Report Number: WUCS-88-09

1988-03-01

A Shared Dataspace Language Supporting Larger-Scale Concurrency

Gruia-Catalin Roman, H. Conrad Cunningham, and Michael E. Ehlers

Our ultimate goal is to develop the software support needed for the design, analysis, understanding, and testing of programs involving many thousands of concurrent processes running on a highly parallel multiprocessor. We are currently evaluating the use of a shared dataspace paradigm as the basis for a new programming language supporting large-scale concurrency. The language is called SDL (Shared Dataspace Language). In SDL, a content-addressable dataspace is examined and altered by concurrent processes using atomic transactions much like those in a traditional database. Associated with each process is a programmer-defined view. The view is a mechanism which allows processes... **Read complete abstract on page 2.**

A Shared Dataspace Language Supporting Larger-Scale Concurrency

Gruia-Catalin Roman, H. Conrad Cunningham, and Michael E. Ehlers

Complete Abstract:

Our ultimate goal is to develop the software support needed for the design, analysis, understanding, and testing of programs involving many thousands of concurrent processes running on a highly parallel multiprocessor. We are currently evaluating the use of a shared dataspace paradigm as the basis for a new programming language supporting large-scale concurrency. The language is called SDL (Shared Dataspace Language). In SDL, a content-addressable dataspace is examined and altered by concurrent processes using atomic transactions much like those in a traditional database. Associated with each process is a programmer-defined view. The view is a mechanism which allows processes to interrogate the dataspace at a level of abstraction convenient for the task they are pursuing. This paper provides an overview of the key SDL features. Small examples are used to illustrate the power and flexibility of the language. They also serve as a backdrop against which we discuss programming style implications of the shared dataspace paradigm.

A SHARED DATASPACE LANGUAGE SUPPORTING LARGE-SCALE CONCURRENCY

Gruia-Catalin Roman, H. Conrad Cunningham
and Michael E. Ehlers

WUCS-88-09

March 1988

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

To appear in *Proceedings of the 8th International Conference on Distributed Computing Systems*, IEEE, San Jose, June 1988.

A SHARED DATASPACE LANGUAGE SUPPORTING LARGE-SCALE CONCURRENCY

Gruia-Catalin Roman, H. Conrad Cunningham, and Michael E. Ehlers

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130

ABSTRACT

Our ultimate goal is to develop the software support needed for the design, analysis, understanding, and testing of programs involving many thousands of concurrent processes running on a highly parallel multiprocessor. We are currently evaluating the use of a shared dataspace paradigm as the basis for a new programming language supporting large-scale concurrency. The language is called SDL (Shared Dataspace Language). In SDL, a content-addressable dataspace is examined and altered by concurrent processes using atomic transactions much like those in a traditional database. Associated with each process is a programmer-defined *view*. The view is a mechanism which allows processes to interrogate the dataspace at a level of abstraction convenient for the task they are pursuing. This paper provides an overview of the key SDL features. Small examples are used to illustrate the power and flexibility of the language. They also serve as a backdrop against which we discuss programming style implications of the shared dataspace paradigm.

1. INTRODUCTION

In recent years multiprocessors consisting of thousands of tightly-coupled processors have become a reality and current technological and market trends suggest that multiprocessors will become commonplace in many computing environments. Nevertheless, the availability of such powerful machines is only the first step toward meeting the ever-increasing user needs for computational resources. The next step is the development of languages and environments that will enable the user to harness the raw power of these machines in a cost-effective manner. This is a difficult problem that must address both cultural and technical concerns. At a cultural level, appropriate training, skills, and experience are missing. At the technical level, concurrent programs consisting of thousands of processes present the programmer with unprecedented degrees of complexity which are further exacerbated by our limited capacity to reason about concurrent computation and to predict the performance of complex programs executing on highly parallel multiprocessors.

Breaking the software barrier requires researchers to reevaluate their current thinking about concurrency, from basic models to languages and environments. Ways must be found to cope with the complexities of *large-scale concurrency*, i.e., computations that require the cooperation of many thousands of (not necessarily identical) processes executing on a highly parallel multiprocessor. Our own efforts in this area have been directed toward evaluating the shared dataspace paradigm as the basis for a language supporting the specification and visualization of large-scale concurrent computations.

