Report Number: WUCS-87-20

1987-01-01

Language and Visualization Support for Large-Scale Concurrency

Gruia-Catalin Roman

SDL (Shared Dataspace Language) is a language for writing and visualizing programs consisting of thousands of processes executing on a highly-parallel multiprocessor. SDL is based on a model in which processes use powerful transactions to manipulate abstract views of a virtual, content-addressable data structure called the dataspace. The process society is dynamic and supports varying degrees of process anonymity. The transactions are executed over abstract views of the dataspace. This facilitates elegant conceptualization of dataspace transformations and compact program representation. Processes and transactions enable SDL to combine elements of both large and fine grained concurrency. The view is a... Read complete abstract on page 2.

Language and Visualization Support for Large-Scale Concurrency

Gruia-Catalin Roman

Complete Abstract:

SDL (Shared Dataspace Language) is a language for writing and visualizing programs consisting of thousands of processes executing on a highly-parallel multiprocessor. SDL is based on a model in which processes use powerful transactions to manipulate abstract views of a virtual, content-addressable data structure called the dataspace. The process society is dynamic and supports varying degrees of process anonymity. The transactions are executed over abstract views of the dataspace. This facilitates elegant conceptualization of dataspace transformations and compact program representation. Processes and transactions enable SDL to combine elements of both large and fine grained concurrency. The view is a novel abstraction mechanism whose significance is derived from the fact that it allows processes to interrogate the dataspace at a level of abstraction convenient for the test they are pursuing. The view also plays a role in the definition of continuously updated, programmer-defined visual abstractions which enable exploration of the program's functionality and performance.

LANGUAGE AND VISUALIZATION SUPPORT FOR LARGE-SCALE CONCURRENCY

Gruia-Catalin Roman

WUCS-87-20

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

As appeared in the Proceedings of the 10th International Conference on Software Engineering, April 11-15, 1988, Singapore, pp. 296-308.

LANGUAGE AND VISUALIZATION SUPPORT FOR LARGE-SCALE CONCURRENCY

Gruia-Catalin Roman

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130

ABSTRACT

SDL (*Shared Dataspace Language*) is a language for writing and visualizing programs consisting of thousands of processes executing on a highly-parallel multiprocessor. SDL is based on a model in which *processes* use powerful transactions to manipulate abstract views of a virtual, content-addressable data structure called the *dataspace*. The process society is dynamic and supports varying degrees of process anonymity. The *transactions* are executed over abstract views of the dataspace. This facilitates elegant conceptualization of dataspace transformations and compact program representation. Processes and transactions enable SDL to combine elements of both large and fine grained concurrency. The *view* is a novel abstraction mechanism whose significance is derived from the fact that it allows processes to interrogate the dataspace at a level of abstraction convenient for the task they are pursuing. The view also plays a role in the definition of continuously updated, programmer-defined *visual abstractions* which enable exploration of the program's functionality and performance.

1. INTRODUCTION

In recent years a number of highly parallel computer system architectures have been proposed, for example, hypercubes[21], the Ultracomputer[20], and the Connection Machine[14]. Such machines have appeared commercially and are expected to become commonplace in the future. Hypercube system products are available from companies such as Intel, NCUBE, and Floating Point Systems, the Connection Machine is marketed by Thinking Machines Corporation, and shared-memory multiprocessors are available from companies such as Elxsi, Encore, Sequent, and BBN. Many of these machines (e.g., the hypercubes) consist of general purpose processing elements (e.g., an off-the-shelf microprocessor) interconnected in a regular communications pattern. With such an architecture, VLSI technology holds the promise of putting one or more nodes on a single chip—making systems with thousands of nodes viable. The present costs of these systems make supercomputer power much more available than ever before. The future holds much promise of further decreases in the price/performance ratio.

The raw power offered by this class of machines makes them attractive for systems requiring high performance and reliability. However, large-scale usage of these machines by the current generation of computer programmers is being viewed with great skepticism by concurrent programming experts. The sources of concern are both cultural and technical. 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.

Our ultimate goal is to develop the software support needed for the design, analysis, understanding, and testing of programs involving thousands, or tens of 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 own concern with large-scale concurrency spans both language and visualization issues. Furthermore, 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—our alternative would have been to pursue the visual programming language path. This paper discusses the choices we made so far and the reasons behind them. We start with a brief review of the concurrent programming language field and follow it by arguments in favor of the shared dataspace paradigm. An overview of the language proposal and a discussion of its implications for an approach to visualization come next. We end with some remarks about the appropriateness and feasibility of our language.

2. CONCURRENT LANGUAGE LANDSCAPE

Concurrency has been a long-standing interest for programming language designers. Concurrent programming languages, i.e., languages that provide explicit mechanisms for expressing concurrent execution, may be divided into four broad categories depending on the nature of the inter-process communication mechanisms they employ. *Shared variables* as used in Concurrent Pascal[4], for example, allow processes to communicate by reading and writing (directly or indirectly via monitors) their values. *Message based communication* requires all information sharing to take place by sending and receiving messages in accordance with some predefined protocol. CSP[15] and Actor languages[1] are representative of this category. *Remote operations,* such as the remote procedure call used in Ada[3], permit a process to invoke operations associated with some other process. The parameters and returned values represent the information shared by the processes involved in the exchange. The last category we choose to call *shared dataspace.* It is comprised of 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[18], Linda[2], and some artificial intelligence languages such as OPS83[9] belong here. Gelernter[10] has used *generative communication* to refer to this category but we found the term shared dataspace to be more explicit. It suggests an analogy with the shared variables mechanism and also alludes to the attempt to incorporate database concepts into the programming language.

