Report Number: WUCS-88-33

1988-10-01

A Shared Dataspace Model of Concurrency -- Language and Programming Implications

Gruia-Catalin Roman and H. Conrad Cunningham

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. This paper reports on the progress we have made toward the development of a formal operational model for Swarm and a few of the language and programming implications of the model. The paper has four parts: an overview of the Swarm language, a presentation of a formal operational... **Read complete abstract on page 2.**

A Shared Dataspace Model of Concurrency -- Language and Programming Implications

Gruia-Catalin Roman and H. Conrad Cunningham

Complete 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. This paper reports on the progress we have made toward the development of a formal operational model for Swarm and a few of the language and programming implications of the model. The paper has four parts: an overview of the Swarm language, a presentation of a formal operational model, 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 SHARED DATASPACE MODEL OF CONCURRENCY
— Language and Programming Implications —

Gruia-Catalin Roman and H. Conrad Cunningham

WUCS-88-33

October 1988
Revised March 1989

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

To appear in the *Proceedings of the 9th International Conference on Distributed Computing Systems*, IEEE, Newport Beach, California, June 1989.

A SHARED DATASPACE MODEL OF CONCURRENCY
— Language and Programming Implications —

Gruia-Catalin Roman and H. Conrad Cunningham

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130

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. This paper reports on the progress we have made toward the development of a formal operational model for Swarm and a few of the language and programming implications of the model. The paper has four parts: an overview the Swarm language, a presentation of a formal operational model, 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.

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 three basic paradigms: *communication via shared variables*, (synchronous or asynchronous) *message-based communication*, and *remote operations*. The three paradigms differ in the mechanisms they provide for communication among concurrent processes. However, all three rely on 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 very 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[10] 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. Languages without name-based accessing of data are rare. Associons[11] is a notable exception—data are represented by a set of tuples which may be altered by means of an operation called the *closure statement*. Although the closure statement facilitates highly parallel execution, Associons is not a concurrent language—it does not provide mechanisms for explicit representation of concurrency.

The first concurrent language to make extensive use of a content-addressable communication medium is Linda[1]. 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[12,13], a language in which processes use powerful transactions to manipulate abstract views of a virtual, content-addressable data structure called the *dataspace*. The Transaction Network[9] is a visual language in which the traditional places and transitions appearing in Petri nets have been replaced by databases and transactions, respectively.

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 and models has just begun, the body of knowledge accumulated to date is too limited to reach any decisive conclusions about the paradigm's long-term viability. Even so, the Linda-related work, our own experiments with SDL, 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.

During the past year 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 the UNITY model[5], 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 study, which is far from being completed, has a very broad scope, encompassing the development of formal (operational and axiomatic) semantic models, novel programming metaphors specific to the shared dataspace paradigm, implementation strategies, and new approaches to visualizing concurrent computations[14].

This paper reports on the progress we have made toward the development of a formal operational model for Swarm and a few of the language and programming implications of the model. The paper has four parts. Section 2 informally overviews the Swarm language. Section 3 follows with a formal operational model. To illustrate the kinds of algorithms one can construct in shared dataspace languages, section 4 presents several solutions to the problem of labeling equal-intensity regions within a digital image. Section 5 highlights the distinctive features of the shared dataspace model.

2. THE SWARM LANGUAGE

To probe the essence of the shared dataspace paradigm, we define a shared dataspace language based on a few simple concepts. By choosing the name *Swarm* for this language, we evoke the image of a large aggregation of small, independent agents cooperating to perform a task.

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 section. An element of the dataspace is a pairing of a *type* name with a sequence of *values*. In addition, a transaction has an associated behavior specification.

