The VAL Language: Description and Analysis

JAMES R. McGRAW
Lawrence Livermore National Laboratory and University of California at Davis

VAL is a high-level, function-based language designed for use on data flow computers. A data flow computer has many small processors organized to cooperate in the execution of a single computation. A computation is represented by its data flow graph; each operator in a graph is scheduled for execution on one of the processors after all of its operands' values are known. VAL promotes the identification of concurrency in algorithms and simplifies the mapping into data flow graphs.

This paper presents a detailed introduction to VAL and analyzes its usefulness for programming in a highly concurrent environment. VAL provides *implicit concurrency* (operations that can execute simultaneously are evident without the need for any explicit language notation). The language uses function- and expression-based features that prohibit all side effects, which simplifies translation to graphs. The salient language features are described and illustrated through examples taken from a complete VAL program for adaptive quadrature. Analysis of the language shows that VAL meets the critical needs for a data flow environment. The language encourages programmers to think in terms of general concurrency, enhances readability (due to the absence of side effects), and possesses a structure amenable to verification techniques. However, VAL is still evolving. The language definition needs refining, and more support tools for programmer use need to be developed. Also, some new kinds of optimization problems should be addressed.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; *data-flow languages*; *VHL*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*multiprocessing/multiprogramming*

General Terms: Design, Languages

1. INTRODUCTION

As the use of computers expands in scientific areas, the demand for faster computers continues to increase at a surprising rate. As one example, physicists at Lawrence Livermore National Laboratory (LLNL) have computations requiring gigaflop speed (one billion *floating point* operations per second) to insure reasonable response time. In the past, improved hardware technology has been responsible for most of the advances in speed; however, those gains have been slowing. Physical constraints (speed of light, power-cooling ratios, etc.) may soon limit the speed achievable by a single processor. One possible solution applies multiple processors to the execution of each task. Data flow computing carries out this strategy on a massive scale (e.g., 1000 processors per task). To achieve high concurrency, appropriate tools must be designed for describing algorithms and mapping them onto this unusual environment. VAL is a high-level language developed at M.I.T. for exactly this purpose. This paper presents the basic

This work was performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract W-7405-ENG-48 and by its Office of Basic Energy Sciences, Mathematical Sciences Branch.
Author's address: Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550.

features of the language through discussion and extensive examples and then analyzes its usefulness in a data flow environment.

1.1 Background

Highly computational tasks often contain substantial amounts of concurrency. At LLNL the majority of these programs use very large, two-dimensional arrays generated within loops. In many cases, all new array values could be computed simultaneously, rather than stepping through the array one element at a time. To date, *vectorization* (i.e., automatic detection of parallel[1] computations by compilers) has been the most effective scheme for exploiting this concurrency. Pipelining and independent multiprocessing forms of concurrency are also applicable to these problems, but neither the hardware nor the software exists to make them workable.

Data flow architecture incorporates concurrency in a graph-oriented system; every computation is represented by a data flow graph. The nodes of the graph represent operations; the directed arcs represent data paths. Execution of a graph is based solely on operand availability; each operation may begin execution as soon as all of its inputs are present. When an operation completes, the results are transmitted via the output arcs to the next operations.[2] Figure 1 illustrates the graph representation of a simple VAL function for computing the mean and standard deviation of three inputs. The concurrency in this graph is limited only by the data dependencies inherent in the computation. If all inputs become available at once, four operations (**plus** and three **square** operations) can begin. Even though in the program text these four operations are scattered, the graph form exposes this general concurrency. Vectorized concurrency is embedded in the execution because the three **square** operations can proceed together. Finally, pipelining is possible if multiple executions of the function proceed together; separate sets of values can work their way through the graph at their own rate.

One major component of data flow research is the design of appropriate architecture for graph execution. Quite a few different hardware organizations have been proposed for data flow [15, 18, 21, 28, 40, 48, 50]. One feature common to all of these approaches (and probably essential for high performance) is that each operator is scheduled for execution on a particular processor by the *hardware* after all operands become available.

Software for data flow programming must help identify concurrency in algorithms (and their corresponding programs) and map that concurrency into graphs. A particular algorithm may have many different data flow graphs ranging from nearly linear to vastly concurrent. Data flow can only achieve ultra-high speeds

[1] In this paper the term *concurrency* means any form of simultaneous activity; *parallel* refers to a special case of concurrency where a specific operation can be applied simultaneously to a set of data (SIMD).