The *shared dataspace paradigm* covers languages in which processes have access to a common, content-addressable data structure (typically a set of tuples) whose components may be asserted, read, and retracted. Associons[8], Linda[1], and OPS83[4] represent three distinct instantiations of this paradigm. Gelernter's work on Linda has shown the paradigm's viability for large scale concurrency. However, he uses the term *generative communication*[5] to refer to this paradigm, but we find the term *shared dataspace* to be more explicit. It suggests an analogy with the shared variables mechanism and also alludes to the incorporation of database concepts into the programming language.

Our own attempt to experiment with the application of the shared dataspace paradigm to large-scale concurrency has led to a language proposal called SDL (*Shared Dataspace Language*). SDL shares with Associons and Linda the use of tuples to represent the dataspace. In SDL, however, the dataspace is examined and altered by concurrent processes using atomic transactions much like those in a traditional database but exhibiting a richer set of operational modes specifically designed for support of large-scale concurrency. Associons use the closure statement, which is very powerful and has high potential for concurrency but is intended for use in what might be considered a single explicit process environment. Linda provides processes with very simple dataspace access primitives (read, assert, and retract one tuple at a time). Another

distinguishing feature, unique to SDL, is the availability of programmer-defined process *views*. The view is a powerful mechanism which supports an abstract notion of locality and provides a way by which structure may be introduced into the tuple-based dataspace. More importantly, it allows processes to interrogate the dataspace at a level of abstraction convenient for the task they are pursuing. To the best of our knowledge, this kind of *relativistic* abstraction mechanism has never been explored before. Its advantages over the fixed encapsulation mechanisms available in modern languages are self-evident.

The objective of this paper is to provide the reader with an introduction to the shared dataspace paradigm and its implications on programming style. SDL is the vehicle by which we do this. Section 2 contains an overview of the key SDL features. In Section 3, small examples are used to illustrate the power and flexibility of the language. They also serve as a backdrop against which we discuss programming style implications of the shared dataspace paradigm. Brief concluding remarks are part of Section 4.

2. LANGUAGE OVERVIEW

For the sake of simplicity, examples given in this paper do not use the full power of SDL. As such, this section presents a simplified version of SDL. Some concepts are omitted and others are explained only to the extent to which they are being used in Section 3. For a more complete definition of the language and discussion of other topics related to SDL such as program visualization, the reader is directed to a recently published companion paper[9].

In SDL, computation is described in terms of a dataspace and a process society. The *dataspace* is a set of tuples. The *process society* is a set of processes. Both the dataspace and the process society undergo continuous change. *Tuples* are asserted, examined, and retracted by processes. Each tuple is owned by the process that asserted it and the owner may be determined by examining the unique tuple identifier associated with each tuple. Typically, tuple identifiers are ignored by application programs but are of interest during debugging and testing. Tuples may be manipulated by any process and can survive the termination of the creating process. *Processes*, in turn, are created by other processes, manipulate tuples, and terminate on their own.

The interactions between processes and the dataspace take place via *transactions* issued by individual processes. At a logical level, all transactions are atomic, i.e., transactions appear to execute serially and either succeed or have no effect on the dataspace. In general, transaction execution involves four subactions: a query evaluation over the dataspace, the retraction of selected tuples specified in the query, the assertion of new tuples, and local actions affecting the control state of the issuing process. Individual processes may initiate concurrent evaluation of multiple transactions with the intent of committing only one of them or may issue an unbounded number of concurrently executing transactions. Consistent with the notion of process/data decoupling, most transactions are independent of the process society state. However, there are instances where coordinated transaction execution by a set of cooperating processes is necessary, especially to maintain clarity in the program. This type of action is represented by a novel transaction type, called a *consensus transaction*, in which several processes perform simultaneous transformations on the dataspace with the composite of these transformations being a single transaction.

