Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency

Gruia-Catalin Roman

Department of Computer Science
Washington University
Campus Box 1045, Bryan 509
One Brookings Drive
Saint Louis, Missouri 63130-4899

(314) 889-6190
roman@cs.WUSTL.edu

H. Conrad Cunningham

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

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

June 14, 2003

Abstract

The term *shared dataspace* refers to the general class of models and languages in which the principal means of communication is a common, content-addressable data structure called a dataspace. *Swarm* is a simple language we have used as a vehicle for the investigation of the shared dataspace approach to concurrent computation. It is the first shared dataspace language to have an associated assertional-style proof system. An important feature of *Swarm* is its ability to bring a variety of programming paradigms under a single, unified model. In a series of related examples we explore *Swarm*'s capacity to express shared-variable, message-passing, and rule-based computations; to specify synchronous and asynchronous processing modes; and to accommodate highly dynamic program and data structures. Several illustrations make use of a programming construct unique to *Swarm*, the synchrony relation, and explain how this feature can be used to construct dynamically structured, partially synchronous computations. The paper has three parts: an overview the Swarm programming notation, an examination of Swarm programming strategies via a series of related example programs, and a discussion of the distinctive features of the shared dataspace model. A formal operational model for Swarm is presented in an appendix.

1 INTRODUCTION

Over the last decade concurrency has been one of the most active and prolific areas of research in computer science. The variety of formal models, languages, and algorithms that have been proposed attests to the vitality of the field and to its ability to respond to the underlying technological currents which demand new ways to manage and exploit parallelism. Nevertheless, despite the multiplicity of forms, much of the work on concurrency is aligned with one of two basic paradigms:

- communication via shared variables (e.g., Concurrent Pascal [5], UNITY [10]),
- communication via message passing (e.g., CSP [18] and Actors [1]), including remote operations (e.g., Ada [3] and Argus [23]) which we view as a highly-structured message passing protocol.

The two paradigms differ in the mechanisms they provide for communication among concurrent processes. However, both rely upon the use of names to identify (directly or indirectly) the communicating parties.

Given this state of affairs, one would naturally pose the question: *Is naming fundamental to achieving cooperation among concurrent processes?* We believe the answer to be *no*. To take an example from nature, it is doubtful that bees making up a swarm have individual names, yet, they cooperate effectively in performing highly complex tasks. The key to communication is not naming but, as Lamport [20] points out, the existence of a persistent communication medium (the beehive, the intruding bear, the bees themselves) and a coherent interpretation of the information it encodes.

In the programming language arena, there are numerous instances where data access is primarily by content rather than by name: logic programming, rule-based systems, and database languages. The first concurrent language to make extensive use of a content-addressable communication medium is Linda [2, 7]. In Linda, processes communicate by examining, inserting, and deleting (one at a time) tuples stored in a *tuple space*. Linda's success has been instrumental in the emergence of other languages using a similar communication paradigm. Our own work on language and visualization support for large-scale concurrency led us to propose SDL [27, 31], a language in which processes use powerful transactions to manipulate abstract views of a virtual, content-addressable data structure called the *dataspace*. In related work, Kimura proposed the

Transaction Network [19], a visual language in which the traditional places and transitions appearing in Petri nets have been replaced by databases and transactions, respectively. Recently, we became aware of several other groups working on similar kinds of languages [4, 24].

We use the term *shared dataspace* to refer to the general class of models and languages in which the principal means of communication is a common, content-addressable data structure. Because the investigation of shared dataspace languages is still in its early stages, the body of knowledge accumulated so far is too limited to reach any decisive conclusions about the paradigm's long-term viability. Even so, the Linda-related work, the general trend toward the integration of database concepts into programming languages, and the growing interest in parallel computation among artificial intelligence researchers make us highly optimistic about the future of shared dataspace languages.

Our research group has embarked on a systematic study of the shared dataspace paradigm. The main vehicle for this investigation is a language called *Swarm*. Following the example of Chandy and Misra's UNITY model [10], the design of Swarm takes a minimalist approach; it provides only a small number of constructs which we believe to be at the core of a large class of shared dataspace languages. The shared dataspace encodes the entire state of the computation using a simple, uniform, content-addressable, tuple-like representation. The transaction construct reduces both communication and computation to the notion of an atomic transformation of the dataspace. In addition, the synchrony relation provides a simple mechanism for establishing (at program initialization or dynamically during execution) a synchronous mode of execution for selected portions of the computation—a feature unique to *Swarm*. The result is a very general model of concurrent computation able to express shared-variable, message-passing, and rule-based computations; to specify synchronous and asynchronous processing modes, both statically and dynamically; and to accommodate highly dynamic program and data structures.

Our study of the shared dataspace paradigm has a very broad scope, encompassing the development of formal (operational and axiomatic) semantic models [12, 13, 30], novel programming metaphors specific to the shared dataspace paradigm [29], and new approaches to visualizing concurrent computations [28]. This paper reports the progress toward the development of the Swarm model and a few of its programming implications. The paper has three parts. Section 2 informally overviews the Swarm model and programming notation. To illustrate the kinds of algorithms one can construct in shared dataspace languages, section 3 presents several solutions

to the problem of labeling equal-intensity regions within a digital image. Section 4 highlights the distinctive features of the shared dataspace model. In addition, an appendix presents a formal operational model for Swarm.

2 THE SWARM NOTATION

This section introduces the Swarm model and notation informally. A formal operational model is relegated to the Appendix. UNITY-like assertional programming logics have also been developed for Swarm; these programming logics are described in [12], [13], and [30].

The presentation in this section begins with a formal specification of a region-labeling problem. Next, using a well-known programming notation, Dijkstra's Guarded Commands (GC) [14], the section presents a simple program to solve this problem. The GC program is then transformed, step by step, into a Swarm program. Key aspects of the shared dataspace model are introduced along the way.

