DESIGNING INTERDISCIPLINARY APPROACHES TO PROBLEM SOLVING INTO COMPUTER LANGUAGES

Daniel E. Cooke
Computer Science Department
Texas Tech University, Lubbock, Texas

Vladik Kreinovich
Computer Science Department
University of Texas at El Paso, Texas

Joseph E. Urban
Computer Science and Engineering Department
Arizona State University,

Many interdisciplinary design efforts require the involvement of computer scientists because of the complexity of the problem solving tools available for the projects. This paper demonstrates how appropriate language design can place high level languages in the hands of scientists and engineers, thus providing a more automated approach to problem solving that may reduce the amount of computer scientist involvement. The language SequenceL *serves as an example of this approach.*

1. Introduction

There is an ongoing discussion at the highest levels of the United States government concerning “data morgues.” The concern has to do with the current hardware capabilities that permit the acquisition and storage of vast amounts of data and the inability of scientists armed with current software technology to process and analyze the data. The current problem actually demonstrates that advances in computer software have not kept pace with advances in computer hardware. If corresponding software advances can be made, the data may be found to be comatose, rather than dead.

Among other root problems, the comatose data is symptomatic of the fact that currently available software technology is not based upon abstractions that appropriately simplify approaches to complex problems and data sets, especially when the data sets are distributed among multiple processing elements. If large data sets containing, e.g., telemetry data, are to be analyzed, then exploratory or data mining programs must be written. When written in traditional computer languages, these programs require scientists to work with computer specialists and, therefore, much time is needed to deploy application programs. Higher level languages could well support this activity - particularly languages providing abstractions that scientists could employ without the assistance of specialists. If more exploratory programs can be written in a reduced amount of time, more of the comatose data can be analyzed. Furthermore, new abstractions may provide new points of view. The differing points of view may lead to new insights into how the data can best be analyzed.

At the root of any technical solution to the comatose data problem will be a computer language. The higher the level of the root language, the faster the technical solutions will be found. The subject of this paper is a language called *SequenceL*. The *SequenceL* effort is part of a NASA University Research Center at UTEP and has led to commercial interest in the language.

The remainder of this paper describes the need for abstraction improvement in general, and then focuses on the abstraction afforded by *SequenceL.*

2. The Need for New Abstractions

Hardware improvements and the general spread of computing and computer applications have created opportunities for scientists and engineers to solve ever more complicated problems. However, there are concerns about whether scientists and engineers possess the software tools necessary to solve these problems and what computer scientists can do to help the situation.

The fundamental software tool for problem solving is the programming language. A programming language provides the abstraction employed in solving problems. In order to keep pace with hardware improvements, computer scientists should continually address the problem of language abstraction improvement. When advances in hardware make problems technically feasible to solve, there should be corresponding language abstraction improvements to make problems *humanly feasible* to solve.

In the recent past, most language studies have resulted in the addition of new features to existing language abstractions. The most significant changes have resulted in additions to language facilities for the definition of program and data structures. These changes have primarily taken place to accommodate the needs for concurrent execution and software reuse. Although it is important to add to the existing abstractions to satisfy immediate technical problems, research also needs to be undertaken to simplify and minimize existing abstractions.

There are application domains where the need for simpler language abstractions is of vital importance. There are estimates that less than 1% of the available satellite data has been analyzed. (Cooke, 1996) There exists the ability to acquire and store the data, but weakness in the ability to determine information content. Soon NASA will have satellites in place that, in sum, will produce a terabyte of data per day. A major problem associated with the analysis of the data sets is the time needed to write the medium-to-small programs to explore the data for segments containing information pertinent to particular earth science problems. Software productivity gains in developing exploratory programs will allow earth scientists to better grapple with the complexity and enormity of satellite and seismic data sets. Software productivity gains can be accrued through languages developed out of foundational research focusing on language design. The goal of the *SequenceL* language design is to allow scientists and engineers to solve complicated problems in a more intuitive way and to have the solutions imply concurrent algorithms that solve the target problems.

3. The SequenceL Approach

SequenceL was introduced as an approach to software development that offers a different, and for some, a more intuitive approach to problem solving. (Cooke, 1993), (Cooke, Demirors, Demirors, Gates, Kraemer, and Tanik, 1996), (Cooke, 1996), (Cooke, 1998), (Cooke, Daniel, 1998), (Cooke and Urban, 1998), (Cooke and Andersen, 2000) The assumption underlying the design of *SequenceL* is that the data product, as produced by software, is the true product of the software developer. Using traditional languages, programmers write explicit algorithms that imply data products. The goal of the *SequenceL* design effort was to provide a language in which specifiers make an explicit statement of the data product, which in turn implies the algorithm. Algorithm-developers must come to know and understand the implied data product. A data-product specifier need not know the implied algorithm. In *SequenceL,* focus turns from the process that produces the product to the product itself.