Large-scale concurrency. Not all models scale up to programs consisting of many thousands of processes. Name management, control and data state coupling, and information transfer bandwidth are potential trouble areas.

Name management. Shared variables, message based communication and remote operations require variable, process, or operation names to establish communication between processes. Such paradigms could be called name-based. For large programs, managing the name space becomes a great programming burden, especially when the names do not contribute to the computation task. Furthermore, in open systems[13] new processes need to pass on their names to the rest of the community; this kind of programming overhead is clearly an artifact of the model and bears no relevance on the solution. The name management problem is less acute in shared dataspace models because anonymous processes can access data directly by its content.

Coupling. Complex program states make program understanding and analysis very difficult. Complexity is most often the result of the interplay between the control and the data state of the program. The shared dataspace paradigm can eliminate most considerations of control state, thus offering the promise of greatly simplified program analyzability. We believe that complete elimination of the control state will prove unrealistic for many practical situations, but, we find the fact that one has the opportunity to limit the complexity of the control component of the state appealing.

Bandwidth. The most compelling reason for adopting a shared dataspace paradigm has to do with the potential for major increases in the bandwidth of information (not data!) transfer. Variables, messages, and parameters, no matter how complex they become, are limited both in their ability to convey information about global aspects of the system state and in the level of abstraction at which they express such information. Unless we allow processes to interrogate the system state at an appropriate level of abstraction, the programmer will be overwhelmed by the mechanics of data communication. The shared dataspace makes access to highly abstract information possible by means of mechanisms we understand well—logical queries. Logical queries have been used extensively in databases and logic programming.

Visualization. As far as visualization is concerned, two distinct approaches seem appropriate: system-defined or programmer-defined visualization. The former is outside the language while the latter is based on constructs included in the language, i.e., it is language-embedded. Most current work places visualization in the runtime support system. If this is the direction of choice, any programming paradigm is suitable because the runtime support system presumably has access to all aspects of the system state. We believe, however, that programmer-defined mechanisms are more general and flexible. The shared dataspace is the only paradigm we are aware of which elegantly accommodates language-embedded visualization. This is because one can conceive of visualization processes completely decoupled from the rest of the process society, yet having access to any information about the computation state.

The considerations discussed above convinced us to investigate shared dataspace languages supporting large-scale concurrency and visualization. Our language proposal, called SDL (Shared Dataspace Language), 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 abstraction mechanism which serves multiple roles. First, it supports an abstract notion of locality. Second, it provides a mechanism for introducing structure into the tuple-based dataspace and for organizing sets of processes into cooperative process societies. Third, it is used to construct complex visual abstractions. Finally, 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.

3. LANGUAGE OVERVIEW

This section introduces the reader to our current thoughts on the SDL design. In SDL concurrent 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 dataspace and 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. By and large, 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 they 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 a few 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, we found one instance where coordinated transaction execution by a set of cooperating processes greatly simplified the program. This novel transaction type, called a *consensus transaction*, is based on a generalization of the well known quiescence detection problem[8].

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 the 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.

In the following sections we introduce the notation used in SDL, give examples, and discuss some methodological implications of our programming paradigm.

3.1. Dataspace

The dataspace D is defined as a finite but large set of tuples with the first position in the tuple being the unique tuple identifier. Formally,

$D \subset I \times V^*$

where I is the set of tuple identifiers (e.g., integers) and V is a domain of values (e.g., atoms and integers). Tuple identifiers encode the identity of the creating process and, for tuples having the same owner, the order in which they were created. The tuple owner may be obtained using **owner**(ι), where $\iota \in I$. Beside being used to support certain visualization features, tuple identifiers allow the environment to order properly the outputs of individual processes.

At the meta level, we will denote tuples as finite sequences of symbols as in

<1723,year,87>

(Because we often present statements outside of the context of a process definition, we will use greek letters for quantified variables, lower case letters and numbers for constants, and upper case letters for named constants).

In SDL, the dataspace membership test assumes the form

1723[year,87]

meaning

$<1723,year,87> \in D$

(Note: A transaction does not see the dataspace but a window. For the time being, however, we will assume that the window is the entire dataspace.)

Since most often the programmer is not interested in the tuple identifier one uses

[year,87]

meaning

$\exists\, \iota : \iota \in I : <\iota,year,87> \in D$[‡]

Furthermore, a do-not-care marker, "*" may be used in any tuple fields which are not of interest to the programmer. For instance, [year,*], *[year,*], and <*,year,*>∈D are all interpreted as

$\exists\, \iota,\nu : \iota \in I,\ \nu \in V : <\iota,year,\nu> \in D$

Some tuples in the dataspace are involved in the interfacing with the input/output environment. Their tuple identifiers encode this fact. The symbols "?" and "!" may be used in the tuple identifier field to indicate that the particular tuple is an input from or output to the environment, respectively. We can use

?[year,*]

to check if a new year has arrived. However, a test of the form

![year,*]

is disallowed in order to permit the output environment to retract all program output without interference from the process society. In other words, we treat the environment as two distinguished processes that exist outside the process society making up the program. The input environment may only assert input tuples while the output environment may only retract output tuples. The process society, on the other hand, may not assert input tuples and may not retract or test output tuples.