Region labeling is a two-dimensional version of the classical leader election problem [11, 22]. Let us consider an image consisting of N rows and M columns. Each discrete point in the image is called a pixel. Each pixel is characterized by its coordinates in the image (i.e., row and column numbers) and intensity (i.e., brightness or color). Two pixels are called neighbors if they have equal intensities and are no further than one row and one column apart. Connected groups of neighboring pixels form equal-intensity regions, henceforth called simply regions. The problem requires all pixels in a region to be labeled with a unique region identifier. If one defines a total ordering on the set of pixel coordinates, the smallest coordinate of any pixel in each region can be used as the region's unique identifier.

Before giving the formal specification for the problem, we need to introduce a few basic concepts and some related notation. A pixel is identified by its coordinates in the image. The predicate $Pixel(P)$ returns *true* if the coordinate P is within the image space, i.e.,

$$Pixel(P) \quad \equiv \quad \langle \exists\, x, y : P = (x, y) :: 1 \leq x \leq N \wedge 1 \leq y \leq M \rangle.$$

The intensity of each pixel is given by the array of constants called *Intensity*. The range of valid intensity values is constrained by two constants, *Hi* and *Lo*, such that

$$\langle \forall\, \rho : Pixel(\rho) :: Lo \leq Intensity(\rho) \leq Hi \,\rangle.$$

The array *PixLabel*, the same size as *Intensity*, associates a label with each pixel. We assume that a label must be a valid coordinate in the image space, i.e.,

$$\langle \forall\, \rho : Pixel(\rho) :: Pixel(PixLabel(\rho)) \,\rangle.$$

For reasons of generality, we do not refer directly to the array *PixLabel*. Instead, we introduce the predicate *is_labeled* which, for the moment, is defined such that

$$is\_labeled(P, L) \quad \equiv \quad (PixLabel(P) = L).$$

Next, we formalize the concept of neighbor by introducing two predicates, *adjacent* and *neighbors*, defined as follows:

$$adjacent(P, Q) \quad \equiv \quad Pixel(P) \wedge Pixel(Q) \wedge (0, 0) < |P - Q| \leq (1, 1)$$

$$neighbors(P, Q) \quad \equiv \quad adjacent(P, Q) \wedge Intensity(P) = Intensity(Q)$$

Here we assume that addition and subtraction of coordinates is done component-wise as follows:

$$(x, y) + (a, b) \quad = \quad (x + a, y + b)$$

$$(x, y) - (a, b) \quad = \quad (x - a, y - b)$$

Further, we assume coordinates are totally ordered by the $<$ relation, such that

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

The predicate *neighbors* defines a symmetric, irreflexive relation on the set of pixels. Thus, the region containing a pixel P, denoted by $R(P)$, can be specified as the set of all pixels related to P via the reflexive transitive closure of this *neighbors* relation[1], i.e.,

[1] We use the notation R^{τ} to denote the reflexive transitive closure of relation R.

$$R(P) \quad = \quad \{\rho : neighbors^{\tau}(P, \rho) :: \rho\}.$$

If the function min is defined for a set of pixels such that $\langle \forall p : p \in S :: min(S) \leq p \rangle$, then $min(R(P))$ designates the smallest pixel coordinate in the region containing pixel P.

Given these definitions, a program is said to solve the region-labeling problem if it satisfies the following properties for every pixel P and intensity ι:

constant $(Intensity(P) = \iota)$**,**

i.e., if the intensity of P is ι, it remains ι throughout the remainder of the computation;

$is\_labeled(P, P)$ **leads_to** $is\_labeled(P, min(R(P)))$**,**

i.e., when starting with an initial labeling where P is labeled with its own coordinate, the computation eventually reaches a state in which P is labeled by the minimum coordinate in the region;

stable $is\_labeled(P, min(R(P)))$**,**

i.e., once P is labeled by the minimum coordinate in the region, its label will never change again.

There is no requirement for the program to terminate. (For formal definitions of **constant**, **leads_to**, and **stable**, the reader should refer to [10].)

One way to solve the region-labeling problem is to associate with each pixel P a process $GLabel(P)$. Its task is to check (forever) whether some pixel Q, adjacent to P, is in the same region as P and has a smaller label. If this is the case, Q's label is adopted also by P.

$GLabel(P)$ ::
do
$\langle \Box\, Q : adjacent(P, Q) ::$
$Intensity(Q) = Intensity(P) \wedge PixLabel(P) > PixLabel(Q)$
$\longrightarrow\ PixLabel(P)\ :=\ PixLabel(Q)\ \rangle$
$\Box$ **true** $\longrightarrow$ **skip**
od

The process $GLabel(P)$ consists of one guarded iteration. The first eight guards are stated compactly in a single statement, parameterized by Q. The value of Q ranges over the eight pixels immediately adjacent to P. These guards implement the labeling activity. The **true** guard ensures

non-termination; otherwise, a pixel having the smallest coordinate in its immediate vicinity could stop the labeling prematurely. As the execution proceeds, the smallest label gradually spreads across the entire region. Nevertheless, all processes continue to execute by taking the **true** guard on each iteration of the loop. Since the pixel coordinates are distinct, the region labels are also distinct.

Of course, for this program to compute the desired result, we must impose some constraints upon its execution. First, the selection of a guard for execution must be fair (i.e., any guard which remains continuously true will eventually be chosen on some iteration of the loop[2]); otherwise, the **true** guard could be always selected and progress may never occur. We also require that any read overlapping a write operation obtains the value either before or after the write. One interesting consequence of this last requirement is that the program defined above executes correctly whether processes execute asynchronously or synchronously! Unfortunately, there is nothing in the GC notation that would allow us to specify such a choice. In the remainder of this section we incrementally transform the GC program into the Swarm program shown in Figure 1.

Data representation. The first transformation involves data representation. In the GC program above, data are represented by a fixed set of shared variables. There are $2{*}N{*}M$ variables logically organized into two arrays: *Intensity* and *PixLabel*. In Swarm, data are represented as data tuples stored in a set called the dataspace. The tuple-based representation is more general than the variable-based approach; it is the first step toward enabling Swarm to accommodate content-based addressing needed to model rule-based computations. Each data tuple consists of a type name and a sequence of values. For the region-labeling program, we introduce two tuple types, *has_intensity* and *has_label*, corresponding to the two arrays. Each array entry is represented by a sequence consisting of the entry's index and value. The values are restricted to valid intensities and labels, respectively. In Swarm, the following tuple type declaration states this restriction:

$$
\begin{array}{l}
[P, L, I : Pixel(P), Pixel(L), Lo \leq I \leq Hi :: \\
\quad has\_intensity(P, I); \\
\quad has\_label(P, L) \\
]
\end{array}
$$

In the Swarm syntax, a comma which separates two logical expressions has the same meaning as the $\wedge$ (logical **and**) operator.