One of the main difficulties in traditional programming is grasping the true nature of the implied data product. Implied items are elusive and often require a large amount of concentration to fully grasp. The effort to gain an understanding of the data product impedes productivity. Complex data products are typically defined recursively or iteratively. Software engineers have long realized that the construction of loops is complex and costly. (Mills and Linger, 1986) Bishop noted that "Since Pratt's paper on the design of loop control structures was published more than a decade ago, there has been continued interest in the need to provide better language features for iteration." (Bishop, 1990)

SequenceL possesses no iterative constructs and accommodates a unique form of recursion where functions may embed themselves among intermediate data results. *SequenceL* is a language for describing a data product in terms of both form and content. The difference between the traditional approach to programming and the *SequenceL* approach is precisely the difference between an implicit product and an explicit statement of the product. Consider as an example a simple program to compute the *mean* value of an unknown number of data values. For example, if the values are *(10,25,30,35,40),* then the *mean* is obtained by:

$$Mean = (10 + 25 + 30 + 35 + 40) \div 5$$

In the traditional approach one states an algorithm (i.e., a step-by-step sequence of instructions) that will produce the desired result. In *SequenceL,* one declares the desired data product:

<u>Traditional Approach - Pseudo Code</u>

1. *Read in the numbers, one at a time, counting them as they are read.*
2. *Add the values together (Sum them).*
3. *Divide the sum by the count obtained in step (1).*

<u>SequenceL Approach - Pseudo Code</u>

Divide the sum of the values by the number of values.

Computer languages of the future must provide abstractions that present more intuitive approaches to problem solving. This fact becomes even more profound when adding the complications that accompany parallel and distributed solutions to problem solving.

4. Nested Parallelisms

Many conventional concurrent languages add constructs to specify parallel execution. (Gelernter, 1985) These new constructs typically fit well into the procedural paradigm. For example, languages like Ada provide for concurrency through a fork and join construct implemented through procedure calls and returns. The Ada approach is a good one because the fork/join fits well into the notion of existing control constructs and the task (nested as a concurrent control structure in the procedure) fits well into the notion of subprograms. The task and the fork/join are orthogonal to the existing procedure and control flow constructs, respectively. To provide for the sharing of information between procedures, languages like Ada included the features needed for message passing.

Conventional approaches to concurrent programming add features to the existing procedural and object-oriented approaches to programming. Adding features can complicate these paradigms. New approaches to programming (Banatre, and LeMatayer, 1993), (Blelloch, 1996), (Breazu-Tannen, Buneman, and Naqvi, 1991), (Gelernter, D., 1985), (Hankin, C., Le Metayer, D., and Sands, D., 1992), (Sipelstein and Blelloch, 1990), (Suciu, 1995), (Cooke, 1998) provide for more intuitive approaches to parallelisms, resulting in programs that are easier to write and easier to understand. *SequenceL* provides a language platform in which parallelisms are realized in a more intuitive way; in effect, as a byproduct of the abstraction itself. (Cooke and Andersen, 2000) In *SequenceL* concurrency is implied and, like other control structures, need not be designed by the *SequenceL* programmer. Thus, at a high level, the problem solver declares data products that imply the potentially concurrent algorithms needed to solve the problems.

As a further simplification, in a uniform abstraction *SequenceL* provides a platform for a full host, query, schema, and integrity constraint language. *SequenceL* is a small language consisting of three constructs for nonscalar processing combined with a data dependency execution strategy. Through the interaction of the *SequenceL* constructs, a specifier can describe complicated problem solutions. The problem solutions are given strictly in terms of descriptions of data structures which are to hold the desired results.

5. The Nonscalar Constructs

Since the focus of *SequenceL* turns a specifier away from the algorithm, the traditional programming language constructs, e.g., input/output, selective, iterative, and arithmetic, are not as relevant as in traditional languages. To reverse the approach of the programmer requires new and different constructs. These new constructs are defined with reference to the sequence **-** the single data object of *SequenceL.*

A **sequence** is distinguished from a **set** because an element may occur more than one time in the sequence. The sequence differs from the **multiset**, because the ordinal positions in the sequence are significant. The example relations below help distinguish multisets and sequences:

RELATION	MULTISET	SEQUENCE
{a,b,a,f,s,a} = {b,a,f,a,s,a}	TRUE	FALSE
{a,b,a,f,s,a} = {a,b,a,f,s,a}	TRUE	TRUE
{a,b,a,f,a} = {a,b,a,f,s,a}	FALSE	FALSE

The equality relation for the sequence is a proper subset of the equality relation for the multiset.

