(1):85–97, January 1987.

[72] E. Albert, K. Knobe, J. D. Lukas, and G. L. Steele Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of the ACM/SIGPLAN PPEALS 1989 (Parallel Programming: Experiences with Applications, Languages and Systems*, pages 42–56, New Haven, Connecticut, July 1988. ACM. Also *SIGPLAN Notices* 23(9), September 1988.

[73] M. J. Quinn, P. J. Hatcher, and K. C. Jourdenais. Compiling C* programs for a hypercube multicomputer. In *Proceedings of the ACM/SIGPLAN PPEALS 1989 (Parallel Programming: Experiences with Applications, Languages and Systems*, pages 57–65, New Haven, Connecticut, July 1988. ACM. Also *SIGPLAN Notices* 23(9), September 1988.

[74] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

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

[76] R. B. Kieburtz and A. Silberschatz. Comments on "Communicating Sequential Processes". *ACM Transactions on Programming Languages and Systems*, 1(2):218–225, October 1979.

[77] A. Silberschatz. Extending CSP to allow dynamic resource management. *IEEE Transactions on Software Engineering*, 9(4):527–531, July 1983.

[78] G. Agha and C. Hewitt. Concurrent programming using Actors. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, Cambridge, Massachusetts, 1987.

[79] G. Agha. Foundational issues in concurrent computing. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Proceedings of the ACM SIGPLAN Workshop on Object-based Concurrent Programming*, pages 60–65. ACM, September 1988. Also *SIGPLAN Notices* 24(4), April 1989.

[80] C. Hewitt and P. de Jong. Open systems. In M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *On Conceptual Modelling*, pages 147–164. Springer-Verlag, New York, 1984.

[81] G. Attardi. Concurrent strategy execution in Omega. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 259–276. MIT Press, Cambridge, Massachusetts, 1987.

[82] C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha. An object-oriented concurrent systems architecture. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Proceedings of the ACM SIGPLAN Workshop on Object-based Concurrent Programming*, pages 91–93. ACM, September 1988. Also *SIGPLAN Notices* 24(4), April 1989.

[83] G.-C. Roman, H. C. Cunningham, and M. E. Ehlers. A shared dataspace language supporting large-scale concurrency. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 265–272. IEEE, June 1988.

[84] A. N. Habermann and D. E. Perry. *Ada for Experienced Programmers*. Addison-Wesley, Reading, Massachusetts, 1983.

[85] P. Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):363–396, November 1978.

[86] G. R. Andrews. Synchronizing resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[87] G. R. Andrews. The distributed programming language SR—mechanisms, design, and implementation. *Software Experience and Practice*, 12(8):719–753, 1982.

[88] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[89] W. Weihl and B. Liskov. Implementation of resilent, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

[90] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[91] T. D. Kimura, W. D. Gillett, and J. R. Cox. Abstract Database System (ADS): A data model based on abstraction of symbols. *The Computer Journal*, 28(3):298–308, 1985.

[92] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[93] E. Shapiro, editor. *Concurrent Prolog: Collected Papers (Vols. 1 and 2)*. MIT Press, Cambridge, Massachusetts, 1987.

[94] K. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[95] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts, 1987.

[96] K. L. Clark. PARLOG and its applications. *IEEE Transactions on Software Engineering*, 14(12):1792–1804, December 1988.

[97] K. Ueda. Guarded horn clauses. In *Logic Programming '85*, pages 168–179. Springer-Verlag, New York, 1986. Lecture Notes in Computer Science #221.

[98] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[99] R. H. Halstead, Jr. Parallel symbolic computing. *Computer*, 19(8):35–43, August 1986.

[100] R. Goldman and R. P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, pages 51–59, July 1989.

[101] B. Zorn, K. Ho, J. Larus, L. Semenzato, and P. Hilfinger. Multiprocessing extensions in Spur Lisp. *IEEE Software*, pages 41–49, July 1989.

[102] P. Hudak. Para-functional programming. *Computer*, 19(8):60–70, August 1986.

[103] P. Hudak. Exploring parafunctional programming: Separating the what from the how. *IEEE Software*, pages 54–61, January 1988.

[104] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Architecture*, pages 1–16. Springer-Verlag, New York, 1985.

[105] D. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.

[106] D. A. Padua and M. J. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[107] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 7 October 1985.

[108] B. Alpern and F. B. Schneider. Safety without stuttering. *Information Processing Letters*, 23:177–180, 8 November 1986.

[109] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[110] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[111] E. W. Dijkstra. Postion paper on "fairness". *Software Engineering Notes*, 13(2):18–20, April 1988. EWD1013.

[112] F. B. Schneider and L. Lamport. Another position paper on "fairness". *Software Engineering Notes*, 13(3):18–19, July 1988.

[113] K. M. Chandy and J. Misra. Another view of "fairness". *Software Engineering Notes*, 13(3):20, July 1988.

[114] R. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, Providence, Rhode Island, 1967. Proceedings of the Symposium on Applied Mathematics.

[115] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[116] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[117] R. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[118] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[119] L. Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.

[120] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, chapter 5, pages 215–273. Academic Press, New York, 1981.

[121] L. Lamport. Logical foundation. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, chapter 2.2, pages 19–30. Springer-Verlag, 1985.

[122] A. Pnueli. Applications of temporal logic specifications to the specification and verification of reactive systems: A survey of current trends. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, New York, 1986. Lecture Notes in Computer Science #224.

[123] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

[124] R. Gerth and A. Pnueli. Grounding UNITY. In *Proceedings of the 5th International Workshop on Software Specification and Design*. IEEE, May 1989.

[125] Z. Manna and A. Pnueli. How to cook a temporal logic for your pet language. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 141–154. ACM, 1983.

[126] R. Gerth. Transition logic. In *Proceedings of the 16th Annual Symposium on Theory of Computing (STOC)*, pages 39–51. ACM, 1984.

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

[128] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[129] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[130] R. M. Fulcomer and G. C. Roman. Using Swarm to label an unbounded image. Washington University, Department of Computer Science, St. Louis, Missouri, September 1988. Private communication.

[131] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

# 15. VITA

Biographical data on the author of this dissertation, Mr. H. Conrad Cunningham:

1. Born September 9, 1954, at Pocahontas, Arkansas, to Harold and Mary Cunningham of Success, Arkansas.

2. Attended elementary and secondary school in the Corning School District, Corning, Arkansas, for twelve years. Graduated from Corning High School as Valedictorian in May, 1972.

3. Attended Arkansas State University, Jonesboro, from August, 1972, until May, 1976. Received Bachelor of Science degree with a major in mathematics in May, 1976. Special Distinction Graduate. Received R. E. Lee Wilson Award, Honor Award in Mathematics, Dean Moore Scholastic Award, Who's Who, and Freshman Chemistry Award.

4. Attended Washington University from August, 1976, until May, 1978. Received Master of Science degree with a major in computer science in May, 1978. Held Graduate Teaching Assistantship.

5. DEC-20 Systems Manager with Washington University's Engineering Computer Laboratory from June, 1978, until June, 1980.

6. Senior Software Engineer, Data Systems Division, General Dynamics Corporation, Fort Worth, Texas, from July, 1980, until July, 1983.

7. Attended Washington University from August, 1983, until present date. Held a Graduate Research Assistantship in the Department of Computer Science or the Engineering Computer Laboratory for that period. Also a Seeley-Mudd Fellow.

8. Membership in Professional and Honor Societies: Phi Kappa Phi, Phi Eta Sigma, Kappa Mu Epsilon, ACM, IEEE Computer Society.

August, 1989