[2]This type of fairness is called weak fairness [16] or justice [21].

The correctness requirements remain essentially the same as for the GC program. The definitions for *is_labeled*, *adjacent*, *neighbors*, and R require only trivial changes to accommodate the new representation. However, because in the Swarm representation more than one intensity or label value could be associated with a pixel, we add a new correctness requirement—an invariant requiring that each pixel have precisely one intensity and one label. As this invariant makes clear, the two tuple types implement the two arrays from the GC program.

Computation/communication. In Swarm, both computation and communication are reduced to atomic transformations of the dataspace. These transformations result from the execution of transactions. A transaction execution involves two steps. (1) A query over the dataspace is evaluated; if successful, values are bound to the transaction's local variables. (2) Successful queries may cause deletions from and insertions into the dataspace. Since multiple sets of dataspace entities may satisfy the same query, the query evaluation manifests nondeterminism. However, once the variables are bound, the dataspace changes are fully specified and deterministic.

Each guarded command in *GLabel* has a simple, direct representation in Swarm. For each Q adjacent to P, one can create a transaction such as the following:

$$\lambda 1, \lambda 2 : neighbors(P, Q), has\_label(P, \lambda 1), has\_label(Q, \lambda 2), \lambda 1 > \lambda 2$$
$$\rightarrow has\_label(P, \lambda 1)\dagger, has\_label(P, \lambda 2)$$

In this transaction, a guard of the **do** is replaced by a query; the assignment is simulated by the deletion (indicated by the † symbol) of the old label for P followed by the insertion of a new label for P.

A more interesting alternative might be to allow Q to be bound in the query as well. Thus eight of the alternatives of the **do** command can be replaced by the single transaction:

$$\rho, \lambda 1, \lambda 2 : neighbors(P, \rho), has\_label(P, \lambda 1), has\_label(\rho, \lambda 2), \lambda 1 > \lambda 2$$
$$\rightarrow has\_label(P, \lambda 1)\dagger, has\_label(P, \lambda 2)$$

The query part searches the dataspace for any neighbor ρ of P which has a smaller label; the action part remains the same. The nondeterminism in this pattern match is substituted for the nondeterminism in transaction selection used earlier to simulate the nondeterminism in the guard selection of *GLabel*. However, there is one important difference. For the GC **do**, we assumed that a guard is selected for execution fairly, but we cannot assume the same for the dataspace pattern match. The Swarm design does not impose any fairness constraint upon the selection of data to satisfy queries. Although in this program the lack of data selection fairness is not a problem, in others it may cause difficulties.

Process representation. In Swarm there is no concept of process and there are no sequential programming constructs. Transaction types and transaction instances are available instead. A transaction specifies an atomic transformation of the dataspace. Transaction instances appear in the dataspace as tuple-like entities consisting of a transaction type name and a sequence of fully instantiated parameters. Transactions present in the dataspace are selected fairly and executed atomically. (A transaction present in the dataspace at any point in the computation will eventually be executed.) Each transaction is automatically removed from the dataspace whenever it is executed, regardless of the success or failure of its query evaluation. Transactions can query the dataspace for the existence of particular tuples and transactions and can insert new tuples and transactions, but cannot delete other transactions from the dataspace. Since Swarm lacks any sequential constructs, sequencing is accomplished by defining continuations in the form of new transactions to be inserted into the dataspace. The counterpart of the process $GLabel(P)$ is a transaction of type *Label* with parameter P, i.e., transaction instance $Label(P)$. Transactions of type *Label* are defined as having the same behavior as the anonymous relabeling transaction introduced earlier, with the added provision that the executing transaction reinserts itself.

$$
\begin{array}{l}
[P : Pixel(P) :: \\
\quad Label(P) \equiv \\
\qquad \rho, \lambda 1, \lambda 2 : neighbors(P, \rho), has\_label(P, \lambda 1), has\_label(\rho, \lambda 2), \lambda 1 > \lambda 2 \\
\qquad\quad \rightarrow has\_label(P, \lambda 1)\dagger, has\_label(P, \lambda 2), Label(P) \\
]
\end{array}
$$

Because insertions follow any deletions, the effect of successfully executing a $Label(P)$ transaction is to relabel pixel P and to reinsert the transaction. What happens, however, if the query fails? The action part is not executed and the transaction is not reinserted! To avoid premature termination, we need something equivalent to the **true** guard used in the GC program:

$$\textbf{true} \longrightarrow Label(P).$$

The static parallel composition operator $\|$ may be used to add this "guard" to the transaction above, giving the following new definition for transactions of type *Label*:

$$
\begin{array}{l}
[P : Pixel(P) :: \\
\quad Label(P) \equiv \\
\qquad \rho, \lambda 1, \lambda 2 : neighbors(P, \rho), has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2), \lambda 1 > \lambda 2 \\
\qquad\quad \rightarrow has\_label(P, \lambda 2) \\
\qquad \| \\
\qquad \textbf{true} \quad \rightarrow Label(P) \\
\quad]
\end{array}
$$

(Note that in redefining $Label(P)$ the dagger has been moved inside the query. This is a shorthand notation useful when the tuples to be deleted appear already in the query.) The query-action components composed by the $\|$ operator are called subtransactions. The entire construct is called a transaction. The interpretation mechanism requires that the evaluations of all subtransaction queries precede any deletions and that all deletions precede any insertions. The result is an interference-free synchronous execution of the subtransactions of a transaction. Such an execution is not generally serializable, i.e., there may not exist any serial execution of the subtransactions resulting in the same dataspace configuration as the parallel execution.