3.2. Basic Transactions

Dataspace membership testing, tuple assertion, and tuple retraction are the simplest SDL transactions. The membership test we have met already in the previous section

[year,87]

This transaction is evaluated only once and may 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 <*,year,87> may leave in the dataspace other instances having different tuple identifiers.

More complex queries may be formed by composing predicates using negation ("¬"), conjunction (","), disjunction ("or"), and parentheses. In such cases, any membership test with a retraction tag must be evaluated (even if its truth value does not affect the result of the query!). For instance, in the transaction

[year,87]† or [year,86]†

even if the first membership test turns out to be successful, the second test must also be performed because the presence of <*,year,86> in the dataspace must trigger its retraction.

To assert a tuple one can use a transaction such as

‡ All logical expressions used in this discussion consist of three parts separated by colons: a list of quantified variables, domain restrictions on the values the variables may assume, and a predicate. The separation between the domain and the predicate is often only aesthetic and sometimes is omitted along with the second colon.

(year,87)

Output tuples are indicated by the "!" symbol

!(year,87)

and, although they are part of the dataspace and are owned by the process executing this transaction, they are not accessible by the process society.

The most common transaction format is

transaction ::=
    query
    transaction_type_tag
    action_list

query ::=
    quantifier
    variable_list :
    binding_query :
    test_query

quantifier ::=
    $\forall \mid \exists \mid \Sigma$

transaction_type_tag ::=
    $\rightarrow \mid \succ \mid \uparrow\uparrow$

The transaction_type_tag[‡] determines some of the operational characteristics of the transaction.

Immediate transactions are tagged by "$\rightarrow$" as in the transaction

$\exists\ \alpha$: [year,α]† : α>87 $\rightarrow$ let N=α, (found,α)

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 <1212,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 <1212,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 <id,found,90> is added to the dataspace. (The function id generates new tuple identifiers.) 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 third position.

If we want to check that all tuples <*,year,*> have a value greater than 87 in the third field, we may use a universal quantifier. The variable α, which is left unbound by the query evaluation, may no longer appear in the action list

$\forall\ \alpha$: [year,α] : α>87 $\rightarrow$ (ok)

Note that we have omitted the dagger so as not to remove all tuples satisfying the query. Also note that, if there are no tuples of the form <*,year,*>, the query is successful by definition. With a little bit more notation we can also find out how many tuples were involved in the above successful query

let N=($\Sigma\ \alpha$: [year,α], α>87 : 1) $\rightarrow$ (ok)

For every successful binding of α a value of 1 is added, and the final result is used in the definition of N. A summation query always succeeds. Other specialized queries such as MIN and MAX will also be included in the language.

Delayed transactions, tagged by "$\succ$", differ from immediate transactions in that, instead of failing, they block the process until a successful evaluation is possible. A delayed transaction such as

$\exists\ \alpha$: [year,α] : α>87 $\succ$ (new_year)

is not executed until the dataspace contains a tuple of the form <*,year,*> with the third 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 transaction is issued and the dataspace does not prevent it from executing, the transaction is eventually executed.

Other kinds of transaction types are currently under investigation, in particular, transactions that support visualization requirements. However, we are not ready to discuss them in any detail at this time.

3.3. 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 may be listed on separate lines or on the same line separated by semicolons. The execution of one transaction must complete before the next one is initiated. In the sequence shown below one array index and a value, both supplied independently by the environment, are paired at random and placed in the dataspace

$\exists\ \rho$: ?[index,ρ]† $\succ$ let X=ρ
$\exists\ \nu$: ?[value,ν]† $\succ$ 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

‡ There are currently three transaction types in the language. Other types, which would be denoted by different symbols, are under investigation.

by "#". These sequences are called *guarded sequences.* In this and all the constructs discussed below, flow of control constructs may appear in sequences anywhere transactions may except at the head of a guarded sequence. (Henceforth, we will take the liberty of using the term sequence of transactions but assume that some of the transactions might be flow of control constructs.) 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 all guarding transactions fail the construct is terminated without executing any of the sequences. No guarding transactions may be evaluated more than once. Since delayed transactions cannot fail, a selection involving only delayed transactions will block until one of the guarding transactions succeeds. This is the case in our example below where either we pair a value with a positive index or we retract a non-positive index

```
[    ∃ 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. The exit from the repetition is 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 inhibits the repetition.

Using the repetition we can now read all input provided by the environment. We take this opportunity to simplify the earlier coding of transactions

```
[ * ∃ ρ,ν : ?[index,ρ]†, ?[value,ν]† :
        ρ>0 ≻ (ρ,ν)
  # ∃ ρ : ?[index,ρ]† : ρ≤0 ≻ skip
  # ¬(?[*,*]) ≻ 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 in which the index/value pairs are sorted by exchanging the value fields of tuples whenever the relation between values is inconsistent with that between indices

```
[ ≈ ∃ρ1,ν1,ρ2,ν2 :
    [ρ1,ν1]†, [ρ2,ν2]†, ρ1<ρ2 :
    ν1>ν2 →
      (ρ1,ν2),(ρ2,ν1) ]
```

The syntax is similar to that for repetition with "*" being replaced by the symbol "≈" which is 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 model is to think that each sequence is started concurrently and that every successful execution of a guarding transaction leads to the creation of a finite but indeterminate number of copies of the entire sequence. The construct terminates when all generated sequences terminate. Using the latter we can return now to the sorting program. The replication contains a single sequence consisting of a single transaction we will call τ. τ searches for two pairs which are out of order. If it finds one such instance, it retracts the two pairs, asserts them back with the value fields exchanged, and terminates successfully. Its successful termination, however, leads to the creation of several copies of τ which execute concurrently. Each copy exhibits the same behavior. When all pairs are sorted, an arbitrary number of copies of τ will still exist in the system. As they discover that no unordered pairs exist they fail one by one until none are left. At that point the replication terminates.