A *view* is associated with each process. The view specifies a window which, like the dataspace, is a set of tuples. The window, however, is computed only at the start of a transaction and is discarded as soon as the transaction commits. The tuples in the dataspace are mapped into the window using the *import* component of the view. Transactions act upon the window as if it represented the whole dataspace. Retractions of tuples in the window are translated to corresponding retractions in the dataspace in accordance with the import rules. The *export* part of the view maps tuples asserted in the window to new tuples in the dataspace. Conceptually, the view allows programmers to consider the dataspace at a level of abstraction that matches the processing requirements of a particular process. This leads to both clarity and brevity. Pragmatically, the view also provides bounds on the scope of the transactions which, in turn, reduce the transaction execution time. Thus, transaction types that might be expensive to implement may be used comfortably when the number of tuples they examine is small.

2.1. Dataspace and View

The dataspace D is defined as a finite but large multiset of tuples where each tuple t is a sequence of values from some domain V (e.g., atoms and integers). At the meta level, we will denote tuples as finite sequences of symbols as in

<year,87>

The dataspace is examined and modified by transactions. However, a process restricts the transactions it issues from operating on the dataspace directly. Invisible to the transaction,

the dataspace is replaced by a window W on which the transaction is evaluated. The transaction computes a *retraction window* Wr and an *assertion window* Wa which are used to update the dataspace.

The window is an abstraction of the dataspace relative to a particular process' needs. The abstraction mechanism is the *view* which defines both the abstraction rule and the way in which changes to the window are mapped back to corresponding changes in the dataspace. In its simplest form, a view may be characterized by two sets called *Import* and *Export*. The import set of a process p is the set of all tuples which p could query and retract, if they exist in the current dataspace configuration. The export set of p is the set of all tuples which p is allowed to add to the dataspace. Given a particular dataspace configuration D, a process p, and a transaction τ, the new dataspace configuration D' is computed as follows:

$$W = Import(p) \cap D$$
$$(Wr, Wa) = \tau(W)$$
$$D' = ((D - Wr) \cup (Export(p) \cap Wa))$$

In SDL, simple views such as these can be defined by using import/export statements consisting of tuples containing constants, wildcard markers ('*'), and bound variables. Given a process p, a view such as

IMPORT
$\alpha : \alpha \leq 87 :\Rightarrow$ (year,α)
EXPORT
(year,*)

defines

$Import(p) = \{ \langle year,\alpha \rangle \mid \alpha \in V \text{ and } \alpha \leq 87 \}$
$Export(p) = \{ \langle year,\alpha \rangle \mid \alpha \in V \}$

(Note: We use Greek letters for quantified variables, lower-case letters and numbers for constants, and upper-case letters for named constants).

2.2. Transactions

Dataspace membership testing, tuple retraction, and tuple assertion are the simplest SDL transactions. The membership test takes the form

[year,87]

This transaction is evaluated only once and must either succeed or fail depending upon whether the query evaluates to true or false. In a particular context, its success or failure may be used to alter the state of the process issuing the transaction; otherwise, it has no effect on the dataspace. To retract a tuple, one simply tags it (in the query) by a †. The transaction

[year,87]†

follows the membership test by the removal of the tuple. Note that retracting one instance of a tuple may leave other instances of it in the dataspace. To assert a tuple one can use a transaction such as

(year,87)

More complex queries may be formed by composing predicates using negation ('¬'), conjunction (','), disjunction ('or'), and parentheses. The general form of a transaction is

transaction ::=
    query
    transaction_type_tag
    action_list

query ::=
    quantifier
    variable_list :
    binding_query :
    test_query

quantifier ::= ∀ | ∃

transaction_type_tag ::= → | ≻ | ↑↑

where the transaction_type_tag determines some of the operational characteristics of the transaction.

Immediate transactions are tagged by '→', as in the transaction

$$\exists\, \alpha : [year,\alpha]\dagger : \alpha > 87 \rightarrow \text{let } N=\alpha, (found,\alpha)$$

The operational interpretation here is as follows. First, the binding_query is evaluated in an attempt to find a tuple of the form <year,*>. If the tuple <year,90> is found, the test query is initiated with α bound to 90. Since 90 is greater than 87, the query succeeds leaving α bound to 90 and the tuple <year,90> tagged for retraction. Next, the tuple is retracted. In the action_list, N is defined to be the constant 90 and the tuple <found,90> is added to the dataspace. Logically all these steps represent a single atomic transformation of the dataspace. The transaction would have failed if, at the time the query was evaluated, the dataspace contained no tuple of the form <year,*> with a number greater than 87 in the second position.