Composing several subtransactions by means of the $\|$ operator is akin to forming a multiple assignment statement from simple assignment statements. Each component involved in the composition is executed every time the compound statement is executed. By contrast, in a guarded selection only one guarded command is executed even if all guards are **true**. The static parallel composition of Swarm also provides a mechanism for several special queries whose success depends upon the success/failure status of the set of "regular" queries (as described above). For instance, in the following redefinition of $Label(P)$, if none of the subtransactions executing regular queries (only one in this case) have succeeded, the second subtransaction's query succeeds and it's action reinserts the transaction.

$$
\begin{array}{l}
Label(P) \equiv \\
\quad \rho, \lambda 1, \lambda 2 : neighbors(P, \rho), has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2), \lambda 1 > \lambda 2 \\
\quad\quad \rightarrow has\_label(P, \lambda 2), Label(P) \\
\| \\
\quad \mathbf{NOR} \rightarrow Label(P)
\end{array}
$$

The built-in special predicates are **OR**, **AND**, **NOR**, and **NAND**, meaning *any*, *all*, *none*, and *not-all*, respectively. These predicates may be combined with regular predicates to form special queries. During transaction execution, the success or failure of predicates in special queries does not affect the evaluation of any other special query in the transaction. The regular predicates in a special query act as filters that may inhibit the actions associated with the respective subtransaction.

Initialization. The last step in our series of transformations is the initialization of the dataspace. For each pixel P, we need to create a data tuple to store the pixel intensity, another data tuple to store the starting label, and a transaction of type *Label*, as shown below.

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

program $RegionLabel(M, N, Lo, Hi, Intensity :$
$\quad 1 \leq M, 1 \leq N, Lo \leq Hi, Intensity(\rho : Pixel(\rho)),$
$\quad [\forall\, \rho : Pixel(\rho) :: Lo \leq Intensity(\rho) \leq Hi]\ )$
definitions
$[\, P, Q, L ::$
$\quad Pixel(P)\ \equiv$
$\quad\quad [\exists\, x, y : P = (x, y) :: 1 \leq x \leq N, 1 \leq y \leq M];$
$\quad adjacent(P, Q)\ \equiv$
$\quad\quad Pixel(P), Pixel(Q), (0, 0) < |P - Q| \leq (1, 1);$
$\quad neighbors(P, Q)\ \equiv$
$\quad\quad adjacent(P, Q), [\exists\, \iota :: has\_intensity(P, \iota), has\_intensity(Q, \iota)]$
$]$
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
$[\, P : Pixel(P) ::$
$\quad Label(P)\ \equiv$
$\quad\quad \rho, \lambda 1, \lambda 2 :$
$\quad\quad\quad neighbors(P, \rho), has\_label(P, \lambda 1)\dagger, has\_label(\rho, \lambda 2), \lambda 1 > \lambda 2$
$\quad\quad\quad\quad \rightarrow\ has\_label(P, \lambda 2), Label(P)$
$\quad\quad \|\quad$ **NOR** $\rightarrow\ Label(P)$
$]$
initialization
$[\, P : Pixel(P) ::$
$\quad has\_label(P, P);$
$\quad has\_intensity(P, Intensity(P));$
$\quad Label(P)$
$]$
end

Figure 1: A Nonterminating Region-Labeling Program in Swarm

The final program is shown in Figure 1. The reader interested in how one might verify this program with respect to the earlier problem specification may turn to [12] or [13].

Synchronous execution. As indicated above, the GC solution would execute correctly under both asynchronous and synchronous assumptions. This is also true of the Swarm version if we define synchronous execution as having the semantics of $\|$: two or more transactions are said to execute synchronously if they complete all regular query evaluations before any deletions and all deletions before any insertions. However, in Swarm, synchrony is not treated as an operational assumption, but as a relation between transactions. If synchronous execution of multiple transactions is desired, one needs to specify the fact explicitly. The information regarding which transactions must be executed synchronously is stored in a third component of the dataspace, the *synchrony relation*. Because the synchrony relation is a part of the dataspace, it can be examined by queries and be modified dynamically by the insertion and deletion of entries in 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. The reflexive transitive closure of the synchrony relation is thus an equivalence relation. When one of the transactions in an equivalence class is chosen for execution, then those members of the class which exist in the transaction space are executed synchronously as if they were a single transaction. This group of related transactions is called a *synchronic group*. (The scope of the special predicates, e.g., **AND**, extends to all regular subtransaction queries in the synchronic group.)

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

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

(where p is a variable and Q is a constant) in the query of some 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. The operation

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

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

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

deletes the relationship. Note that both the base synchrony relation and its closure can be tested in a query, but only the base synchrony relation can be directly modified by the action.

By default, the synchrony relation is empty. Initial couplings can be specified by putting insertion operations into the initialization section. For instance, by adding

$$[P : Pixel(P), Pixel(Q) :: Label(P) \sim Label(Q)\,]$$

to the initialization section of the program shown in Figure 1, we transform it, as discussed later, into a correct synchronous version.

In summary, 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 entities addressed by their values. 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. 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. The execution of a transaction is modeled as a transition between dataspaces. An executing transaction examines the dataspace, then deletes itself from the transaction space and, depending upon the results of the dataspace examination, modifies the dataspace by inserting and deleting tuples and synchrony relation entries and by inserting (but not deleting) other transactions. A Swarm program begins executing from a valid initial dataspace and continues until the transaction space is empty. On each execution step a transaction is chosen nondeterministically from the transaction space along with all other transactions belonging to the same synchronic group. The entire synchronic group is executed. The transaction selection is fair in the sense that each transaction in the transaction space will eventually be chosen.

3 PROGRAMMING METAPHORS

In this section we illustrate the kinds of programming strategies made possible by the proposed Swarm constructs. The region-labeling problem introduced in section 2 serves as a vehicle for explaining the various programming alternatives available in Swarm. Most of the programs in this section are variations of the *RegionLabel* program given in Figure 1. To distinguish among similar transactions in the various solutions, we append unique extensions (numbers and letters) to the base transaction names.

Reasoning about concurrent computations is generally done in terms of progress (i.e., 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.