[2] For historical reference, the concept of using data flow graphs to represent a computation was independently developed by four groups: Karp and Miller [26], Rodriguez in a 1967 thesis, later published as a technical report [44], Adams [4], and Sutherland in unpublished work at Lincoln Laboratory, Cambridge, Mass. The basic program execution mechanism for the graphs has its roots in three different efforts: Seeber and Lindquist's [45], Shapiro, Saint, and Presberg's [46], and Miller and Cocke's [35]. Finally, this view of task execution is a direct application of Petri nets [38, 39].

(a)

```
function Stats(X, Y, Z: real returns real, real)
  let
    Mean real := (X + Y + Z)/3;
    SD real := SQRT((X² + Y² + Z²)/3 - Mean²);
  in
    Mean, SD
  endlet
endfun
```

(b)

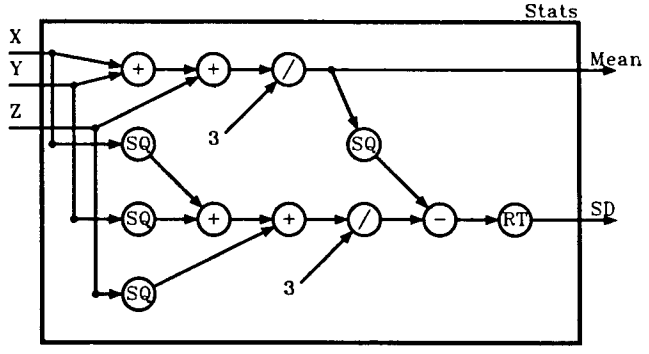

Fig. 1. (a) A simple statistics function. (b) Its corresponding data flow graph.

if the graphs lean toward the latter. Many software options are available, including writing pure data flow graphs [10, 17, 30], altering FORTRAN to generate graphs [29, 37], using one of the newer concurrent languages [11, 53], and (as always) defining a new language [3, 7, 14, 23, 47, 49]. Regardless of the approach taken, the ultimate criterion for evaluating data flow software must be its effectiveness in generating highly concurrent graphs for solving large problems.

The VAL language [3] is currently being developed by a group at M.I.T. under the direction of Dennis. Their goal is a language suitable for the expression and translation of algorithms into data flow graphs. During the past three years LLNL has cooperated in the design and evaluation of VAL. Emphasis at LLNL has been on the usefulness of VAL for current programming problems and its potential for future applications.

1.2 Overview of Paper

This paper presents the design goals and major features of VAL with examples and an analysis of its current status. To illustrate use of the language, one sample program is used throughout the text. Appendix A contains a complete program for adaptive quadrature [19]; nearly all VAL program segments in this paper are taken directly from this source and are accompanied by function names and line numbers to help identify the context. The reader is advised to scan Appendix A prior to continuing with the rest of this paper.

The next section of the paper presents the design goals for VAL. These goals had a strong influence on the basic structure of the language, which is detailed in Section 3. Section 4 discusses the implementation of the adaptive quadrature program, working from the basic algorithm through the design and including an analysis of its speed. Section 5 analyzes VAL, focusing on three aspects of the

language: its definition (both practical and theoretical), its ease of use by programmers, and its translation to various machines.

2. DESIGN PRINCIPLES

The goal of data flow software is to identify concurrency in algorithms and to map as much concurrency as possible into the graphs. In response to this goal, two basic design principles have been followed in the development of VAL. First, VAL provides *implicit concurrency* (operations that can execute independently are evident without the need for any explicit language notation). Programmers must know what concurrency a computer system will be able to exploit, but in an environment of 1000 processors they cannot be expected to identify all of it explicitly. The second design principle in VAL is to help programmers write *correct* programs. The complexity of a multiprocessing environment must not increase the time and effort needed to arrive at a working program. This section examines these design principles in more detail, concentrating on how they influenced VAL.

2.1 Implicit Concurrency

Execution speed in a data flow environment depends heavily on a programmer's ability to write programs that contain large amounts of concurrency. As shown in Section 1, concurrency may take many forms, but it certainly includes concurrency of individual operations. At this level it is totally unreasonable to ask programmers to state which operations can proceed together; automatic means must be used to provide this information to the computer. Unfortunately, programmers must be aware of the amount of concurrency found through these means because of their overall responsibility for speed. They must be able to determine when a program executes slowly because of a poor algorithm and when it is due to a poor implementation. These conditions suggest that a language definition should let programmers focus on the presentation of an algorithm but constrain their style sufficiently so that concurrency can be *easily* identified by both compiler and reader.