A *SequenceL* data object may be singleton (e.g., [99]), or nonsingleton (e.g., [[1],[2],[3]] or [[[1],[2],[3]],[[10],[20],[30]]]). Complex structures of ***sequences*** containing ***sequences*** can be described. Like the list of LISP (McCarthy, 1960) and the array of APL (Iverson, 1962) and J (Iverson, 1994), the ***sequence*** of *SequenceL* can be used to build any data structure. (McCarthy, 1960)

For the sake of readability, the []'s around the singletons of nonsingleton structures are eliminated (i.e., stand-alone singletons will continue to be written as, e.g., [99], but singletons comprising a nonsingleton structure will be written as, e.g., [1,2,3] or in the case of a nested sequence structure, [[1,2,3],[10,20,30]]).

The *SequenceL* paradigm results in a language that replaces the traditional iterative and recursive constructs with three constructs that provide one with the ability to apply operations to input (or domain) sequences. When an operation is applied to a domain sequence, a range sequence is produced.

The *regular* construct is used when an operation is to apply to **all** elements of the domain sequence. In doing so, the construct results in the application of an operator to corresponding elements of the constituent operands. For example,

$$+\left[\begin{array}{|cc|}\hline 3 & 6\\ 4 & 7\\ 5 & 8\\ 6 & 9\\ \hline\end{array}\;\begin{array}{|cc|}\hline 10 & 3\\ 20 & 2\\ 30 & 4\\ 40 & 5\\ \hline\end{array}\right] = \begin{array}{|cc|}\hline 3+10 & 6+3\\ 4+20 & 7+2\\ 5+30 & 8+4\\ 6+40 & 9+5\\ \hline\end{array} = \begin{array}{|cc|}\hline 13 & 9\\ 24 & 9\\ 35 & 12\\ 46 & 14\\ \hline\end{array}$$

If the operands are not of the same cardinality and/or levels of nesting, they are normalized according to the semantics given in (Cooke, Daniel, 1998). In producing the range, a regular operation reduces the domain in terms of levels of nesting. The regular operation results in an intuitive specification of concurrency, similar to that performed in a vector processor. This solution is depicted by the eight addition operations above. These operations can be performed simultaneously.

The *irregular* construct is used when an operation is to be applied in a selective manner. Selection can occur based upon the **position** of an element, e.g., select all elements whose subscript in the domain sequence is an odd number. Selection can also occur based upon the **value** of an element, e.g., select all odd valued elements in the domain sequence. In the irregular processing, sequences can be combined in a wide variety of ways. In the example function below, the desire is to produce the *ith* number when the *ith* number is evenly divisible by *2:*

```
Evens { CONSUME(numbers)
        PRODUCE(Next) } WHERE   Next =

    numbers(i) when numbers(i) mod 2 = 0
        else
    []

Using i from numbers
```

The irregular construct typically reduces the domain in terms of cardinality when producing the range sequence.

The *generative* construct expands the domain sequence when producing the range. The simplest form of this construct allows for the expansion of integers within some range and is denoted by the ellipse. For example *[10,...,20] = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20].* An expression between two sets of ellipses can be employed to provide additional conditions for membership in the expanded sequence: *[10,...,*(pred,10), ...,100] = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100].*

Within a *SequenceL* function, each construct can be combined in any conceivable way with any other construct. *SequenceL* functions execute based upon a data dependency execution strategy. The denotational semantics of *SequenceL* are given in (Cooke, Daniel, 1998). In Friesen (1995) *SequenceL* was shown to be capable of representing the Universal Turing Machine. The implementation strategy is defined by the *SequenceL* computational model presented in the next section.

6. The Computational Model

The *SequenceL* computational model can be viewed as a modification of the Petri net computational model. This model can be described as a 4-tuple (*O, F, i , o*) where:

O: is a set of *objects;*
F: is a set of *function symbols.*

In order to describe i and o, we must first describe the set S of all *sequences* and the set T of all possible terms. The set S of all sequences is defined using the following inductive definition:

- for every $n \geq 0$, if $o_1,\ldots,o_n$ are objects, then the expression $[o_1,\ldots,o_n]$ is a sequence (in particular, when n=0, the empty sequence [ ] is a sequence);
- for every $n \geq 0$, if each element $s_1,\ldots,s_n$ is a sequence or an object, then the expression $[s_1,\ldots,s_n]$ is a sequence; and
- no other expressions are sequences.

For example, if 2, 3, and 4 are objects, then [2], [3,4], and [[3,4],[2]] are sequences. Concatenation can be defined in the usual way: $[s_1,\ldots,s_n] + [s'_1,\ldots,s'_{n'}] = [s_1,\ldots,s_n, s'_1,\ldots s'_{n'}]$.