Delayed transactions, tagged by '≻', differ from immediate transactions in that, instead of failing, they block the process until a successful evaluation is possible. The delayed transaction

$$\exists\, \alpha : [year,\alpha] : \alpha > 87 \succ (new\_year)$$

is not executed until the dataspace contains a tuple of the form <year,*> with the second field greater than 87. However, a delayed transaction is not guaranteed to detect the first instance when the dataspace allows it to be successful. As far as fairness is concerned, we assume only that, if a

delayed transaction is issued and remains enabled indefinitely, the transaction is eventually executed.

Consensus transactions. Central to the shared dataspace paradigm is the notion of process/process and process/data decoupling. The transactions introduced so far are consistent with this principle. Transaction execution involves only the dataspace; it is independent of the state of the process society. However, the very fact that processes cooperate in solving a given task, leads naturally to cases where processes in a particular community must reach some common agreement, i.e., a *consensus*, before further processing anywhere in the community may proceed. This kind of situation occurs frequently in concurrent programs and in implementations of concurrent languages. Program termination in the UNITY model[3], task termination in Ada, the simulation of clocked systems, and the exit from a cobegin-coend block in some concurrent languages involve various forms of multiparty consensus. Actually, the two-way synchronization commonly used in many concurrent languages, such as CSP[7], is nothing more than a special case of the more general notion of consensus.

Our solution to consensus problems is to provide in SDL a specialized and powerful transaction type called a *consensus transaction.* In its simplest form, the consensus is an explicit n-way synchronization among processes that are members of the same consensus set. A *consensus set* is defined as a set of processes closed under the transitive closure of the relation

$$p \text{ needs } q \equiv (\mathit{Import}(p) \cap \mathit{Import}(q) \cap D \neq \emptyset)$$

A consensus transaction is executed whenever all processes in the consensus set are ready to execute consensus transactions. Determination that consensus has been reached is very similar to the *quiescence detection* problem[2]. The composite effect on the dataspace is computed by first performing the retractions associated with each of the participating transactions and then the corresponding additions to the dataspace.

Syntactically, consensus transactions are tagged by '††.' Semantically, except for their participation in the consensus, they act like delayed transactions in the sense that they block until consensus is reached. We will illustrate the use of consensus transactions and the process definitions they encourage in Section 3.

2.3. Transaction Sequencing

A process may sequence the execution of transactions by means of four flow-of-control constructs: sequence, selection, repetition, and replication. To form a *sequence*, two or more transactions are listed, separated by semicolons. The execution of one transaction must complete before the next one is initiated. In the sequence shown below, an index and a value are paired at random and placed in the dataspace.

```
∃ ρ : [index,ρ]† ≻ let X=ρ ;
∃ ν : [value,ν]† ≻ let Y=ν ;
(X,Y)
```

Sequences may be terminated prematurely by issuing the *exit* action.

The *selection* construct functions like a case statement consisting of several sequences separated by '[]'. These sequences are called *guarded sequences.* The transaction heading a guarded sequence is called a *guarding transaction.* Successful execution of one of the guarding transactions leads to the execution of its successors in the sequence followed by the termination of the construct. If more than one guarding transaction can be successfully executed, an arbitrary one (but only one) of them is selected for execution. If no guarding transactions can be successfully executed, the selection fails, i.e., the construct is terminated without execution of any sequence. Note that failure of the selection does not cause the process to abort or terminate, rather the selection is modeled as a 'skip' statement in this instance. Since delayed transactions cannot fail, a selection involving delayed transactions will block until one of the guarding transactions succeeds. This is the case in the example below, where either a value is paired with a positive index or a non-positive index is retracted.

```
[   ∃ ρ : [index,ρ]† : ρ>0 ≻ let X=ρ ;
    ∃ ν : [value,ν]† ≻ let Y=ν ;
    (X,Y)
[]  ∃ ρ : [index,ρ]† : ρ≤0 ≻ skip
]
```

The *repetition* construct works similarly but is restarted after each selection. Termination of the repetition normally occurs when a selection terminates without selecting a guarded sequence. Termination of the repetition can made explicit by including the action *exit* in the action_list of some transaction in one of the guarded sequences. In such cases the *exit* action terminates the guarded sequence and the repetition.