3.1 Computational Progress Metaphors

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. This mode of computation is characterized by a static set of transactions whose execution must be serializable. In general, the static processing structure is attractive because it often simplifies analysis while the asynchronous execution lends itself to operational models based on the interleaving of atomic actions. Here we present three solutions that employ this mode of computation. Each program is justified by adopting a distinct programming philosophy. Our intent is, in part, to demonstrate Swarm's ability to accommodate the traditional concurrent programming paradigms (i.e., message passing and shared variables) as well as the emerging interest in rule-based computing.

For the first solution we turn to the program shown in Figure 1. The transaction space consists of one transaction per pixel renamed $Label1(P)$ and with the special query **NOR** replaced by **true**:

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

Each $Label1$ transaction is anchored at a pixel; the transaction repeatedly relabels its pixel to smaller labels held by neighbor pixels:

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

Eventually the winning label propagates throughout the entire region. Because each $Label1(P)$ transaction reinserts itself, the transaction space is left unchanged. Also, recall that *has_label* and *has_intensity* tuples implement two fixed-size arrays indexed by P. The program is clearly a shared-variable solution of the type one would implement in UNITY.

By contrast, a message-passing solution would have to permit only the analog of $Label1(P)$, say $Label1a(P)$, to access the intensity and label of P. This requires the introduction of communication channels. We use a tuple of type *channel* associated with a pair of pixels P and Q to transmit the label and intensity of P between $Label1a(P)$ and $Label1a(Q)$. The destination $Label1a(Q)$ acknowledges receipt of the data by deleting the tuple and stops future transmissions between different regions by leaving the tuple untouched. The labels received from channels within the same region are used to update the local label as before.

$$
\begin{array}{l}
[\, P : Pixel(P) :: \\
\quad Label1a(P) \;\equiv \\
\qquad [\!\| \;\; \delta : adjacent(P,\delta) :: \\
\qquad\qquad \lambda 1, \iota 1 : \\
\qquad\qquad\qquad has\_label(P,\lambda 1), has\_intensity(P,\iota 1), (\forall\, \lambda 2, \iota 2 :: \neg channel(P,\delta,\lambda 2,\iota 2)) \\
\qquad\qquad\qquad \rightarrow\; channel(P,\delta,\lambda 1,\iota 1)\;] \\
\qquad \| \\
\qquad\qquad \rho, \lambda 1, \lambda 2, \iota : \\
\qquad\qquad\qquad channel(\rho,P,\lambda 2,\iota)\dagger, has\_label(P,\lambda 1)\dagger, has\_intensity(P,\iota), \lambda 1 > \lambda 2 \\
\qquad\qquad\qquad \rightarrow\; has\_label(P,\lambda 2) \\
\qquad \| \\
\qquad\qquad \rho, \lambda 1, \lambda 2, \iota : \\
\qquad\qquad\qquad channel(\rho,P,\lambda 2,\iota)\dagger, has\_label(P,\lambda 1), has\_intensity(P,\iota), \lambda 1 \leq \lambda 2 \\
\qquad\qquad\qquad \rightarrow\; \textbf{skip} \\
\qquad \| \qquad \textbf{true} \;\; \rightarrow\; Label1(P) \\
\,]
\end{array}
$$

The initial dataspace configuration remains unchanged, except for the transaction renaming, since the channel tuples are created when needed.

Yet another solution can be generated if one approaches the same problem from a rule-based perspective. The dataspace can be viewed as the working memory of an OPS5-like program with transactions functioning as production rules. The first rule is a generalization of the $Label1$ transaction type. It states that if two pixels, ρ and δ, are in the same region and have different labels, the smaller of the two labels should be applied to both pixels. The result is a transaction which, instead of being anchored at some pixel P, may be applied to any pixel pair satisfying the query:

$$
\begin{array}{l}
[\, :: \\
\quad Label1b() \;\equiv \\
\qquad \rho, \delta, \lambda 1, \lambda 2 : has\_label(\delta,\lambda 1)\dagger, neighbors(\delta,\rho), has\_label(\rho,\lambda 2), \lambda 1 > \lambda 2 \\
\qquad\qquad \rightarrow\; has\_label(\delta,\lambda 2) \\
\qquad \| \qquad \textbf{true} \;\; \rightarrow\; Label1b() \\
\,]
\end{array}
$$

By exploiting transitivity, a second rule could be added. It states that if $\rho 1$ has the label $\rho 2$ and $\rho 2$ has the label $\rho 3$, then we can change the label of $\rho 1$ to be $\rho 3$. This is possible because all three pixels must be in the same region and $\rho 3$ is the smallest of the three:

$$
\begin{array}{l}
[:: \\
\quad Label1c() \;\equiv \\
\qquad \rho 1, \rho 2, \rho 3 : has\_label(\rho 1,\rho 2)\dagger,\; has\_label(\rho 2,\rho 3),\; \rho 1 > \rho 2, \rho 2 > \rho 3 \\
\qquad\qquad \rightarrow\; has\_label(\rho 1,\rho 3) \\
\qquad \| \qquad \textbf{true} \;\; \rightarrow\; Label1c() \\
\,]
\end{array}
$$

Clearly, the last two transactions introduced use the pattern matching power available to Swarm queries. By contrast, the transactions discussed earlier use the tuples in highly structured ways which may be formally characterized. This suggests a general approach to the development of multiparadigm concurrent software. Starting with a validated Swarm solution, through successive refinements one can restructure the solution so that different groups of transactions exhibit data access patterns specific to one of the three paradigms above. The result is a program whose components can be readily mapped to a heterogeneous group of languages and machines.