Similarly, the set T of all terms can be defined by the following inductive definition:

- for every $n \geq 0$, if each element $t_1,\ldots,t_n$ is either a sequence, a function symbol, or an object, then the expression $[t_1,\ldots,t_n]$ is a term; and
- no other expressions are terms.

The difference between S and T is that in S only objects are allowed, while in T function symbols are also allowed. In terms of S and T, i and o are defined as:

i: is a mapping that assigns, to every function symbol $f \in F$, a set $i(f) \subseteq S \times S$ of pairs of sets of sequences. The set $i(f)$ defines the possible *inputs* of the function f; and

o: is a mapping that assigns, to every function symbol $f \in F$, a function $o(f)$ from $i(f)$ to T. The set $o(f)$ defines the possible *outputs* of the function f.

This model provides a means to *compute* a term tÎ T from an expression that contains function symbols, into a sequence s Î S that contains no function symbols. Informally, this computation is performed as follows:

- *mark* one-to-many function symbols in term t where $i(f) \subseteq T$; and
- apply $o(f)$ to all marked function symbols to eliminate the marked function symbols; then:

  - if the resulting expression has no remaining function symbols, this expression is returned as the result of the computation; otherwise
  - if the resulting expression still contains function symbols, then mark those elements f of T, where $i(f) \subseteq T$, and repeat the above procedure.

To formally describe this computation requires a notion of marking and the notion of applying marked functions.

Notion of Marking. Let F^* be a set with the same number of elements as F, and let f ® f^* be a one-to-one correspondence between F and F^*. Elements of the set F^* are called *marked* function symbols. In the definition of a term, marked function symbols are allowed in addition to un-marked symbols, which results in the definition of *marked terms*. Formally, the set T^* of *marked terms* can be defined as follows:

- for every $n \geq 0$, if each element $t_1,\ldots,t_n$ is an object, a function symbol, a marked function symbol, or a marked term, then the expression $[t_1,\ldots,t_n]$ is a marked term;
- no other expressions are marked terms.

Concatenation of marked terms is defined in a manner similar to the concatenation of sequences.

An *un-marking* function *u: O* È *F* È *F** È *T** ® *O* È *F* È *T*, which erases all the markings is defined as follows:

- for an arbitrary object $o \in O$, the un-marking function returns the object: $u(o) = o$;
- for an arbitrary un-marked function symbol object $f \in F$, the un-marking function returns the function symbol: $u(f) = f$;
- for an arbitrary **marked** function symbol object $f^* \in F^*$, the un-marking function returns the function symbol: $u(f^*) = f$; and
- the application of the un-marking function to an expression means applying each element of this expression: $u([t_1,\ldots,t_n]) = [u(t_1),\ldots,u(t_n)]$.

If $u(t) = t'$, i.e., if t' is obtained from t by an *un-marking*, then it can also be said that t is obtained from t' by *marking*.

Application of marked functions. Applying marked functions, *t'*® *t*, obtains an object, a function symbol, or a term t from a corresponding object, function symbol, or *marked* term t':

- if $o \in O$ is an object, $o \rightarrow o$;
- if $f \in F$ is an un-marked function symbol, then $f \rightarrow f$;
- if a pair of sequences (s,s'), where $s=[s_1,\ldots,s_n]$ and $s'=[s'_1,\ldots,s'_{n'}]$, belongs to $i(f)$, then $[s_1,\ldots,s_n, f^*,s'_1,\ldots,s'_n] \rightarrow o(f)(s,s')$ - where $o(f)(s,s')$ is the result of applying a function $o(f)$ to the pair (s,s');
- if $t'_1 \rightarrow t'_1,\ldots, t'_n \rightarrow t'_n$, then $[t'_1,\ldots, t'_n] \rightarrow [t_1,\ldots, t_n]$; and
- no other pairs (t',t) are related by $\rightarrow$.

Final definition for a computation process. A *computation process* is a finite sequence $t_1,t'_1, \ldots, t_m,t'_m ,t_{m+1}$ in which:

- $t_1, \ldots, t_m$ are terms;
- $t'_1, \ldots, t'_m$ are marked terms;
- for every i from 1 to m ,
    - t'_i is a marking of t_i , and
    - t_{i+1} is obtained from t'_i by applying marked functions (i.e., $t'_i \rightarrow t_{i+1}$); and
- t_{m+1} is a sequence.

A process begins with t_1 and ends with t_{m+1}. A sequence s is the *result of computing* the term t, if there exists a computation process that starts with t and ends with s.