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

2.1. Program Organization

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

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

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)]

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

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

Program header. The program header associates a name with the program and optionally defines a set of program parameters. An invocation of the program with specific argu-

```
program RegionLabel1 ( Rows, Cols, Lo, Hi, Intensity :
        1 ≤ Rows, 1 ≤ Cols, 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 ≤ Cols, 1 ≤ y ≤ Rows ] ;
    P neighbors Q ≡ Pixel(P), Pixel(Q), P ≠ Q,
        [∃ x,y,a,b : P=(x,y), Q=(a,b) :
                a−1 ≤ x ≤ a+1, b−1 ≤ y ≤ b+1],
        [∃ ι :: has_intensity(P,ι), has_intensity(Q,ι)] ;
    P is_labeled L ≡ has_label(P,L)
    ]
tuple types
    [ P, L, I : Pixel(P), Pixel(L), Lo ≤ I ≤ Hi :
        has_label(P,L) ;
        has_intensity(P,I)
    ]
transaction types
    [ P : Pixel(P) :
    Label(P) ≡
            ρ,λ1,λ2 : P is_labeled λ1†, ρ is_labeled λ2,
                ρ neighbors P, λ1 > λ2
                    → P is_labeled λ2, Label(P)
        ||  NOR     → Label(P)
    ]
initialization
    [ P : Pixel(P) :
        has_label(P,P), has_intensity(P,Intensity(P)),
        Label(P)
    ]
end
```

Figure 1: Nonterminating Region Labeling in Swarm

ment values causes the substitution of the values for the parameter names throughout the program's body. The argument values must satisfy the constraint predicates given in the program header. The program in Figure 1 is named *RegionLabel1*; it has parameters *Rows, Cols, Lo, Hi,* and *Intensity* constrained as indicated. *Intensity* is an array of input values indexed by the pixel coordinates.

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

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

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

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

$$\begin{array}{l} variable\_list_1 : query_1 \rightarrow action_1 \\ \| \quad \ldots \\ \| \quad variable\_list_n : query_n \rightarrow action_n \end{array}$$

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. *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.

2.2. 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. Using this feature programs can dynamically form groups of transactions; these groups are executed as if they were a single transaction.

The synchrony relation is a symmetric relation on the set of valid transaction instances. The reflexive transitive closure of the synchrony relation is an equivalence relation. 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 synchrony relation can be examined and modified in much the same way as the tuple and transaction spaces can. The predicate

Label(p) ~ Label(Q)

(where *p* is a variable and *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 ≈ can be used in a predicate to examine whether transaction instances are related by the closure of the synchrony relation. (The scope of the global predicates, e.g., **AND**, extends to all local subtransactions in the synchronic group.)

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

Label(p) ~ 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 instances are related by the synchrony relation, then

(Label(p) ~ Label(q))†

deletes the relationship. Note that the closure relation can be examined, but that only the base synchrony relation can be modified.

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

3. A FORMAL MODEL

In this section we present an operational, state-transition model for Swarm. This model formalizes the concepts expressed informally in the previous section and lays the foundation for our development of a programming logic for the language. The reader uninterested in formal semantics can proceed to the next section on programming implications.

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 paper we restrict ourselves to integers (set **Int**) and booleans (set **Bool**). **Nam** is the set from which names of tuple and transaction types are drawn ($\mathbf{Nam} \cap \mathbf{Val} = \varnothing$).

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^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 ⟨**TP**, **TR**, **SR**, **ID**⟩ where:

TP : Nam → Val* → Bool
is the characteristic function for data tuple types. **TP**(*name,values*) = **true** iff *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 → Val* → Beh
is the characteristic function for transaction types. **TR**(*name,values*) ≠ ε iff *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**):

$$\mathbf{TPS} = \{ n, v : n \in \mathbf{Nam} \wedge v \in \mathbf{Val}^* \wedge \mathbf{TP}(n,v) : (n,v) \}$$

$$\mathbf{TRS} = \{ n, v : n \in \mathbf{Nam} \wedge v \in \mathbf{Val}^* \wedge \mathbf{TR}(n,v) \neq \varepsilon : (n,v) \}$$

SR is a subset of Pow(**TRS** × **TRS**).