VAL meets this design goal of implicit concurrency by using completely functional language features. Programmers write expressions and functions (i.e., procedures that take input values and produce a result without otherwise altering the program environment). The significant property possessed by functions and expressions is that of *noninterference*; once the values of all inputs are known, execution cannot influence the results of any other operations ready to execute. This leads to a general rule programmers can use to determine concurrency:

> If two operations do not depend on the outcomes of each other, then they can execute simultaneously.

Here the term "operations" may refer to basic machine instructions, language-defined expressions, and function invocations. Not surprisingly, this is the same rule that applies to the execution of data flow graphs. Referring to the program in Figure 1, we can use this rule to "find" the same concurrency exposed by the graph. The three operations in the *Mean* calculation are strictly sequenced because the result of one is an input to the next. However, the above rule does

not limit concurrency to be within one expression. Hence, in the *SD* calculation the three **square** operations do not require input from each other; so they may execute concurrently with the first **plus** in the *Mean* computation. Similar observations on the remainder of the program yield the additional concurrency. In VAL all concurrency stems from this one rule; there are no explicit commands for concurrent execution and likewise no commands for synchronization.

VAL's restriction of allowing only expressions and functions is the key limitation on programming style. The noninterference property requires that language features be incapable of producing side effects. The problem with eliminating side effects is that almost everyone's programming experience is tied to them. Assignment statements stand out as the principal example. The concept of updating memory (i.e., modifying variables) must be thrown away. Another well-known (but rarely recommended) language feature, *aliasing*, must also be banned. These two examples only begin the list of common features that must be avoided, but they illustrate the enormity of the situation. A programmer can overcome this situation by adapting to a value-oriented environment. VAL uses values. In a value system, new values are defined and then used in many places, but no value can ever be modified. While this approach may seem rash at first, there is a precedent in pure LISP [33], and, in fact, the transition is not really that difficult.

2.2 Aids for System Design

The other design principle embraced in VAL is to help programmers produce correct and time-efficient programs. Programming for a highly concurrent system is certain to add some complexity beyond that encountered for sequential ones. To counter this added difficulty, the programming environment must provide adequate support. The language should be clear and concise; all potential actions should be understood by the programmer. Furthermore, the language must help simplify critical programming chores such as error handling, debugging, and speed analysis.

One major effect of this design principle was the selection of an algebraic syntax (i.e., infix operators in expressions, where possible). Looking at the previous comments, some variant of pure LISP seems like a natural choice. However, the readability of LISP drops off drastically when dealing with large arithmetic expressions and with some standard control structures. Since most programs requiring the high performance of data flow involve extensive numerical computations and large expression evaluations, a conventional algebraic notation seems appropriate.

Other important effects of this design principle are visible in VAL's definition of legal programs and in its handling of run-time errors. Scope rules and type checking tend to be restrictive, and they must be tightly enforced. Responses to run-time errors are also well defined. If an array subscript is out of bounds or if some computation causes an overflow, the language definition specifies precisely what results are produced by the language. This is in contrast to the standard FORTRAN view (which is shared by most languages) that the results are undefined and hence may be implemented in any way convenient for the compiler.

With this brief description of the design principles, the language features to be presented may be somewhat easier to follow. For further discussion of the

motivation and reasoning behind VAL and other data-flow-like languages, see [2].

3. LANGUAGE FEATURES

This section surveys the major features of VAL, concentrating on the more interesting and unusual parts of the language. The presentation takes an informal approach to describing parts of the language and uses examples extensively. Implicit concurrency resides in almost every feature of the language, and this fact is emphasized throughout the discussion. The principal areas of interest in VAL are data types, values, basic expressions, parallel expressions, sequencing expressions, and error handling.

3.1 Data Types

VAL data types are similar to those found in most current languages. **Boolean**, **integer**, **real**, and **character** comprise the basic language-defined types. Each of these types carries an appropriate set of operators (e.g., +, −, &, |, ~ (not), >, =). New types may be constructed through **array**, **record**, and **oneof** (discriminated union) definitions. The principal differences between VAL types and types in most other languages are in the definition and use of constructed types, in type-checking rules, and in the existence of error values within every type.