Consider the following example:

```
[ bbcbb ]
match{CONSUME(pred(n)),PRODUCE(next)} =
        where next =
                [       [success]        when    =(n,1)
                                else
                        [pred(2..n-1),match] when =(pred(1),pred(n))
                                else
                        [failure]                              ]
```

The argument, *pred(n)*, consumes the function's predecessor from the database and n obtains the cardinality of the predecessor. The second *when-clause* (highlighted above) succeeds, resulting in the next state of the database being produced:

```
[ bcb ]
match{CONSUME(pred(n)),PRODUCE(next)} where next =
     [     [success]            when   =(n,1)
                                else
           [pred(2..n-1),match]      when   =(pred(1),pred(n))
                                else
           [failure]                          ]
```

The second *when-clause* succeeds for the final time, resulting in the next state of the database being produced:

```
[ c ]
match{CONSUME(pred(n)),PRODUCE(next)} where next =
     [     [success]            when   =(n,1)
                                else
           [pred(2..n-1),match]      when   =(pred(1),pred(n))
                                else
           [failure]                          ]
```

The first *when-clause* succeeds, resulting in the final state of the database:

```
[success]
```

The next section demonstrates the computational model through a data mining example.

7. Distribution of Functions

The computational model can be extended through the introduction of a Vector Random Access Machine (VRAM) introduced in (Blelloch, 1996). The VRAM provides a virtual model for computation consisting of an unlimited number of processing elements that have simultaneous access to a shared memory. The expression t is the shared memory in the computational model of *SequenceL*. The VRAM requires the introduction of an infinite set of processors, P = $\{p_1, p_2, p_3, \ldots\}$ and the introduction of an operator to indicate simultaneous execution ||, e.g., $a||b$ indicates that processes a and b take place simultaneously. Now, given all marked functions $F^* = \{f^*_1, f^*_2, \ldots, f^*_n\}$ based upon expression t, one can distribute the computations to processors with:

$$p_1(o(f^*_1)(s_1, s'_1)) \;||\; p_2(o(f^*_2)(s_2, s'_2)) \;||\; \ldots || \; p_n(o(f^*_n)(s_n, s'_n))$$

When combined with the irregular and the embedded data dependent strategy for execution one can, for example, establish powerful divide and conquer solutions for data mining that do not require the recursion (and resulting depth in complexity) (Blelloch, 1996) required by NESL:

[[simple text search for sinister]
search(CONSUME(pred(n),succ(m)), PRODUCE(pred, succ, next))where next =
[[pred(x), leave, succ]
] Using x from [[1,…,m],[2,…,m+1],…,[n-m+1,…,n]]
[sinister]]

where

leave(CONSUME(pred,succ), PRODUCE(next)) where next =
[[true] when pred = succ
else
[]]

In the example above, *i* will produce non-null results as it ranges from *1* to *(n-m)+1 = (31-8)+1* which is equal to *24.* Since *search** is marked, a processor is applied to *search:*

p_1*([[simple text search for sinister]*
search*(CONSUME(pred(n),succ(m)), PRODUCE(pred, succ, next)) where next =
[[pred(x), leave, succ]
] Using x from [[1,…,m],[2,…,m+1],…,[n-m+1,…,n]]
[sinister]]

The application of a processor to *search* results in the next state of *t,* where all occurrences of *leave* are marked:

[[simple text search for the word sinister], [sinister],

[simple t], ***leave****, [sinister]*
[imple te], ***leave****, [sinister]*
[mple tex], ***leave****, [sinister]*
[ple text], ***leave****, [sinister]*
[le text], ***leave****, [sinister]*
[e text s], ***leave****, [sinister]*
[text se], ***leave****, [sinister]*
[text sea], ***leave****, [sinister]*
[ext sear], ***leave****, [sinister]*
[xt searc], ***leave****, [sinister]*
[t search], ***leave****, [sinister]*
[search], ***leave****, [sinister]*
[search f], ***leave****, [sinister]*
[earch fo], ***leave****, [sinister]*
[arch for], ***leave****, [sinister]*
[rch for], ***leave****, [sinister]*
[ch for s], ***leave****, [sinister]*
[h for si], ***leave****, [sinister]*
[for sin], ***leave****, [sinister]*
[for sini], ***leave****, [sinister]*
[or sinis], ***leave****, [sinister]*
[r sinist], ***leave****, [sinister]*
[siniste], ***leave****, [sinister]*
[sinister], ***leave****, [sinister]*
]

The marked occurrences of *leave* will each have a processor assigned, resulting in *24* concurrent computations of the *leave* function.