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

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

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:

$\mathbf{L} \cap \mathbf{G} = \varnothing$

$\mathbf{L} \subseteq [\, \mathbf{Bool}^* \rightarrow \mathbf{DS} \rightarrow \mathbf{Val}^* \rightarrow \mathbf{Bool} \times \mathbf{DS} \times \mathbf{DS} \,]$
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 **Bool*** argument is a "dummy" included for compatibility with **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 the previous section. The **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**, **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 $\mathbf{L} \cup \mathbf{G}$, **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**, $(\mathbf{Sr}.d)^*$ 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:

$$\mathbf{ES} = (\mathbf{DS} \times \mathrm{Fs}(\mathbf{TRS}))^{\omega}$$

For all $e \in \mathbf{ES}$ 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 2. 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 iff 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 2 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. 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 the previous section 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:

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

where
  $\mathrm{Synch}(S,d) \equiv$
    $(S = \emptyset \wedge \mathbf{Tr}.d = \emptyset) \vee$
    $(S \neq \emptyset \wedge S \subseteq \mathbf{Tr}.d \wedge$
      $(\forall\, t, t' : t \in S \wedge t' \in S : (t,t') \in (\mathbf{Sr}.d)^*) \wedge$
      $(\forall\, t, x : t \in S \wedge x \in \mathbf{Tr}.d \wedge x \notin S :$
        $(t,x) \notin (\mathbf{Sr}.d)^*))$
and
  $\mathrm{subtrans}(S,t,i,\sigma) \equiv t \in S \wedge 0 \leq i < \#\mathbf{TR}(t) \wedge \sigma = (\mathbf{TR}(t))_i$
and
  $\mathrm{loc}(b,S) \equiv$
    $(\mathrm{SEQ}\ t, i, \sigma : \mathrm{subtrans}(S,t,i,\sigma) \wedge \sigma \in \mathbf{L} : b(t,i))$
and
  $\mathrm{Update}(d,S,d',v,b) \equiv$
    $\mathbf{Tp}.d' = (\mathbf{Tp}.d - (\cup\, t, i, \sigma : \mathrm{subtrans}(S,t,i,\sigma) \wedge b(t,i) :$
        $\mathbf{Tp.D}.\sigma(\mathrm{loc}(b,S),d,v(t,i))))$
      $\cup\, (\cup\, t, i, \sigma : \mathrm{subtrans}(S,t,i,\sigma) \wedge b(t,i) :$
        $\mathbf{Tp.I}.\sigma(\mathrm{loc}(b,S),d,v(t,i)))$
    $\wedge\ \mathbf{Tr}.d' = (\mathbf{Tr}.d - S)$
      $\cup\, (\cup\, t, i, \sigma : \mathrm{subtrans}(S,t,i,\sigma) \wedge b(t,i) :$
        $\mathbf{Tr.I}.\sigma(\mathrm{loc}(b,S),d,v(t,i)))$
    $\wedge\ \mathbf{Sr}.d' = (\mathbf{Sr}.d - (\cup\, t, i, \sigma : \mathrm{subtrans}(S,t,i,\sigma) \wedge b(t,i) :$
        $\mathbf{Sr.D}.\sigma(\mathrm{loc}(b,S),d,v(t,i))))$
      $\cup\, (\cup\, t, i, \sigma : \mathrm{subtrans}(S,t,i,\sigma) \wedge b(t,i) :$
        $\mathbf{Sr.I}.\sigma(\mathrm{loc}(b,S),d,v(t,i)))$

Figure 2: The Transition Relation

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

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:

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

This is the set of all execution sequences which begin in a valid initial dataspace, execute a synchronic group of transac-

tions 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 a programming logic for Swarm[6]. This programming logic is similar in style to the logic for UNITY[5]. The above concept of fairness is a central assumption of the logic; it is essential to proofs of liveness (progress) properties of Swarm programs.

4. PROGRAMMING IMPLICATIONS

In the preceding sections 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 section we provide some of the motivation for these particular choices. We do this by identifying the kinds of programming strategies made possible by these language constructs.

Reasoning about concurrent computations is generally done in terms of liveness (progress) 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. 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 problem of labeling connected regions of equal intensity within a digital image. Throughout this subsection 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.

In this section we discuss a series of solutions to the region-labeling problem. To distinguish among similar transactions in the various solutions, we append unique numbers to the base transaction names.