VAL arrays are unusual because the array bounds are not part of the type definition; they are extra information associated with each individual array. The only type information associated with an array is its component type (which may be any type, including **array**). Hence, arrays use the declarative form illustrated here:

type *Interval_list* = **array**[*Interval*]; (*AQ*, 3)[3]

The program-defined type *Interval_list* is represented by an array of *Intervals*. Since the length of the array is not part of the type, this one type describes all finite-length, ordered sequences of *Intervals*. Access to individual elements of the sequence must be made through **integer** subscripts. Because the range information is eliminated from the type, strong type-checking can be enforced without getting into the PASCAL difficulty that arrays of different bounds cannot be passed to the same procedure. This option also increases possibilities for useful array operations, such as

(1) shifting the origin	**array_adjust**;
(2) adding elements at either end	**array_addh**, **array_addl**;
(3) deleting elements at either end	**array_remh**, **array_reml**;
(4) catenating two arrays	||
(5) merging two arrays	**array_join**;
(6) setting array bounds	**array_seth**, **array_setl**;
(7) testing array bounds	**array_limh**, **array_liml**.

The catenate operator is infix; all others are function invocations. This approach

[3] This notation is used to indicate the location of the corresponding program segment in Appendix A. In this case the line is taken from line 3 of the *Adaptive_Quadrature* (*AQ*) function.

to array definitions yields a very flexible and dynamic data-structuring facility. String manipulation is a special case of the above primitives operating on arrays of characters.

Array construction in VAL takes an unusual form in order to improve possibilities for concurrency. All elements of an array can be specified simultaneously, thus allowing all of the evaluations of array entries to proceed together. The syntax for array construction is an arbitrarily long list of ordered pairs, with the list enclosed in square brackets. The first of the pair is the index position in the array, and the second is an expression representing the value to be held in that position. For example,

[1: *left_interval*; 2: *right_interval*] (*Bti*, 16–17)

is an expression that creates an array with two elements. The values bound to the two identifiers become the values for the corresponding array locations. In this case, there is no real concurrency because the expressions are simple values. In general, however, all expressions inside the array construction can be computed simultaneously.

Record-constructed data types differ from most current approaches to record types in both their static type properties and their construction. The type of a VAL record is based solely on the constituent field names and their associated types. In particular, neither the order of definitions nor any type name bound to the structure influences the type-matching process. Record construction is patterned after the array construction scheme to promote concurrency.

record[*x_low*: *low*; *Fx_low*: *lowv*; (*Int*, 5–8)
x_high: *high*; *Fx_high*: *highv*]

The above expression builds a record with four fields. Each field name is followed by an expression representing the value to be entered in the record. As with arrays, all field values can be computed concurrently.

The final type constructor, **oneof**, permits discriminated union types as in CLU [32]. Values of this type appear to be different types at different times during execution. For example, in the adaptive quadrature algorithm, the result of analyzing an interval will be either two new intervals (the old one divided in two) or no intervals at all (if the analysis is complete). Such a result type can be specified by

type *Result_info* = **oneof**[*none*: **null**; (*AQ*, 4)
more: *interval_list*];

An instance of type *Result_info* can represent a value of type *Interval_list* (i.e., an array with two entries) or of type **null**. The tag fields (*none*, *more*) identify the constituent types. When an object gets a value, one of the tags must be specified along with the appropriate information. For example,

make *Result_info*[*more*: *two_intervals*] (*Bti*, 18)

builds an object of type *Result_info*, whose constituent type is an *Interval_list*. In this case the identifier *two_intervals* has been previously bound to an array containing the two new intervals. The keyword **make** indicates that a value of the **oneof** type is to be constructed. Later, in the discussion of sequencing

expressions, an operation (**tagcase**) is introduced for accessing the information in a **oneof** value. As in the case of records, two **oneof** types are equivalent if the field names and the constituent types are the same.

Within this framework for type definition and use, VAL imposes strong, structural type-checking. In VAL two types are equivalent if they represent identical structures; the names bound to the types do not influence this analysis. Every function has specific types expected for each parameter. In a function call, actual and formal parameters must have equivalent types. Automatic type conversions are *never* made by the language. As an example, the add operator can take two **integers** or two **reals**, but it will not accept one of each. Similarly, when values are bound to identifiers, the type of the value must be equivalent to the declared type of the identifier (i.e., no **real** can be bound to an **integer** identifier). Rather than having implicit type coercion, the language provides functions that will do conversions for all of the reasonable cases. This approach was taken partly as an aid to programmers to make it completely clear where conversions occur in the program, and partly to simplify the language rules with respect to mismatched types. As an implementation note, all type checking defined in VAL can be done at compile time.