[[simple text search for the word sinister], [sinister],

p_1*([simple t],* ***leave****, [sinister]) ||*
p_2*([imple te],* ***leave****, [sinister]) ||*
p_3*([mple tex],* ***leave****, [sinister]) ||*
p_4*([ple text],* ***leave****, [sinister]) ||*
p_5*([le text],* ***leave****, [sinister]) ||*
p_6*([e text s],* ***leave****, [sinister]) ||*
p_7*([text se],* ***leave****, [sinister]) ||*
p_8*([text sea],* ***leave****, [sinister]) ||*
p_9*([ext sear],* ***leave****, [sinister]) ||*
p_{10}*([xt searc],* ***leave****, [sinister]) ||*
p_{11}*([t search],* ***leave****, [sinister]) ||*
p_{12}*([search],* ***leave****, [sinister]) ||*
p_{13}*([search f],* ***leave****, [sinister]) ||*
p_{14}*([earch fo],* ***leave****, [sinister]) ||*
p_{15}*([arch for],* ***leave****, [sinister]) ||*
p_{16}*([rch for],* ***leave****, [sinister]) ||*
p_{17}*([ch for s],* ***leave****, [sinister]) ||*
p_{18}*([h for si],* ***leave****, [sinister]) ||*
p_{19}*([for sin],* ***leave****, [sinister]) ||*
p_{20}*([for sini],* ***leave****, [sinister]) ||*
p_{21}*([or sinis],* ***leave****, [sinister]) ||*
p_{22}*([r sinist],* ***leave****, [sinister]) ||*
p_{23}*([siniste],* ***leave****, [sinister]) ||*
p_{24}*([sinister],* ***leave****, [sinister]*
]

The 24 occurrences of the function *leave*, combined with associated arguments can exist in shared memory and produce the final result:

[[simple text search for the word sinister], [sinister],
[true]]

Lacking the ability to combine features similar to the irregular and generative constructs, it would appear that a series of recursive calls are necessary in order to produce a divide and conquer solution to this search problem in NESL.

Although the computational model appears to duplicate much data in shared memory, the duplication of data is not necessary since all inputs are *read-only.* Optimizations will result in no duplication.

8. Simplified Problem Solutions in *SequenceL*

In this section, two example problems are specified in *SequenceL*, and are programmed in JAVA. Thus the reader can compare the *SequenceL* and JAVA solutions to the same problems. The goal here is for the reader to see that much of the language-based complexity introduced even by modern languages like JAVA, is implied by the simpler *SequenceL* solutions. Thus, *SequenceL* provides a language base that may reduce the need for the involvement of sophisticated computer scientists in the solution of interdisciplinary problems.

8.1. Data Parallelisms in *SequenceL*

Data parallelisms in *SequenceL* are definable in a straightforward manner through the use of the data dependent execution strategy of *SequenceL* functions. Given the following database configuration, a *word-search* example is represented in an intuitive manner:

Here is a test string

Search(**Consume**(pred(n),succ(m)), **Produce**(next)) **where** next =

pred(x) = succ

Using x **From** [[1,...,m],...,[n-m+1],…,n]]

test

In this example, the cardinalities of the predecessor and the successor are obtained in identifiers *n* and *m*, respectively. Based upon the *using clause*, the *predecessor's* subscript *x* obtains values (in order) from the generated sequences:

[[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7],[5,6,7,8],[6,7,8,9],[7,8,9,10],[8,9,10,11],[9,10,11,12],
[10,11,12,13],[11,12,13,14],[12,13,14,15],[13,14,15,16],[14,15,16,17],[15,16,17,18],
[16,17,18,19],[17,18,19,20],[18,19,20,21]]

The *using clause* helps subdivide the larger data set into *18* smaller sets – much like the parceling of data accomplished in lines *35-39* and *7-12* in the JAVA version presented in Exhibit 1. The function results in the following set of relations being added to the *SequenceL* program:

[[here] = [test], [ere] = [test], [re i] = [test], [e is] = [test], [is] = [test], [is a] = [test],
[s a] = [test], [a t] = [test], [a te] = [test], [tes] = [test], [test] = [test], [est] = [test],
[st s] = [test], [t st] = [test], [str] = [test], [stri] = [test], [trin] = [test], [ring] = [test]]

The concurrent evaluation of the resulting conditions is now clearly implied due to the computational model of *SequenceL* - a model that allows the execution of any function or operator as soon as the data required for the operator or function is available:

[[here] = [test] || [ere] = [test] || [re i] = [test] || [e is] = [test] || [is] = [test] || [is a] = [test] ||
[s a] = [test] || [a t] = [test] || [a te] = [test] || [tes] = [test] || [test] = [test] || [est] = [test] ||
[st s] = [test] || [t st] = [test] || [str] = [test] ||[stri] = [test] || [trin] = [test] || [ring] = [test]]

After concurrent evaluation, the vector of boolean results remains in the database:

[false, false, false, false, false, false, false, false, false, false, true, false, false, false, false, false, false, false]

The parallelisms in *SequenceL* are more intuitive in that the parallelisms do not result in the separation of elements of functionality and, since parallelisms are implied, the solution does not require the use of additional constructs as is seen in the *thread, run(), try,* etc. required in the JAVA concurrent solution.