Static asynchronous computation. First, we consider the case of an asynchronous computation with a static transaction space. In a manner similar to Figure 1, the transaction space consists of one transaction per pixel:

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

Each *Label1(P)* transaction recreates itself and thus leaves the transaction space unchanged:

```
[ P : Pixel(P) :
Label1(P) ≡
        ρ,λ1,λ2 : P is_labeled λ1†, ρ is_labeled λ2,
            ρ neighbors P, λ1 > λ2
                → P is_labeled λ2
    ‖   true    → Label1(P)
]
```

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) : ···, Label2(P,P) ] ;
```

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

```
[ P, L : Pixel(P), Pixel(L) :
Label2(P,L) ≡
    [ ‖δ : P = L, δ next_to P :
        ι : has_intensity(P,ι), has_intensity(δ,ι)
                → Label2(δ,P)
    ]
    ‖
        λ : P is_labeled λ†, λ > L
                → P is_labeled L
    ‖
    [ ‖δ : δ ≠ L,  δ next_to P :
        λ, ι : P is_labeled λ, λ > L,
            has_intensity(P,ι), has_intensity(δ,ι)
                → Label2(δ,L)
    ]
]
```

Each *Label2(P,L)* transaction consists of three groups of subtransactions. In the first and third groups we use a new Swarm feature, the subtransaction generator. For $P = L$, the first group includes a subtransaction for each δ such that δ **next_to** P; otherwise, the group is null. (The **next_to** predicate is defined like **neighbors** except that the pixels are not required to have equal intensities.) 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 supertransaction that covers the entire image:

```
[ ::
Label3( ) ≡
    [ ‖ ρ : Pixel(ρ) :
        δ,λ1,λ2 : ρ is_labeled λ1†,δ is_labeled λ2,
            δ neighbors ρ, λ1 > λ2
                → ρ is_labeled λ2 ]
    ‖  true    → Label3( )
]
```

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 ~. 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

```
[ P : Pixel(P) :
    has_label(P,P), has_intensity(P,Intensity(P)),
    Label1(P) ] ;
[ P, Q : Pixel(P), Pixel(Q), P ≠ Q,
    [∃ x,y,a,b : P=(x,y), Q=(a,b) :
        Intensity(P) = Intensity(Q),
        a−1 ≤ x ≤ a+1, b−1 ≤ y ≤ b+1 ]
    : Label1(P) ~ Label1(Q) ]
```

All the transactions working on the same region form a synchronic group which recreates 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:

```
[ P : Pixel(P) :
Label4(P) ≡
        ρ, λ1, λ2 : P is_labeled λ1†,ρ is_labeled λ2,
            ρ neighbors P, λ1 > λ2
                → P is_labeled λ2
    ‖   ρ : ρ neighbors P, ¬(Label4(ρ) ~ Label4(P))
                → Label4(ρ) ~ Label4(P)
    ‖   true    → Label4(P)
]
```

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

4.2. 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[7], quiescence[4], and global snapshot algorithms[3] are representative of this paradigm. Algorithms for detecting the termination of a diffusing computation may be adapted to detecting

```
program RegionLabel2( Rows, Cols, Lo, Hi, Intensity :
            1 ≤ Rows, 1 ≤ Cols, 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 ≤ Cols, 1 ≤ y ≤ Rows ] ;
    P neighbors Q ≡
        Pixel(P), Pixel(Q), P ≠ Q,
        [∃ x,y,a,b : P=(x,y), Q=(a,b) :
            a−1 ≤ x ≤ a+1, b−1 ≤ y ≤ b+1],
        [∃ ι :: has_intensity(P,ι), has_intensity(Q,ι) ] ;
    P is_labeled L ≡ is_labeled(P,L) ;
    P is_the_winner ≡ wins(P) ;
    P is_not_the_winner ≡ ¬wins(P) ;
    P is_a_child_of Q ≡ is_a_child_of(P,Q) ;
    P is_not_a_child_of Q ≡ ¬is_a_child_of(P,Q)
    ]
tuple types
    [ P, Q, L, I :
        Pixel(P), Pixel(Q) ∨ Q = nil, Pixel(L), Lo ≤ I ≤ Hi :
            is_labeled(P,L) ;
            has_intensity(P,I) ;
            is_a_child_of(P,Q) ;
            wins(P)
    ]
transaction types
    [ P : Pixel(P) :