Since delayed transactions never fail, if used in a replication they continue to exist forever preventing the replication from terminating. Under these circumstances, the *exit* action issued by some transaction in one of the concurrent sequences may be used to terminate the construct by aborting all sequences that have not passed their respective guarding transactions.

3.4. Process

SDL supports the definition of parameterized process types, henceforth called *process definitions.* Ignoring process views which are discussed in the next section, process definitions assume the following format:

```
PROCESS type_name(parameters)
    ...
    BEHAVIOR
        sequence_of_statements
```

where a statement is a transaction or a flow of control construct.

A process that computes the factorial could be written as

```
PROCESS Factorial(FID)
  ...
  BEHAVIOR
      ∃ ν : [FID,factorial_request,ν]† :
          ν>0 ≻ let N=ν
      [ ≈ ∃ μ : 1≤μ≤N : ¬[FID,μ] →
            (FID,μ) ]
      [ ≈ ∃ μ1,μ2,ι1,ι2 :
          ι1[FID,μ1]†, ι2[FID,μ2]† :
          ι1≠ι2 → (FID,μ1×μ2) ]
      ∃ μ : [FID,μ]† →
            (FID,factorial_result,μ)
```

where the parameter FID is used to prevent interference between multiple instances of factorial processes executing concurrently.

Computing 5!, for instance, requires the assertion of an appropriate tuple and the creation of a factorial process. This can be done in a single transaction composed of several actions:

```
let FID=new, Factorial(FID),
(FID,factorial_request,5)
```

A unique tag FID is obtained using the built-in function **new**, a process of type Factorial is created using the value FID as parameter, and a tuple requesting the computation is asserted. At the termination of the factorial type process above the tuple <*,FID,factorial_result,120> will be left in the dataspace. Process termination occurs when the last statement is executed or upon execution of the *abort* action in a successful transaction.

Although SDL is designed to support anonymous processing, name-based communication may be simulated. Unique tags, which the system makes available through the built-in function **new**, may be used to accomplish process-to-process interactions. Consider a process that needs to acquire a server from a pool of servers. The server may assert a tuple indicating its availability and a unique tag to be used in requests directed to the server. After that it waits for work requests

```
let SID=new
(free,SID)
[request,SID]† ≻ skip
```

The process needing a server, in turn, seeks a free server. When a server becomes available, it is acquired and the unique tag is remembered for subsequent direct interactions.

```
∃ σ : [free,σ]† ≻ let SID=σ
(request,SID)
```

Continuations (i.e., tagging results for use by other than the service requester) may also be programmed in the same manner.

3.5. View

In our discussion so far we have assumed that each transaction τ has access to the entire dataspace D. We can think of τ as a function which, given D, returns two sets of tuples: the *retraction set* Dr (tuples to be retracted) and *assertion set* Da (tuples to be asserted). After computing Dr and Da, D' (the new dataspace configuration) may be computed by performing all retractions and all assertions in this order

$$(Dr, Da) = \tau(D)$$
$$D' = ((D - Dr) + Da)$$

(Note: Set union and difference are represented by "+" and "−".)

The view associated with a particular process restricts its transactions from operating on the dataspace directly. Invisible to the transaction, the dataspace is substituted by a window W on which the transaction τ acts as before. The transaction computes a *retraction window* Wr and an *assertion window* Wa which are used to update the current window

$$(Wr, Wa) = \tau(W)$$
$$W' = ((W - Wr) + Wa)$$

The window exists only during the execution of the transaction. Tests such as [year,87], assertions such as (year,87), and retractions such as [year,87]† are relative to W, not to D.

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. The view is characterized by two functions called *Import* and *Export*. Given a particular dataspace configuration D, *Import* computes the window W and a *retraction function* ψ which maps tuples in the window to sets of tuples in the dataspace. The role of ψ is to trace retractions in the window back to retractions in the dataspace. The mapping of assertions in the window to assertions in the dataspace is controlled by *Export*. Each tuple asserted by the process is mapped into one or more tuples to be added to the dataspace. Therefore, the new dataspace configuration is computed as follows

$$(W, \psi) = \mathit{Import}(D)$$
$$\text{where } \psi : W \rightarrow \mathit{Powerset}(D)$$

$$(Wr, Wa) = \tau(W)$$

$$W' = ((W - Wr) + Wa)$$

$$D' = ((D - \psi(Wr)) + \mathit{Export}(Wa))$$

In SDL, every process has an explicit view. A language implementation may use the explicit view to achieve performance improvements and carry out compilation-time checks. We are interested, however, in the view concept as a novel programming construct and in the manner by which it helps the programmer deal with some of the difficulties associated with large-scale concurrency. SDL provides support for programmer-defined views. The process type definition may be augmented to include the view

```
PROCESS type_name(parameters)

    IMPORT
        import_definitions

    EXPORT
        export_definitions

    BEHAVIOR
        sequence_of_statements
```

The definition of import and export is specified using *import/export statements* having the following general syntax

```
import_statement ::=
    variable_list :
    binding_query :
    transfer_tuples :=>
    target_tuple

export_statement ::=
    variable_list :
    binding_query :
    transfer_tuple :=>
    target_tuples
```

If the relation between a transfer tuple and the target tuple is identity, only the target tuple is listed using "*" and constants in the appropriate fields. If the second colon is omitted, the transfer tuples are assumed to appear in the binding_query. Semicolons may be used to separate several statements on a single line.