8.2. Data Parallel Data Mining of Images in *SequenceL*

Although the example data parallel problem solution developed so far in this paper is rather simple, the example scales up to many real-world data mining problems involving image processing and security-based text searches. The searching of image databases follows the same parceling and scatter/gather approach to programming. The difference is that the objects for which one is searching are characterized mathematically. These mathematically defined objects serve as a *kernel,* which drives the search much like the string *Here is a test string* served as the object for which the text string search was done in this paper. An example of an image-based kernel is the following Gaussian-Laplacian operator often employed for edge detection:

$$GL(x)(y) = \frac{1}{p\,0.3^4} \left[1 - \frac{(x^2+y^2)6.0}{2*0.3^2} \right] 2.7183 \left[\frac{(x^2+y^2)6.0}{2*0.3^2} \right]$$

The operator is applied for values from *1* to *n* for *x* and *y. The SequenceL* function corresponds directly to the mathematical formula:

Laplace(**Consume**(succ(n)) **Produce**(Next)) **where** next(n,n)=

$$\frac{1}{3.14*0.3\text{^}4} \quad 1 - \frac{(x\text{^}2+y\text{^}2)*6.0}{2*0.3\text{^}2} \quad 2.7183\text{^} \frac{(x\text{^}2+y\text{^}2)*6.0}{2*0.3\text{^}2}$$

Using x,y **Form** [1,...,n]*[1,...,n]

In JAVA (see Exhibit 2), the programmer must decompose the formula and construct loops to perform the appropriate computations. Furthermore, to perform concurrent processing of the image in JAVA requires sophisticated design typically involving a computer scientist. As with the other solutions, a concurrent solution to apply the mask to a larger image for edge detection is implied by the *SequenceL* solution and requires no additional effort on the part of the programmer.

9. Summary

Language constructs for iteration and concurrency are abstracted out of the *SequenceL* language abstraction, freeing the problem solver from much of the difficult effort required to produce algorithms that imply the data products they desire. Thus, scientists and engineers are more apt to employ *SequenceL* in solving complex problems with greater ease, freeing them from the need to consult with Computer Scientists as much as they would if they were solving problems using traditional computer languages. One of the scarcest resources for any large interdisciplinary project team is the computer scientist who is sophisticated enough to be capable of producing complex, concurrent problem solutions. *SequenceL* is a language for which the goal has been to free problem solving teams from much of the technical detail that requires the attention of computer scientists.

9. Acknowledgement

Research sponsored under NASA/CS - 1314-44-1476.

10. References

Banatre, J.P. and LeMatayer, D., 1993, "Programming by Multiset Transformation," *Communications of the ACM*, Vol. 36 No. 1, pp.98-111.

Bishop, J., 1990 "The Effect of Data Abstraction on Loop Programming Techniques," *IEEE Trans. Soft. Eng.* Vol. SE-16, Number 4, pp. 389-402.

Blelloch, G., 1996, "Programming Parallel Algorithms," *Communications of the ACM*, Vol. 39, No. 3, pp. 85-97.

Breazu-Tannen, V., Buneman, P., and Naqvi, S., 1991, "Structural Recursion as a Query Language," *In Proceedings of 3rd International Conference on Database Programming Languages,* Nafplion, Greece, Morgan Kaufmann, pp. 9-19.

Cooke, D. E., 1993, "Possible Effects of the Next Generation Programming Language on the Software Process Model," *International Journal on Software Engineering and Knowledge Engineering*, Vol. 3, No. 3, September 1993, pp. 383-399.

Cooke, D. E., Demirors, O., Demirors, E., Gates, A., Kraemer, B., and Tanik, M. M., 1996, "Languages for the Specification of Software," *Journal of Systems and Software,* 32:269-308.

Cooke, D.E., 1996, "An Introduction to SEQUENCEL: A Language to Experiment with Nonscalar Constructs," *Software Practice and Experience*, Vol. 26, No. 11, 1205-1246.

Cooke, D., 1998, "SequenceL Provides a Different way to View Programming," to appear in *Journal of Computer Languages*.

Cooke, Daniel, 1998, "SequenceL Provides a Different Way to View Programming," *Computer Languages* 24: 1-32.