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:

$$\begin{array}{l}
[\, P, L : Pixel(P), Pixel(L) :: \\
\quad Label2(P, L) \ \equiv \\
\qquad [\!\| \ \delta : P = L, adjacent(P, \delta) :: \\
\qquad\quad \iota : has_intensity(P, \iota), has_intensity(\delta, \iota) \\
\qquad\qquad\quad \rightarrow \ Label2(\delta, P) \\
\qquad] \\
\qquad \| \\
\qquad\quad \lambda : has_label(P, \lambda)\dagger, \lambda > L \\
\qquad\qquad\quad \rightarrow \ has_label(P, L) \\
\qquad \| \\
\qquad\quad [\!\| \ \delta : \delta \neq L, adjacent(P, \delta) :: \\
\qquad\quad \lambda, \iota : has_intensity(P, \iota), has_intensity(\delta, \iota), has_label(P, \lambda), \lambda > L \\
\qquad\qquad\quad \rightarrow \ Label2(\delta, L) \\
\qquad\quad] \\
]
\end{array}$$

Each $Label2(P, L)$ transaction consists of three groups of subtransactions. For $P = L$, the first group of subtransactions includes a subtransaction for each pixel δ such that $adjacent(P, \delta)$; 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. This kind of solution maps straightforwardly into a network architecture which supports dynamic process creation and migration. One can think of $Label2(P, L)$ as a having identity L and executing at location P. The migratory pattern for these transactions is highly structured and predefined. This need not be the case. Less differentiated transactions could be designed to move across the dataspace in search of work. This kind of programming style has been used very effectively in Linda. In the following example we extend this metaphor one step further by initially creating a single transaction, $Label2a((1,1))$, which, upon succeeding in relabeling one pixel, clones itself. Each continuation, in turn, seeks work and clones itself when successful:

$$
\begin{array}{l}
[P : Pixel(P) : \\
\quad Label2a(P) \ \equiv \\
\qquad \rho, \delta, \lambda 1, \lambda 2 : has\_label(\delta, \lambda 1)\dagger, neighbors(\delta, \rho), has\_label(\rho, \lambda 2), \lambda 1 > \lambda 2 \\
\qquad\qquad \rightarrow \ has\_label(\delta, \lambda 2), Label2a(\rho), Label2a(\delta) \\
]
\end{array}
$$

Note that the parameter P has no impact on which data are used by the transaction. It simply provides a distinct identity and places some control on the potential explosion in concurrency caused by successfully executing transactions. Of course, when the labeling is completed, all transactions eventually fail, causing the transaction space to become empty.

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

$$
\begin{array}{l}
[\ :: \\
\quad Label3(\,) \ \equiv \\
\qquad [\!\| \ \rho : Pixel(\rho) :: \\
\qquad\quad \delta, \lambda 1, \lambda 2 : neighbors(\rho, \delta), has\_label(\rho, \lambda 1)\dagger, has\_label(\delta, \lambda 2), \lambda 1 > \lambda 2 \\
\qquad\qquad\quad \rightarrow \ has\_label(\rho, \lambda 2) \\
\qquad] \\
\quad \| \qquad \textbf{true} \ \rightarrow \ Label3(\,) \\
]
\end{array}
$$

This kind of solution, typical for many SIMD machines such as the Connection Machine, creates an unnecessary coupling between independent regions of the image. Because the structure of the image varies, one cannot conceive of a transaction which processes a single region, independently of all the other regions.

For static data, Swarm's 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:

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

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:

$$
\begin{array}{l}
[\, P : Pixel(P) :: \\
\quad Label4(P) \ \equiv \\
\qquad \rho, \lambda1, \lambda2 : neighbors(P,\rho), has\_label(P,\lambda1)\dagger, has\_label(\rho,\lambda2), \lambda1 > \lambda2 \\
\qquad\qquad \rightarrow \ has\_label(P,\lambda2) \\
\qquad \| \quad \rho : neighbors(P,\rho), \neg(Label4(P) \sim Label4(\rho)) \\
\qquad\qquad \rightarrow \ Label4(P) \sim Label4(\rho) \\
\qquad \| \qquad \textbf{true} \ \rightarrow \ Label4(P) \\
]
\end{array}
$$

Gradually, the $Label4(P)$ transactions associated with the same region are brought into synchrony with each other. As discussed in more detail in the next section, the computation could be made to terminate when no relabeling takes place anywhere in the region and the synchronic group can not expand any further. To accomplish this we would need to replace the **true** query with the special query **OR**. This solution is particularly interesting because it illustrates the kind of power and flexibility that can be gained by exploiting the dynamic nature of the synchrony relation.

3.2 Stable State Detection Metaphors

Having considered several 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.

```
program RegionLabel5(M, N, Lo, Hi, Intensity :
        1 ≤ M, 1 ≤ N, Lo ≤ Hi, Intensity(ρ : Pixel(ρ)),
        [∀ ρ : Pixel(ρ) :: Lo ≤ Intensity(ρ) ≤ Hi] )
definitions
  [ P, Q, L ::
    Pixel(P) ≡ [∃ x, y : P = (x, y) :: 1 ≤ x ≤ N, 1 ≤ y ≤ M] ;
    adjacent(P, Q) ≡
      Pixel(P), Pixel(Q), (0, 0) < |P − Q| ≤ (1, 1);
    neighbors(P, Q) ≡
      adjacent(P, Q), [∃ ι :: has_intensity(P, ι), has_intensity(Q, ι)]
  ]
tuple types
  [ P, Q, L, I : Pixel(P), Pixel(Q) ∨ Q = nil, Pixel(L), Lo ≤ I ≤ Hi ::
      has_label(P, L) ; has_intensity(P, I) ; is_a_child_of(P, Q) ; wins(P)
  ]
transaction types
  [P : Pixel(P) ::
    Label5(P) ≡
        ρ, λ1, λ2 :
          neighbors(P, ρ), has_label(P, λ1)†, has_label(ρ, λ2), λ1 > λ2
              → has_label(P, λ2)
      ‖ ρ, δ, λ1, λ2 :
          neighbors(P, ρ), has_label(P, λ1), has_label(ρ, λ2), λ1 > λ2,
          is_a_child_of(P, δ)†
              → is_a_child_of(P, ρ)
      ‖ true → Label5(P) ;
    Track1(P) ≡
        δ, λ : has_label(P, λ), is_a_child_of(P, δ)†,
          [∀ ρ : neighbors(P, ρ) :: has_label(ρ, λ), ¬is_a_child_of(ρ, P) ]
              → is_a_child_of(P, nil)
      ‖ has_label(P, P), is_a_child_of(P, nil) → wins(P)
      ‖ λ : has_label(P, λ), ¬wins(λ) → Track1(P)
  ]
initialization
  [P : Pixel(P) ::
      has_intensity(P, Intensity(P)); has_label(P, P); is_a_child_of(P, P);
      Label5(P); Track1(P)
  ]
end
```

Figure 2: A Region-Labeling Program with Termination Detection
Using a Classic Algorithm to Detect the Termination of a Diffusing Computation

Coordinated detection. The first paradigm could be called coordinated detection, a computation which executes a special protocol to detect the desired condition. Termination [15], quiescence [9], and global snapshot algorithms [8] 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 1 to form the program shown in Figure 2. 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.