Operationally, the import statement may be viewed as consisting of an existentially quantified query over the dataspace which is evaluated repeatedly until no new variable bindings, i.e., matches, are found. For each successful match, a tuple is assigned a tuple identifier and is added to the window. All the transfer tuples (in the dataspace) that contributed to the creation of a particular target tuple (in the window) will be deleted if the respective target tuple is deleted.

For instance, given the dataspace

$$D = \{ <*,A,6>, <*,A,8>, \\ <*,B,1>, <*,B,6>, <*,B,7> \}$$

and the import statement

$$\alpha : [B,\alpha], [B,\alpha+1] : [A,\alpha] :=> (\alpha),$$

the window is defined as

$$W = \{ <*,6> \},$$

and, after executing the transaction

$$\exists\ \alpha : [\alpha]\dagger : \alpha>3 \rightarrow \text{skip},$$

the dataspace becomes

$$D = \{ <*,A,8>, \\ <*,B,1>, <*,B,6>, <*,B,7> \}$$

While on import several transfer tuples may be mapped to a single target tuple in the window, on export a single transfer tuple in the window may be mapped to several target tuples in the dataspace. These restrictions are consistent with the definition of ψ and *Export* and simplify the implementation. Moreover, they are consistent with the notion that the view is an abstraction mechanism which hides low-level details in which the process is not interested.

Having discussed the syntax and semantics of the basic language constructs used in defining the process view, we turn now to consideration of the pragmatics of the view concept. In particular, we would like to illustrate the view's ability to induce structure in both the dataspace and the process society.

There are two interesting ways of structuring the dataspace. The obvious way is to think of the dataspace as a collection of disjoint subspaces with processes divided into groups depending on which dataspaces they examine and modify. Partition identification tags may be placed into each tuple and the views may use them to indicate the relation between processes and subspaces. Dataflow graphs and Petri nets, for instance, could be simulated in this manner.

Another approach involves a dual structuring of the dataspace and the process society into *domains* and *communities*, respectively. A community of processes is a set of processes defined by taking the transitive closure of the relation *import set overlap*, i.e., "the import set of p overlaps the import set of q." (The *import set* is defined as ψ(W).) Similarly, the domain is constructed by the union of the import sets of a process community. As an illustration, let us consider an open polygon made of a finite set of distinct points in 2-D space having strictly positive coordinates. Their relative positions in the polygon are given by a symmetric and non-reflexive binary relation called *neighbors*. We will assume that exactly two points have a single neighbor (the end points) and that all the other points have exactly two distinct neighbors. Associated with each point there is a process of type Node which takes two parameters: the point coordinates P and the number of neighbors N. The processes cooperate in choosing one of the endpoints as master. The community is formed by allowing each process to import tuples involving its point and the neighboring points.

Initially the endpoints vote for themselves while the inner points vote for a nonexisting point (0,0). The endpoint votes are propagated in opposite directions until the larger coordinate reaches the other endpoint. At this time, the loser declares the other endpoint to be the winner.

```
PROCESS Node(P,N)

    IMPORT
        α,λ : neighbors(α,P), [node,α,vote,λ] :=>
            (he_votes,λ)
        λ : [node,P,vote,λ] :=> (i_vote,λ)
        (master,*)

    EXPORT
        λ : [i_vote,λ] :=> (node,P,vote,λ)
        (master,*)

    BEHAVIOR
        [    N=1 → (i_vote,P)
             [    ∃ λ1,λ2 :
                       [i_vote,λ1]†, [he_votes,λ2] :
                       λ1<λ2 ≻
                         (i_vote,λ2), (master,λ2)
               # [master,*]≻ skip
             ]
          #  N=2 → (i_vote,(0,0))
             [ * ∃ λ1,λ2 :
                       [i_vote,λ1]†, [he_votes,λ2] :
                       λ1<λ2 ≻
                         (i_vote,λ2)
               # [master,*]≻ exit
             ]
        ]
```

The above example also illustrates the manner in which the abstraction power of the view hides some of the details of the dataspace definition, thus leading to more compact and elegant programs. Two other examples of useful abstractions are given below in terms of two import statements used in processes that need to know about the polygon used earlier. First, we consider a process interested only in the endpoints of the polygon. The import statement looks as follows:

```
α1,α2,β1,β2,γ1,γ2, :
     [node,α1,vote,*], [node,α2,vote,*] :
       ¬(neighbors(α1,β1),
         neighbors(α1,γ1), β1≠γ1),
       ¬(neighbors(α2,β2),
         neighbors(α2,γ2), β2≠γ2) ,
           α1≠α2 :=> (polygon,α1,α2)
```

Second, we consider a process interested in the convex hull of the same polygon. We assume that for every three points in the polygon we have constructed a clockwise triangle stored in the dataspace as tuples of the form <*,triangle,*,*,*>. The import statement looks identical to Rem's associon solution[18].

```
α, β : [triangle,α,β,*], ¬[triangle,α,*,β] :=>
    (hull,α,β)
```

Finally, before concluding this section we want to comment on the relation between the abstraction mechanism embodied by the view and performance monitoring. As indicated earlier, we treat the input/output environment as processes. The same could be done for the runtime environment (hardware and software) supporting the execution of the program. For this reason, although we have not done it yet, we plan to provide for built-in functions which allow one to request the inclusion in the process window of performance data regarding individual processes and physical resources involved in the program execution. This scheme will allow a single, elegant abstraction mechanism, the view, to support debugging and visualization of both functional and performance properties, separately and together. The practical significance of such a capability cannot be underestimated as highly parallel multiprocessors become common components of systems demanding high performance and reliability.