Cooke, D.E. and Urban, J.E., 1998, "The Application of the SequenceL Language to Complicated Database Applications," invited paper in the *Proceedings of IEEE Workshop on Application-Specific Software Engineering and Technology*, (pp. 166- 171)

Cooke, Daniel E. and Andersen, Per, 2000, "Automatic Parallel Control Structures in SequenceL," *Software Practice and Experience*, Volume 30, Issue 14, 1541-1570.

Friesen, B., 1995, *The Universality of BagL,* Masters Thesis, University of Texas at El Paso.

Gelernter, D., 1985, "Generative Communications in Linda," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112.

Hankin, C., Le Metayer, D., and Sands, D., 1992, *A Calculus of Gamma Programs,* Publication Interne no 674, Juillet, IRISA, France.

Iverson, K., 1962, *A Programming Language,* Wiley, New York.

Iverson, K., 1994, *J Introduction and Dictionary,* Iverson Software Inc. Toronto.

McCarthy, J., 1960, "Recursive Functions of Symbolic Expressions and Their Computation by Machine," *Communications of the ACM*, Vol. 3, No. 4, April 1960, pp. 184-195

Mills, H., and Linger, R., 1986, "Data Structured Programming: Programming without Arrays and Pointers," *IEEE Trans. Soft. Eng.* Vol. SE-12, Number 2, pp. 192-197.

Sipelstein, J., and Blelloch, G., 1990, *Collection-Oriented Languages*, Report, School of Computer Science, Carnegie Mellon University, CMU-CS-90-127.

Suciu, D., 1995, *Parallel Programming Languages for Collections*, Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania.

```
class wrdsrch2 extends Thread{
String text;
String target;
boolean found;
int i;

wrdsrch2(String in, String targ, int k)    {
        target=targ;
        text=in;
        found=false;
        i=k;
}

public void run()    {
        if(text.equals(target))
                {found = true;}
}

public static void main (String args[]) {
        int i, j, k, n, n1;

        String s = "here is a test string";
        String s1 = "test";
        char[] sample = s.toCharArray();
        char[] find = s1.toCharArray();

        System.out.println(sample);

        n  = sample.length;
        n1 = find.length;
        String send;

        wrdsrch2 w[] = new wrdsrch2[(n-n1)+1];

        for(i=0;i<=n-n1;i++)
                {send = s.substring(i,i+n1);
                 w[i] = new wrdsrch2(send,s1,i);
                }
        System.out.println("To Run ");

        for(i=0;i<=n-n1;i++)
                {w[i].start();}

        for(i=0;i<=n-n1;i++)
                {try {w[i].join();
                 catch (InterruptedException ignored) { }
                }
        System.out.println("The answer is:  ");

        for(i=0;i<=n-n1;i++)
                {System.out.println(w[i].found);}
}}
```

Exhibit 1. Data Parallelism in a Word Search Problem.

```
import java.io.*;

class Lapl{

public static void main (String [] args) throws IOException {

PrintStream f = new PrintStream(new FileOutputStream("lapl.out"));

float kernel[][] = new float[13][13];
float sigma=(float)0.3, r_factor=(float)6.0, sigma4, sigma2,x2;
float r2,deltax,deltay,sum,sigma2x2,sigma4xpi;
float r2sigma2x2, sum_mask;
int i,x,y,centerxy;

// Computation is
// 1/pi*sigma**4 [1 - x**2 + y**2/2*sigma**2] e**(x**2+y**2/2*sigma**2)
//
// Compute sigma**2
            sigma2 = (float) Math.pow((float)0.3,(float)2.0);
// Compute 2*sigma**2
            sigma2x2 = sigma2 * 2;
// Compute sigma**4
            sigma4=(float) Math.pow((float)0.3,(float)4.0);
// Compute 1/pi * sigma**4
            sigma4xpi = 1/(sigma4*(float)3.1416);
            centerxy = (int) 13/2;

            for (x=0;x<13;x++)
               for (y=0;y<13;y++)
                        {           deltax = x-centerxy;
                                    deltay = y-centerxy;
// Compute x**2 + y**2
                                    r2 = (float) Math.pow(deltax,(float)2)+
                                                (float)Math.pow(deltay,(float)2);
                                    r2 = r2 * r_factor;
// Compute x**2 + y**2/2*sigma**2
                                    r2sigma2x2 = - r2/sigma2x2;
// Final Computation for formula above for x,y
                                    kernel[x][y]=sigma4xpi *
                                                   (1+r2sigma2x2) *
                                                   (float)Math.pow((float)2.7183,(float)sigma2x2);
                        };

            for (x=0;x<13;x++)
               {f.println("row "+ x);
                for (y=0;y<13;y++)
                        f.print(kernel[x][y]);
                f.println(" ");
              };

}}
```

Exhibit 2. Gaussian-LaPlacian Operator in JAVA.