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 *RegionLabel*2 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 λ to a smaller value.

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

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 regular query which fails if the pixel is properly labeled with respect to its neighbors, (2) a second regular query which fails if the $Track3$ transaction for the pixel is in synchrony with the $Track3$ transactions for all neighbors, (3) a special query which succeeds and recreates $Track3$ if any of the regular queries of any transaction in the synchronic group succeeds, and (4) a special query which succeeds if all regular queries fail and this detector transaction is associated with the winning pixel:

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

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:

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

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 DISCUSSION

The objective of this section is to relate *Swarm* to other research endeavors that have been instrumental in its conception. In doing so, we examine *Swarm* from three distinct perspectives—as a model, a language, and a programming methodology.

4.1 Models

The best way to relate Swarm to existing work is to compare it to the UNITY model [10]. Because UNITY's approach and style have greatly influenced the development of the shared dataspace model, the distinctions between the two are easiest to draw. In UNITY concurrent computations are defined by a fixed set of statements and variables. Each statement may include multiple conditional assignments. In an infinite computation each statement is executed infinitely often. The computation can be stopped as soon as the program reaches a fixed point—termination is considered to be an implementation issue and not a computation concern. UNITY is defined in terms of a very small set of constructs, is able to model both synchronous and asynchronous computations, and includes a powerful proof system.

In Swarm, the fixed set of variables has been replaced by an unbounded set of tuples; the conditional assignment has been replaced by transactions which can examine, insert, and delete elements of the dataspace. The interpretations of the $\|$ operator are similar in UNITY and Swarm. However, the latter places no restrictions on the composition of subtransactions into transactions—the synchronous nature of subtransaction execution guarantees no interference among subtransactions. Swarm permits both the creation of new transactions and the dynamic coupling of transactions

in the transaction space. As a further distinction, a transaction is removed from the transaction space as soon as it executes. Clearly, the class of UNITY programs is a proper subset of the class of Swarm programs. The additional features are provided at the expense of a more complex proof system. In some sense, one can think of Swarm as a model which allows for trade-offs between expressive power and proof complexity.

Swarm shares the philosophy and goals of UNITY, but attempts to reach beyond the two dominant concurrency paradigms (shared variables and message passing). The rich class of computing styles possible in Swarm is interesting, not only as an academic exercise, but as a practical matter as well. First, reasoning about software systems which involve multiple computing paradigms is made possible by the availability of the Swarm proof logics. Second, unbounded and unstructured problems (e.g., operations on very sparse matrices) benefit from the highly dynamic nature of the model. Third, we expect the partial synchrony present in the formation of synchronic groups to open new opportunities for enhanced performance on some classes of algorithms and to suggest new architectural features for future classes of multiprocessors. We know of no other formal studies of algorithms which uses dynamic partial synchrony. Finally, we believe that Swarm and its proof logics can serve as an example of how to build similar proof logics for the parallel rule-based systems currently under development.

4.2 Languages

Among existing languages Swarm's closest relative is Linda [7]. In his advocacy of the Linda language, Gelernter has relied heavily upon the temporal and spatial decoupling that can be achieved when data are accessed by content rather than by name. Our experience, however, shows that the degree of decoupling one can achieve depends greatly upon the power of the atomic transactions available to the programmer (accessing one tuple at a time is very limiting) and on the ability to organize the computation dynamically in response to the unpredictable structure of the data being processed (e.g., on a region-by-region basis in the labeling problem). Neither Linda nor the traditional approaches to concurrent computation, such as the UNITY paradigm and the data-parallel [17] computing style used to write Connection Machine algorithms, can accomplish this. Linda's limitations are the result of a language development philosophy different from that of Swarm—a philosophy which favors an efficient implementation over programming convenience and the capability to reason formally about programs. The limitations of UNITY and data-parallel

programs result from the fixed computational structures imposed by their programming models and notation.

All other shared dataspace languages of which we are aware, such as Associons [25, 26] and OPS5 [6], are only marginally concerned with concurrency. Moreover, the closure statement of Associons and the production rules of OPS5 have straightforward Swarm implementations. The potential impact of languages such as Swarm on the future generation of expert system shells is an interesting question that deserves careful consideration. Swarm's advantages might rest with its ability to organize transactions into synchronic groups in response to the changing dataspace configuration and with the availability of a proof system. Its disadvantage may be found in the nondeterministic selection of transactions to be executed—expert systems often have complex scheduling rules based on rule priorities and the age of the data.

4.3 Methodologies

In Swarm, the *replicated worker* [2, 7] metaphor proposed for Linda 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 (synchronic groups) as a program composition mechanism remains to be investigated.

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. 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 $Label2(P, L)$ and $Label2a(P)$ 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 monitor either the tuple or the transaction spaces, seeking to determine the end of a particular phase in the computation.

The reliance on formal reasoning about concurrent computations is also at the base for our approach to program visualization [28]. The visualization approach uses invariants and progress conditions to determine the kinds of visual representations which are most likely to convey the workings of the program. Actually, because the entire computational state is given by the dataspace, new and highly effective approaches to the visualization of concurrent computations are made possible.

5 CONCLUSIONS

In this paper we have defined a language paradigm called shared dataspace, a paradigm in which computations are performed using an anonymous, content-addressable communication medium acted upon by atomic transactions. To probe the essence of this paradigm, we have defined a relatively simple programming notation called Swarm. This paper has overviewed the Swarm model and notation and discussed some of their programming implications and distinctive features. We emphasized the generality of the model and its potential impact on programming style. This is, in part, because we see Swarm, not so much as a language, but as a framework for investigating concurrency paradigms and languages. This work forms the basis for further investigation of programming methodologies [29], proof systems [12, 13, 30], and approaches to program visualization [28].