3.6. Consensus Transaction

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. These kind of situation occurs frequently in concurrent programs and in implementations of concurrent languages. Program termination in the UNITY model[7], 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 is nothing more than a special case of the more general notion of consensus.

Our solution to consensus-type problems is to provide in SDL a specialized and powerful transaction type called a *consensus transaction*. In its simplest form, the consensus transaction may be thought of as an explicit n-way synchronization among processes that are members of the same process community (defined earlier). Since communities are formed dynamically, through changes in the import set overlap relation among processes, the parties participating in the consensus must be determined during program execution. This makes the consensus transaction different from other forms of consensus considered in the literature[8].

A consensus involves the coordinated execution of a set of transactions issued by processes that make up a consensus set. To define *coordinated execution*, let us consider a finite set of transactions T and a dataspace configuration D. If some transaction τ_i in T is executed alone, the new dataspace configuration D' would be computed as before by determining the corresponding assertion and retraction sets Da_i and Dr_i. In the case of a coordinated execution, D' is computed by doing the retractions specified by each Da_i followed by the assertions specified by each Dr_i.

The transactions participating in the coordinated execution are determined by the processes that make up the consensus set. A *consensus set* C is a set of processes satisfying the following properties:

(1) *Willingness to reach consensus is explicit.* Each process p_i in C is ready to execute successfully a transaction τ_i tagged as a consensus transaction.

(2) *All requests for participation by others in consensus are satisfied.* Every process p_j whose import set overlaps tuples examined by τ_i during its successful execution is included in C.

(3) *Participation requests are symmetric.* For any two processes p_i and p_j in C, if the tuples examined by τ_i overlap the import set of p_j, then the tuples examined by τ_j must overlap the import set of p_i.

(4) *All participants are needed.* The set C is closed under the transitive closure of the relation "p_i and p_j requested each other's participation."

Syntactically, consensus transactions are tagged by "↑↑." Semantically, outside of their participation in the consensus, they act like delayed transactions in the sense that they block if a consensus may not be reached. To illustrate the use of consensus transactions and the kind of compact process definitions they encourage, we will consider once more the problem of selecting a leader in a polygon. This time, however, we will eliminate the restriction that the polygon may not be closed. This will show how a significantly more complex problem leads to a simpler solution when we take advantage of the power of the consensus transactions.

```
PROCESS Node(P,N)

    IMPORT
        α,λ : :
            neighbors(α,P), [node,α,vote,λ] :=>
              (he_votes,λ)
        λ : : [node,P,vote,λ] :=> (i_vote,λ)

    EXPORT
        λ : : [i_vote,λ] :=> (node,P,vote,λ)
        (master,P)

    BEHAVIOR
        (i_vote,P)
        (Σ λ : [he_votes,λ] : 1)=N ≻ skip
        [ * ∃ λ1, λ2 :
                [i_vote,λ1]†, [he_votes,λ2] :
                λ1<λ2 ≻
                  (i_vote,λ2)
          # ∀ λ1, λ2 :
                [i_vote,λ1], [he_votes,λ2] :
                λ1=λ2 ↑↑
                  exit
        ]
        [i_vote,P] → (master,P)
```

Each node starts by voting for itself after which it waits for all its neighbors to cast their vote. When this happens, the node enters the voting loop which is exited only if the consensus transaction is successful. The consensus transaction in each node checks for agreement between its vote and the votes of the neighbors. If all votes are the same, the node is willing to participate in a consensus and requests the participation of all its neighbors. When all nodes forming the polygon see agreement with their neighbors, all votes are for the same master and the consensus is reached. Upon exiting, the winning node declares itself a master.

The consensus transaction, maybe more than anything else, is illustrative of 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. Preliminary analysis indicates the potential for implementations whose time complexities are linear in the number of processes forming the consensus set.

4. VISUALIZATION

Previous sections have been concerned with the issue of convenient representation of large-scale concurrent programs. This section deals with a visual approach to program understanding. The interest in visualization is motivated by the programmer's need, during testing and debugging, for rapid assimilation of large amounts of information concerning the continuously changing program state. The high bandwidth required to accomplish the information transfer makes the use of high-resolution graphics a necessity.

In recent years, work on visualization has become an important endeavor in the computer science community; our group alone has been involved in three separate projects where visualization played a key role. Three research directions seem to dominate the field: visual programming, program visualization, and data visualization. *Visual programming* is concerned with the development of two-dimensional, icon-driven programming languages for ease of understanding and of maintenance. Show and Tell[16], Pict[11], and Kimura's Transaction Network[17] are languages that fall into this category. *Program visualization* is concerned with the development of visual environments in which program structure and program execution can be displayed graphically. PV[5] and BALSA[6] are representative examples for this kind of work. Finally, there is significant investment being expended in the area of *data visualization*. Research efforts, such as the spatial management of data at CCA[12] and our own study of geographic data processing requirements[19] are interested in the display of large data sets for easy understanding of relationships among data entities and for easy browsing.