The following example uses the repetition to perform the operation of the previous example on all index tuples.

```
*[  ∃ ρ,ν : [index,ρ]†, [value,ν]† : ρ>0 ≻ (ρ,ν)
[]  ∃ ρ : [index,ρ]† : ρ≤0 ≻ skip
[]  ¬([index,*]) ≻ exit
]
```

The last construct we discuss is the *replication.* Although both selection and repetition allow for concurrent initiation of multiple

transactions, they are essentially sequential constructs since only one guarding transaction is permitted to commit. By contrast, the replication provides for unbounded concurrent execution of transactions. To explain the semantics of the construct, let us consider the following replication which sorts index/value pairs by exchanging the value fields whenever they are not in accordance with the indices.

$$\approx[\quad \exists \iota 1,\iota 2,\nu 1,\nu 2 : [\iota 1,\nu 1]\dagger, [\iota 2,\nu 2]\dagger, \iota 1<\iota 2 : \nu 1>\nu 2 \rightarrow (\iota 1,\nu 2),(\iota 2,\nu 1) \;]$$

The syntax is similar to that for repetition with '$*$' being replaced by the symbol '$\approx$' to be suggestive of parallelism.

Conceptually, we can think of this construct as consisting of an unbounded number of textual copies of each of the transaction sequences that make it up, all executing concurrently. An alternate, operational model is for each sequence to be started concurrently, with every successful execution of a guarding transaction leading to the creation of a finite, but indeterminate number of copies of the transaction sequence. The construct terminates when all generated sequences terminate.

2.4. Process

SDL supports the definition of parameterized process types, henceforth called *process definitions*. Process definitions assume the format

PROCESS *type_name(parameters)*
  IMPORT
    import_definitions
  EXPORT
    export_definitions
  BEHAVIOR
    sequence_of_statements

where a statement is a transaction or a flow-of-control construct. For a given program the set of process definitions is static but processes may be created dynamically as in the transaction

$$\exists \alpha : [\text{year},\alpha] \rightarrow \text{Statistics}(\alpha)$$

where an instance of the process Statistics is started for some year found in the dataspace. Process termination occurs when the last statement is executed or upon execution of the *abort* action in a successful transaction.

Although each process definition is required to have an explicit view specification, in the context of this paper, we will omit it whenever the view covers the entire dataspace, i.e., when it is not restricted in a meaningful way.

3. PROGRAMMING STYLE

To show the difference in style between traditional approaches and the shared dataspace paradigm we present here solutions to three well understood problems: summing the values of an array, accessing and sorting a property list, and region labeling. Our objective is two-fold. First, by taking traditional solutions and recoding them using SDL we hope to show SDL's flexibility and assist the reader in making the transition to the shared dataspace paradigm. Second, by providing typical shared dataspace solutions for the same problems we want to demonstrate the expressive power of the language and the changes in programming style. (Note: To simplify the examples, we sometimes omit the lists of quantified variables since the variables are already distinguished by Greek letters.)

3.1. Array Summation

In the first example we consider an array A of integer values with the index ranging from 1 to $N=2^n$. Parallel summation can be carried out in n phases. In the first phase we add to each even position the value of the preceding odd position and drop from consideration all the odd positions. In the next phase we treat the even positions as if they were an array of length 2^{n-1} and repeat the procedure until we obtain an array of length one containing the summation result.

The algorithm maps equally well on shared-variable or message-based models. Let us consider first a synchronous shared variable solution, as one might use on the Connection Machine[6], and its simulation in SDL. A reasonable initial configuration would be one in which the dataspace D contains one tuple for each array entry $A(k)$, i.e.,

$$D = \{ \langle k,A(k)\rangle \mid 1\leq k\leq N \}$$

and the initial process society is

$$\{ \mathit{Sum1}(k,1) \mid 1\leq k\leq N \text{ and } k \text{ is even} \}$$

where the $\mathit{Sum1}(k,j)$ are given by

PROCESS Sum1(k,j)
  $\exists\, \alpha,\beta : [k-2^{j-1},\alpha]\dagger, [k,\beta]\dagger \dagger\dagger (k,\alpha+\beta)$
  [k **mod** $2^{j+1} = 0$ $\dagger\dagger$ Sum1(k,j+1)
  [] k **mod** $2^{j+1} \neq 0$ $\dagger\dagger$ skip
  ]

The first transaction in *Sum1* replaces two array entries by one containing the sum. The selection construct creates a new *Sum1* process for each of the new array entries. The consensus transaction is used to force synchronous execution of all the processes present in each phase j. (Since we assume that all processes import the entire dataspace the consensus set is represented by the entire process society).