APPENDIX: A FORMAL MODEL

In this appendix we present an operational, state-transition model for Swarm. This model formalizes the concepts expressed informally in section 2 and lays the foundation for the development of Swarm programming logics [12, 13, 30]. The model presented here is similar to the one presented in [29].

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 model we restrict ourselves to integer (the set **Int**) and boolean (the set **Bool**) values and finite sequences thereof. **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. If R is a binary relation on some set, then R^τ 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^∞ denotes the set of all infinite sequences. The symbol ε signifies the empty (zero-length) sequence. Sequence elements are indexed with natural numbers beginning with 0. The notation s_i designates the ith 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 the data tuple types. For all *name* and *values*, $\mathbf{TP}(name, values) = true$ 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 the transaction types. For all *name* and *values*, $\mathbf{TR}(name, values) \neq \varepsilon$ 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 $(\mathbf{R} \cup \mathbf{S})^*$ where:

$\mathbf{R} \cap \mathbf{S} = \emptyset$**.**

$\mathbf{R} \subseteq [\ \mathbf{Bool}^* \rightarrow \mathbf{DS} \rightarrow \mathbf{Val}^* \rightarrow \mathbf{Bool} \times \mathbf{DS} \times \mathbf{DS}\ ]$ is a set of behaviors for subtransactions which involve only *regular* predicates in their queries. Each element of **R** 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

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

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

$\mathbf{S} \subseteq [\ \mathbf{Bool}^* \rightarrow \mathbf{DS} \rightarrow \mathbf{Val}^* \rightarrow \mathbf{Bool} \times \mathbf{DS} \times \mathbf{DS}\ ]$ is a set of behaviors for subtransactions involving the *special* predicates **AND**, **OR**, **NAND**, and **NOR** as discussed in section 2. The $\mathbf{Bool}^*$ arguments represent the success and failure results of all the regular subtransaction queries executed in the same step. The function range is interpreted in the same way

as in **R**. Given a dataspace d, a sequence of regular query results b, and a sequence of values for the subtransaction variables v

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

because the special predicates are commutative and associative.

Swarm subtransactions can be translated to **R** and **S** functions in a straightforward manner.

For convenience, we define a number of prefix operators. For any dataspace d in **DS**, **Tp.**d, **Tr.**d, and **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 **Tp.**d yields the tuple space a. For any subtransaction behavior s in $\mathbf{R} \cup \mathbf{S}$, **Q.**$s$, **D.**$s$, and **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 section, 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 $i \geq 0$, **Ds**.e_i is the first component of e_i (the "current" dataspace) and **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 3. 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 G if and only if G is a synchronic group of d's transaction space and d' is a possible result of the synchronous execution of all the subtransactions in G 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 special predicates depend upon the success or failure of the regular subtransactions as well as directly upon the dataspace.

Some of the notation in Figure 3 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. The "array" b is used to model the "passing" of the success/failure results of the regular subtransactions to the special transactions. In lines 10 and 11 the queries for the special transactions depend upon the elements of b corresponding to regular subtransactions. In the definition of $reg(b, G)$ 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 section 2 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:

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

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. (An unfair execution sequence would be one in which a transaction enters the transaction space, but is never executed.)

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

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

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.

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

where

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

and

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

and

$$
\quad \mathit{reg}(b, G) \equiv \langle \mathrm{SEQ}\ t, i, \sigma : \mathit{subtrans}(G, t, i, \sigma) \wedge \sigma \in \mathbf{R} : b(t, i) \rangle
$$

and

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

Figure 3: The Transition Relation **step**

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. Using this state-transition model to capture the desired notion of program execution, we have developed two programming logics for Swarm. The first programming logic, described in [12] and [13], provides a proof system for a subset of Swarm without the synchronic group feature; the second logic, described in [12] and [29], generalizes the proof system to handle synchronic groups. We define the Swarm logics in terms of the same logical relations as UNITY [10] (**unless**, **ensures**, and **leads-to**), but must reformulate several of the concepts to accommodate Swarm's distinctive features. We have constructed our logics carefully so that most of the theorems developed for UNITY can be directly adapted to the Swarm logic. The above concept of fairness is a central assumption of the logics; it is essential to proofs of progress (liveness) properties of Swarm programs.

Acknowledgements

This work was supported by the Department of Computer Science, Washington University, Saint Louis, Missouri. We thank Jerome R. Cox, department chairman, for his support and encouragement. In developing the Swarm model and logic, we have benefited from discussions with many colleagues at Washington University—Ken Cox, Wei Chen, Howard Lykins, Mike Ehlers, Jan Tijmen Udding, Dan Kimura, and Rose Fulcomer Gamble. We also thank the referees and Jayadev Misra for their helpful comments and the Department of Computer and Information Science at The University of Mississippi for enabling the second author to continue this work.

References

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

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

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

[4] H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the 1988 International Conference on Computer Languages*, pages 82–91. IEEE, October 1988.

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] H. C. Cunningham. *The Shared Dataspace Approach to Concurrent Computation: The Swarm Programming Model, Notation, and Logic*. PhD thesis, Washington University, Department of Computer Science, St. Louis, Missouri, August 1989. Advisor: G.-C. Roman.

[13] H. C. Cunningham and G.-C. Roman. A UNITY-style programming logic for a shared dataspace language. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.

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

[15] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

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

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

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

[19] 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.

[20] L. Lamport. On interprocess communication. *Distributed Computing*, 1:97–111, 1986.

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

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

[23] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[24] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *Proceedings of the 1988 Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA)*, pages 276–284. ACM, September 1988. Also *SIGPLAN Notices* 23(11), November 1988.

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

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

[27] 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.

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

[29] 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.

[30] G.-C. Roman and H. C. Cunningham. The synchronic group: A concurrent programming concept and its proof logic. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 142–149. IEEE, May 1990.

[31] 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.