    Label5(P) ≡
            ρ,λ1,λ2 : P is_labeled λ1†,
                ρ is_labeled λ2, ρ neighbors P, λ1 > λ2
                    → P is_labeled λ2
        ‖   ρ,δ,λ1,λ2 : P is_labeled λ1,P is_a_child_of δ†,
                ρ is_labeled λ2, ρ neighbors P, λ1 > λ2
                    → P is_a_child_of ρ
        ‖   true    → Label5(P) ;

    Track1(P) ≡
            λ,δ : P is_labeled λ, P is_a_child_of δ†,
                [∀ ρ : ρ neighbors P :
                    ρ is_labeled λ, ρ is_not_a_child_of P]
                    → P is_a_child_of nil
        ‖   P is_labeled P, P is_a_child_of nil
                    → P is_the_winner
        ‖   λ : P is_labeled λ, λ is_not_the_winner
                    → Track1(P)

    ]
initially
    [ P : Pixel(P) :
        is_labeled(P,P), is_a_child_of(P,P),
        has_intensity(P,Intensity(P)),
        Label5(P), Track1(P)
    ]
end
```

Figure 3: Region Labeling in Swarm
Using a Classic Algorithm to Detect the Termination of a Diffusing Computation

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 3. The key modification is the introduction of a tuple *is_a_child_of*(ρ,δ) 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(P)* 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 activity which may occur once the stable state is established. Of course, this is possible only if the transaction space is dynamic. In the dynamic asynchronous solution presented above, when labeling 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:

```
[ P : Pixel(P) :
Track2(P) ≡
        alive(P), [∃ ρ :: Label2(ρ,P)]
                → Track2(P)
    ||  alive(P), [∀ ρ :: ¬Label2(ρ,P)]
                → P is_the_winner
]
```

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

The *Track3* 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(λ)* whenever the it relabels any pixel labeled λ to a smaller value.

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:

```
[ P : Pixel(P) :
Track3(P) ≡
        ρ,λ1,λ2 : P is_labeled λ1, ρ is_labeled λ2,
            ρ neighbors P, λ1 > λ2
                → skip
    ||  ρ : ρ neighbors P, ¬(Track3(P) ~ Track3(ρ))
                → Track3(P) ~ Track3(ρ)
    ||  OR      → Track3(P)
    ||  NOR, P is_labeled P → P is_the_winner
]
```

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. The approach is compatible with all labeling solutions presented earlier. This approach 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:

```
[ P : Pixel(P) :
Track4(P) ≡
        P is_labeled P, P is_not_the_winner
            → Track4(P)
    ||  P is_labeled P,
        [∀ ρ,δ : ρ is_labeled P, δ neighbors ρ :
                δ is_labeled P ]
            → P is_the_winner
]
```

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.

5. DISCUSSION

The desire to assist programmers in the development and analysis of concurrent computations motivates the shared dataspace model put forth in this paper. 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. On-going work on a proof system for shared dataspace languages and on declarative visualization techniques[14] promise to contribute to increased analyzability of shared dataspace programs. This section discusses the shared dataspace paradigm in relation to the current landscape of concurrent programming models, languages, and methodologies.

Models. The best way to relate the shared dataspace model (SDM) to existing work is to compare it against the UNITY model[5]. Because UNITY's approach and style have influenced greatly 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 SDM, 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 SDM. However, the latter places no restrictions on the composition of subtransactions into transactions—the synchronous nature of subtransaction execution guarantees no interference among subtransactions. SDM permits both the creation of new transactions and dynamic coupling among transactions in the transaction space. As a further distinction, a transaction is removed from the transaction space as soon as it executes. It is trivial to show that the class of UNITY programs is a proper subset of the class of SDM programs. The additional features are provided at the expense of a more complex proof system. At the base level one can think of SDM as a model which allows for trade-offs between expressive power and proof complexity. We believe, however, that the flexibility built into SDM is required when faced with unbounded and unstructured problems. Moreover, SDM covers a richer class of computing styles. The dynamic coupling introduced by the synchrony relation, for instance, models systems which can be reconfigured to behave synchronously on a region-by-region basis. Transactions are better suited for modeling production rule systems than assignment statements are.

Languages. Among existing languages Swarm's closest relative is Linda[1]. In his advocacy of the Linda language, Gelernter has relied heavily upon the temporal and spatial decoupling that can be achieved when data is 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[8] 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 which we are aware of, such as Associons[11] and OPS5[2], 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.

Methodologies. In Swarm, the *replicated worker* 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 liveness 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, merging 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 3. 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:

```
[ P : Pixel(P) :
SpeedUp(P) ≡
        ρ,λ : P is_labeled ρ†, ρ is_labeled λ
            → P is_labeled λ, SpeedUp(P)
]
```

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(δ,λ)* transactions for each neighbor δ of *P* whenever *P* is relabeled with λ.

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.

The reliance on formal reasoning about concurrent computations is also at the base for our approach to program visualization. The visualization approach uses invariants and progress conditions to determine the kinds of visual representa-

tions 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.

6. CONCLUSIONS

In the introduction we asked whether *naming* is fundamental to achieving cooperation among concurrent processes. To investigate this question we have defined a language paradigm called shared dataspace which causes computations to be 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 shared dataspace language called Swarm. This paper has overviewed the Swarm language, presented a formal operational model for the language, and discussed some of the programming implications and distinctive features of the model and language. This work forms the basis for further investigation of appropriate programming methodologies, proof systems, approaches to program visualization, and implementation techniques. Although this work is far from complete, the results to date are highly encouraging.

Acknowledgments: This work was supported by the Department of Computer Science, Washington University, Saint Louis, Missouri. The authors express their gratitude to Jerome R. Cox, department chairman, for his support and encouragement. We thank our colleagues Ken Cox, Howard Lykins, Wei Chen, and Mike Ehlers for their contributions to the shared dataspace research and their reviews of this paper. We also thank the referees for their helpful comments.

7. REFERENCES

1. Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," *Computer* **19**(8) pp. 26-34 (August 1986).
2. Brownston, L., Farrell, R., Kant, E., and Martin, N., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, Massachusetts (1985).
3. Chandy, M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems* 3(1) pp. 63-75 (February 1985).
4. Chandy, M. and Misra, J., "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," *ACM Transactions on Programming Languages and Systems* 8(3) pp. 326-343 (July 1986).
5. Chandy, M. and Misra, J., *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Massachusetts (1988).
6. Cunningham, H. C. and Roman, G.-C., "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).
7. Dijkstra, E. W. and Scholten, C. S., "Termination Detection For Diffusing Computations," *Information Processing Letters* **11**(1) pp. 1-4 (August 1980).
8. Hillis, W. D. and Steele Jr., G. L., "Data Parallel Algorithms," *Communications of the ACM* **29**(12) pp. 1170-1183 (December 1986).
9. Kimura, T. D., "Visual Programming by Transaction Network," pp. 648-654 in *Proceedings of the 21st Hawaii International Conference on System Sciences*, IEEE, Kona, Hawaii (January 1988).
10. Lamport, L., "On Interprocess Communication," *Distributed Computing* **1** pp. 97-111 (1986).
11. Rem, M., "Associons: A Program Notation with Tuples Instead of Variables," *ACM Transactions on Programming Languages and Systems* 3(3) pp. 251-262 (July 1981).
12. Roman, G.-C., Cunningham, H. C., and Ehlers, M. E., "A Shared Dataspace Language Supporting Large-Scale Concurrency," pp. 265-272 in *Proceedings of the 8th International Conference on Distributed Computing Systems*, IEEE, San Jose, California (June 1988).
13. Roman, G.-C., "Language and Visualization Support for Large-Scale Concurrency," pp. 296-308 in *Proceedings of the 10th International Conference on Software Engineering*, IEEE, Singapore (April 1988).
14. Roman, G.-C. and Cox, K. C., "Declarative Visualization in the Shared Dataspace Paradigm," in *Proceedings of the 11th International Conference on Software Engineering*, IEEE, Pittsburgh (May 1989).