An asynchronous solution can be constructed by creating a process for each array entry still under consideration in each phase and by tagging each piece of data with the phase in which it must be considered. Initially, the dataspace consists of one tuple per entry in the array

$$D = \{ \langle k,A(k),1\rangle \mid 1 \le k \le N \}$$

and the process society is

$$\{ \mathit{Sum2(k,j)} \mid 1 \le j,k \le N \text{ and } k \bmod 2^j = 0 \}$$

where the *Sum2(k,j)* are given by

PROCESS Sum2(k,j)
$\exists\ \alpha,\beta : [k-2^{j-1},\alpha,j]\dagger,\ [k,\beta,j]\dagger \succ (k,\alpha+\beta,j+1)$

The use of the delayed transaction enables each process to wait until all of its data is ready. In the shared variable model the tuples $\langle k,*,j\rangle$ would reuse the same location of an array *A(k,j)* while in a message-based model the tuple $\langle k,*,j\rangle$ would become a message (*) between a process in the phase *(j−1)* and a process in the phase *j*.

Although these two solutions are easily expressed in SDL, they are not as elegant or compact as the program

PROCESS Sum3
$\approx[\quad \exists : [\nu,\alpha]\dagger,\ [\mu,\beta]\dagger : \nu \neq \mu \rightarrow (\mu,\alpha+\beta)\ ]$

which assumes a dataspace like that of the first summing example. This algorithm preserves the essential idea of generating parallel partial sums without introducing the synchronization required by a strict phase-by-phase processing. Furthermore, it leaves undefined the degree of parallelism that is actually present at execution time—it depends upon the availability of computing resources on the particular machine.

We find the third solution preferable; it conveniently expresses the desired computation while imposing minimal control constraints that could potentially limit the concurrency in execution. This approach requires a sophisticated language implementation to provide efficient program execution. The first two solutions, where the programmer supplies more control structure, require less of the language implementation for efficient execution. However, our objective is to ease the task of the programmer in expressing parallel programs, not necessarily the task of the language implementors.

3.2. Property List

The second example involves operations on a property list consisting of property names and values. The property list is structured as a linked list where each node in the list is represented by a four-tuple of the form

<node_id,property_name,value,next_node_id>

The objective here is to show SDL's capacity to deal with distributed data structures.

The first program simulates a recursive traversal of the list in search of a particular property. In place of the normal recursive calls, a new process is created to continue the search. The initial process society consists of one *Search* process with *id* being the first element of the property list and *P* being the property desired.

PROCESS Search(id,P)
[$\exists\ \nu : [id,P,\nu,*] \succ (P,\nu)$
[] $\exists\ \pi : [id,\pi,*,nil] : \pi \neq P \succ (P,not\_found)$
[] $\exists\ \pi,\iota : [id,\pi,*,\iota] : \pi \neq P,\ \iota \neq nil \succ Search(\iota,P)$
]

It is unlikely, however, that the programmer would go to the trouble of simulating the recursion when the language permits one to address data by contents. The preferred solution would take the form

PROCESS Find(P)
[$\exists\ \nu : [*,P,\nu,*] \succ (P,\nu)$
[] $\neg\exists\ \nu : [*,P,\nu,*] \succ (P,not\_found)$
]

Here it is assumed that there is only one property list in the dataspace and that the list is stable (not modified) during the execution of this process. Multiple lists can be handled by tagging each list with an additional unique identifier. Instability, however, cannot be handled as easily. It leads to added implementation complexity and penalties to ensure serializability of transactions. However, it is important to note that views provide a mechanism by which the penalities can be minimized by limiting transactions to small portions of the dataspace.

Before concluding this section, we provide one last example program, sorting the property list by names. This is done by associating a process *Sort(node_id,next_node_id)* with each node in the property list.