Relative to the above taxonomy, the SDL debugging/testing environment falls into the program visualization category. However, the reliance on the shared dataspace as a means for decoupling the application and visualization processing leads us to incorporate elements of data visualization methodologies. An important feature of our approach is the emphasis on programmer defined visualizations. The motivation behind this strategy is the realization that any predefined program visualization, by necessity, is in terms of programming language constructs and, therefore, is unable to capture the semantics of the application. When processing a matrix of elevation values, for instance, it is much more meaningful to see a three-dimensional surface rather than an array of numbers.

In SDL, *visual abstraction*, i.e., the graphic display of an abstract representation of the program state, is logically decomposed into two parts: abstraction and rendering. As before, the abstraction mechanism is the view. Its role, in this case, is to map the state of the dataspace to a set of tuples which can be interpreted by built-in device-dependent rendering rules and converted to images on a display device. The programmer can output tuples to a visualization device in a manner similar to the input and output interactions presented earlier. Visualization processes can also be defined, with their execution controlled by the programmer. Syntactically, a visualization process takes the form

```
PROCESS type_name(parameters)

    IMPORT FOR virtual_dev_name(parameters)
        mapping_to_dev_dependent_window
```

By convention, a visualization process may be started by a transaction such as

```
let VID=new,
See_Voting_in_Action(VID), (VID)
```

and may be stopped by using

```
[VID]†
```

which signals the visualization process to terminate.

As an example, let us consider the visualization of the earlier leader selection program on a graphics device capable of displaying lines and points of specified colors. We may want to use a visualization process which displays the polygon as white lines and encodes the vote of each node as a color other than white. When a master node is selected all other nodes turn white. The definition of such a process might look as follows

```
PROCESS See_Voting_in_Action(VID)

    IMPORT FOR Raster_screen_device(VID)

        α,β : [node,α,vote,*], [node,β,vote,*] :
            neighbors(α,β), α<β:=>
              (line,white,α,β)
        α,λ,β : [node,α,vote,λ], ¬ [master,β] :=>
              (point,colorize(λ),α)
        α,λ,β : [node,α,vote,λ], [master,β] :
            α≠β :=> (point,white,α)
```

The process behavior is implicit. The "FOR" clause on import indicates that the behavior of this process is defined externally by a handler specific to the particular virtual device. Moreover, the window is restricted to containing only tuples recognized by the particular device. Changes in the dataspace are automatically reflected in the information presented on the screen. The programmer can interact with the display by making the view a function of certain control tuples whose assertion and retraction would change the contents of the window.

In Figure 1 we show an image depicting one state in the execution of a complex program. (The image is a black and white reproduction of a single frame from a 10 minute videotape containing a complete visualization of a simulated program execution. The visualization uses color, location, and geometric symbols to encode the state of the program.) Although the visualization has been

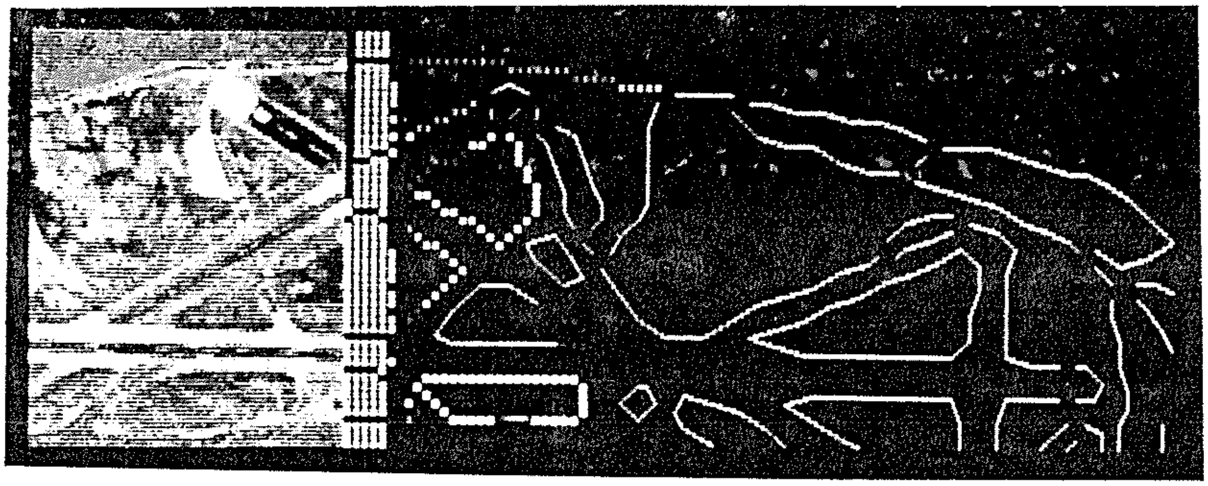
Ground truth
Input processes
Voting and line building processes
Processes ready for output

Figure 1: Visualization of a sample complex program execution.

simulated, it is consistent with the results that could have been obtained by executing a visualization process similar to the one above. The program assumes an airborne platform which scans the terrain below (an airport) constructing an image unbounded on one side. A hardware edge detector transforms the incoming image into a binary edge image (red and blue pixels in the videotape) which the program converts to a symbolic representation as polygons (red lines in the videotape). Non-edge points and edge points having three or more neighboring edge points are eliminated. As a preliminary step to generating the polygons, a version of the voting program discussed earlier is used to select a master in chains of two-way connected edge points (in the videotape, each pixel's vote is color-coded). The master defines the starting point for the polygon construction along each chain.