PROCESS Sort(node_id,next_node_id)
IMPORT
(node_id,*,*,*); (next_node_id,*,*,*)
EXPORT
(node_id,*,*,*); (next_node_id,*,*,*)
BEHAVIOR
next_node_id = nil → exit
*[$\exists : [\iota1,\pi1,\nu1,\iota2]\dagger,\ [\iota2,\pi2,\nu2,\iota3]\dagger : \pi1 > \pi2$
$\succ (\iota1,\pi2,\nu2,\iota2),\ (\iota2,\pi1,\nu1,\iota3)$
[] $\exists : [\iota1,\pi1,\nu1,\iota2],\ [\iota2,\pi2,\nu2,\iota3] : \pi1 \le \pi2$
†† exit
]

Two interesting features of this program are the formation of process communities by means of

import set overlap and the use of consensus transactions to specify the termination of a distributed computation. Each *Sort* process enters the sorting loop, which is exited only if the consensus transaction is successful. The consensus transaction in each *Sort* process checks for proper ordering between two consecutive nodes in the list. At any point in time when its two nodes are properly ordered, a *Sort* process is willing to participate in a consensus. When all *Sort* processes see ordered entries in the list, the sort is complete; the consensus transaction then takes place with the processes exiting their respective loops.

The consensus transaction, perhaps more than anything else, illustrates the expressive power we want to put in the hands of the programmer. It is a high-level concept; it occurs frequently in programming; it relates closely to some important concurrent language constructs and concepts; and it holds the promise for efficient implementation.

3.3. Region Labeling

Our last example is a common problem in computer vision. After subjecting a digitized image to a threshold operation T, contiguous regions of equal intensity are identified and labeled. Assuming a predicate *neighbor(ρ1,ρ2)* to tell if two pixels are 4-connected, the threshold and labeling operations can be performed by a single process issuing many parallel transactions.

```
PROCESS Threshold_and_label
  ≈[    ∃ : [image,ρ,ν]†
              → (threshold,ρ,T(ν)), (label,ρ,ρ)
      ≈[    ∃ : neighbor(ρ1,ρ2), [label,ρ1,μ]†,
                [label,ρ2,λ], [threshold,ρ1,τ],
                [threshold,ρ2,τ] : μ<λ
                    → (label,ρ1,λ)
      ]
  ]
```

Each instance of the first transaction replaces a pixel at xy-location ρ having intensity ν by a threshold value $T(\nu)$ at the same location. Instances of the second transaction seek out neighboring pixels having the same threshold values but different labels and propagate the label of the largest xy-coordinate covered by the region. The resulting program is representative of the *workers model*, often used in Linda programming, where a number of processes are created and sent out to seek work in the dataspace.

In this program, the labeled regions are not available for further processing until the entire program completes execution. Waiting for all regions to be labeled is often unreasonable, as in the case of an image which results from continuous terrain scanning from an airborne platform. One apparent solution to this problem is to detect the fully labeled regions with a separate process. However, the availability of process views facilitates the formulation of a different, less obvious, solution.

In this region labeling program, the threshold and labeling operations are performed by separate processes. The threshold process applies the threshold operation and then creates a labeling process for each pixel in the image.

```
PROCESS Threshold
  ≈[    ∃ ρ,ν : [image,ρ,ν]†
              → (threshold,ρ,T(ν)), Label(ρ,T(ν))
  ]
```

The actual labeling is performed by the *Label* processes. As shown below, the labeling process first assigns a label r (i.e., its own location) and waits for all its neighbors belonging to the same region to make their own label assignments. Once the import set becomes stable, the sorting loop shown earlier may be used to assign to all pixels in a region the label of the largest coordinate covered by the region. When the labeling is complete in a given region, the threshold values are discarded.

```
PROCESS Label(r,t)
  IMPORT
      ρ, λ : neighbor(ρ,r), [threshold,ρ,t] : [label,ρ,λ]
                :=> (label,ρ,λ)
      ρ : neighbor(ρ,r) : [threshold,ρ,t]
                :=> (threshold,ρ,t)
      ρ, ν : neighbor(ρ,r) : [image,ρ,ν]
                :=> (image,ρ,ν)
  EXPORT
      (label,r,*)
  BEHAVIOR
      (label,r,r) ;
      ¬[image,*,*] ≻ skip ;
      ∀ ρ : [threshold,ρ,*] : [label,ρ,*] ≻ skip
    *[    ∃ μ,ρ,λ : [label,r,μ]†, [label,ρ,λ] : μ<λ
                ≻ (label,r,λ)
      []    ∀ μ,ρ,λ : [label,r,μ], [label,ρ,λ] : μ=λ
                †† exit
      ]
      [threshold,r,*]†
```

In defining the import set for the *Label* process, we took advantage of the fact that SDL allows the view to depend upon the current configuration of the dataspace. At the point where the sorting iteration is entered, for instance, the import set is stable and consists of the threshold and labeling information about those 4-connected neighbors of r that belong to the same region. This ensures that the consensus transaction involves only labeling processes operating over the same region. Consequently, in contrast to the first solution, the labeling program allows the formation of communities of processes which work

asynchronously on some distributed data structure, e.g., a region, and synchronize whenever they believe that a subtask is complete. We could call this style of programming the *community model.* Processes interlocked by the overlap between pairs of import sets form a closed community. Although a process need not wait for the entire community to be formed before starting its work, it must somehow ensure that all its neighbors exist. Otherwise, individual decisions based on incomplete information can undermine the communal objective and lead to premature termination or deadlock on the part of some processes.

Given our limited experience with writing SDL programs, we are not yet in a position to provide a good critical evaluation of the worker and community models. A hybrid approach will probably prove to be the right solution, but this is a topic of current investigation.

4. CONCLUSIONS

Our ultimate goal is to develop the software support needed for the design, analysis, understanding, and testing of programs involving many thousands of concurrent processes running on a highly parallel multiprocessor. No single technical development will solve all the problems associated with what we have come to call *large-scale concurrency.* It is certain, however, that one must consider both novel programming paradigms and innovative exploratory environments. The search for new programming paradigms must be concerned above all with offering *programming convenience* and with encouraging programmers to *maximize program concurrency.* Future environments must provide, among other things, powerful *visualization* capabilities which will assist programmers in understanding the behavior and performance characteristics of the programs they develop—there is no other way for humans to assimilate voluminous information about the continuously changing program state.

Our concern with large-scale concurrency spans both language and visualization issues. However, we believe that the language should come first and the visualization second. For this reason we started our investigation by seeking a programming paradigm appropriate for large-scale concurrency and yet conducive to an elegant solution of the visualization problem. The shared dataspace paradigm is attractive for several reasons. First, it offers a high degree of flexibility in programming through its content-addressable data structures and its decoupling of the control and data states. Its powerful and compact data transformations, especially the replication construct and the consensus transaction, maintain a simple program structure while providing for maximal concurrency during execution. The global dataspace, and its decoupling from the control state, should also provide a solid foundation for reasoning about global system states. Finally, the shared dataspace is the only paradigm we are aware of which elegantly accommodates programmer-defined visualization. Potentially one can create visualization processes completely decoupled from the rest of the process society, yet having complete access to the data state of the computation.

Acknowledgment: This work was supported by the Department of Computer Science, Washington University, St. Louis, Missouri. We thank our colleagues Wei Chen and Ken Cox for their helpful reviews of the manuscript.

5. REFERENCES

1. Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," *COMPUTER* **19**(8) pp. 26-34 (August 1986).
2. 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).
3. Chandy, M. and Misra, J., *Parallel Program Design: A Foundation,* Addison-Wesley, Reading, Massachusetts (1988).
4. Forgy, C. L., *OPS83: User's Manual and Report,* Production Systems Technologies, Inc. (March 1985).
5. Gelernter, D., "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems* **7**(1) pp. 80-112 (January 1985).
6. Hillis, W. D. and Steele, G. L. Jr., "Data Parallel Algorithms," *Communications of the ACM* **29**(12) pp. 1170-1183 (December 1986).
7. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM* **21**(8) pp. 666-677 (August 1978).
8. 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).
9. Roman, G.-C., "Language and Visualization Support for Large-Scale Concurrency," in *Proceedings of the 10th International Conference on Software Engineering,* IEEE, Singapore (April 1988).