The same visualization mechanism may also be used to display performance data about the executing processes and the underlying hardware resources. We plan to provide for this by means of specialized built-in functions which may be used to import/export performance data. The process identifiers are the principal means for accessing the performance data and other information about executing processes. Hardware resources will be abstracted as processes and treated in the same manner as the processes making up the application program. The emphasis on visual abstractions of both functional and performance properties of concurrent systems will promote the integration of functional and performance considerations into the design methodology and will enable a better understanding of programs and algorithms for which we lack formal analysis tools.

5. CONCLUSIONS

The brief history of the computer science field has shown that progress in the software arena has been associated with increases in the level of abstraction at which we reason about and write programs. Our language and visualization proposals represent another step in this direction. The view introduces the notion of relativistic abstraction in the programming language by allowing individual processes to see the same dataspace at a level of abstraction appropriate for that process. The transaction is a powerful data transformation executed at a level of abstraction defined by the process view. Visual abstractions are employed in debugging, analysis, and testing.

Technological advances in the areas of highly parallel multiprocessors and graphic workstations have clearly led us to the choices we have made. The ever increasing computing power of the processors makes it possible to support the large communities of hidden processes necessary for maintaining the views and executing the powerful transactions. The quality of the graphics displays will enable us to render complex and dynamic visual abstractions of large process societies and dataspaces. The same technology, however, allows us to go only so far. Higher levels of abstraction, while conceptually attainable, would come with significant performance penalties. It is our conviction that the direction we have taken represents the middle ground between lofty ideals and engineering practice, lofty enough to advance the state of the art and realistic enough to find its way into common usage in the not too distant future.

Acknowledgments: This work was supported by the Department of Computer Science, Washington University, Saint Louis, Missouri. H. C. Cunningham, M. E. Ehlers, and T. D. Kimura have been instrumental in the shaping of these ideas. W. Chen helped formulate the semantics of the consensus transaction. K. C. Cox developed the software for the simulated execution of a large-scale concurrent program and, together with A. F. Laine, participated in the videotape production. R. H. Lykins reviewed and edited the manuscript. Their help and the support of the department chair, J. R. Cox, is gratefully acknowledged.

6. REFERENCES

1. Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press, Cambridge, Massachusetts (1986).
2. Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," *COMPUTER* 19(8) pp. 26-34 (August 1986).
3. ANSI, Inc., "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A-1983, American National Standards Institute, Inc., Washington, D.C. (January 1983).
4. BrinchHansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* 1(2) pp. 199-206 (1975).
5. Brown, G. P., Carling, R. T., Kramlich, D. A., Herot, C. F., and Souza, P., "Program Visualization: Graphical Support for Software Development," *Computer* 18(8) pp. 27-35 (August 1985).
6. Brown, M. H. and Sedgewick, R., "A System for Algorithm Animation," *Computer Graphics* 18(3) pp. 177-186 (July 1984).
7. Chandy, M., "Concurrent Programming for the Masses (1984 Invited Address)," pp. 1-12 in *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing,* ACM, New York (1985).
8. 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).
9. Forgy, C. L., *OPS83: User's Manual and Report,* Production Systems Technologies, Inc. (March 1985).
10. Gelernter, D., "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems* 7(1) pp. 80-112 (January 1985).
11. Glinert, E. P. and Tanimoto, S. L., "Pict: An Interactive Graphical Programming Environment," *Computer* 17(11) pp. 7-25 (November 1984).
12. Herot, C. F., "Spatial Management of Data," *ACM Transactions on Database Systems* 5(4) pp. 493-513 (December 1980).
13. Hewitt, C. and deJong, P., "Open Systems," pp. 147-164 in *On Conceptual Modelling,* ed. M. L. Brodie, J. Mylopoulos, and J. W. Schmidt,Springer-Verlag, New York (1984).
14. Hillis, W. D., *The Connection Machine,* MIT Press, Cambridge, Massachusetts (1985).
15. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM* 21(8) pp. 666-677 (August 1978).
16. Kimura, T. D., Choi, J. W., and Mack, J. M., "A Visual Language for Keyboardless Programming," Technical Report WUCS-86-8, Washington University, Department of Computer Science, St. Louis, Missouri (June 1986).
17. Kimura, T. D., "Visual Programming by Transaction Network," pp. 648-654 in *Proceedings of 21st Hawaii International Conference on System Sciences,* IEEE, Kona, Hawaii (January 1988).
18. 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).
19. Roman, G.-C., "Formal Specifications of Geographic Data Processing Requirements," pp. 434-446 in *Proceedings of the 2nd International Conference on Data Engineering,* IEEE, New York (February 1986). Outstanding Paper Award.
20. Schwartz, J. T., "Ultracomputers," *ACM Transactions on Programming Languages and Systems* 2(4) pp. 484-521 (October 1980).
21. Seitz, C. L., "The Cosmic Cube," *Communications of the ACM* 28(1) pp. 22-33 (January 1985).