The last unusual aspect of VAL's data types is the presence of special error values which are an integral part of every data type. Included among these values are **undef**, **pos_over** (positive overflow), **miss_elt** (missing array element), and **unknown**. They provide a simple mechanism for handling run-time errors without violating any type-correctness properties. The language also defines functions that test for the presence of these values, and all of the operators have well-defined propagation rules in the face of these values. More details on this subject are included in the section on error handling.

3.2 Values

The VAL programming philosophy is value-oriented as opposed to the more traditional variable orientation. Most languages have concepts like "variables" and "memory updating," which imply that objects are mutable or modifiable. In VAL these concepts have been replaced by a value system where every object is immutable. The basis for this view resides in the target environment of data flow graphs. Once a value is computed by some operator and put on an output arc, the same value must be transmitted to all receiving operators. At first glance this restriction may seem severe to programmers, possibly making VAL unusable. Fortunately, this is not so.

In a value-oriented language the idea of binding values to identifiers is still available. The important point is to prevent identifiers from being used as variables; hence this rule:

> Once an identifier is bound to a value, that binding remains in force for the entire scope of access to that identifier.

This is commonly referred to as the *single-assignment*[4] rule [2, 40, 47]. Each

[4] While technically accurate, this term can be confusing because assignment brings to mind the concept of variables. A more appropriate term would be *defined constants*, which has been used to describe a similar feature in ALGOL 68.

time a scope of access (block, function, etc.) is entered, new bindings can be made which remain in force until that scope of access is completed. Notice that on each scope entry the identifier-value bindings may be different, distinguishing these bindings from normal constants.

Array and record operations are most affected by the value orientation. Structured objects must be viewed as single values. Hence, arrays and records can never be modified. The only option is to build a new array or record that has the same values in all of the old positions except for the particular element that is to be changed. Enforcement of this rule is accomplished by requiring that every identifier-value binding be made to a full identifier (not to a field or subscript position). So, for example, no binding can begin with "$a[i] := \ldots$" To compensate for these difficulties in building a structure, VAL provides other schemes for allowing arrays and records to be built completely (in parallel if possible) and then bound to an identifier for use. The simple array construction operator was illustrated earlier as part of the description of arrays; a more complex array construction is introduced in the discussion of parallel expressions.

3.3 Basic Expressions and Functions

The VAL language contains expressions and completely functional operators. There are no "statements" in the conventional sense. All active (i.e., nondeclarative) language features are functions; they use values provided by the current execution environment and have the sole effect of producing a set of result values. The rules for construction of functions and expressions provide protection against side effects and at the same time help identify many forms of implicit concurrency.

VAL functions look and act like functions in most conventional languages but have a few extra restrictions to enhance concurrency. In function definitions the formal parameters name and type all values to be passed on each call. The body of a function is some expression which produces the result. Within that body only the formal parameters and locally defined identifiers may be accessed; no access to global identifiers (except type names) is permitted. Also, since values are bound to the formal parameters at the call site, the body of a function cannot attempt to rebind those values (single-assignment rule). These rules eliminate any potential side effects normally caused by functions. All parameters are passed by value, and their evaluations can always proceed concurrently. This concurrency is exploited in the adaptive quadrature program at the very end where the call

Integrate(*low*, *Evaluate_Function*(*low*), *high*, *Evaluate_Function*(*high*))

allows the two calls to *Evaluate_Function* to proceed together.

The simplest expressions in VAL look just like expressions in most other languages. Infix operators (e.g., +, −, *, and /) can be applied to identifiers to compute some result. Among other things, the identifiers can be bound to simple types (**integer**, **real**, etc.), structured types (subscripted arrays and record field-accessing), and function calls. The amount of concurrency in an expression depends on the expression. The standard operator precedences are honored by VAL, but values for the two operands for any infix operator can always be

```
let                                                          (Bti, 4-19)
   left_interval  :  Interval
                  := record[x_low:    left;
                            Fx_low:   leftv;
                            x_high:   mid;
                            Fx_high:  midv];
   right interval :  Interval
                  := record[x_low:    mid;
                            Fx_low:   midv;
                            x_high:   right;
                            Fx_high:  rightv;];
   two intervals  :  Interval_list
                  := [1: left_interval;
                      2: right_interval];
in make Result_info[more: two_intervals]
endlet
```

Fig. 2. A **let-in** expression illustrating temporary environment expansion. It builds an array containing information on two intervals.

computed in parallel. So in the expression

```
(right − mid) * (rightv + midv) * 0.5                        (CQ, 11-12)
+ (mid − left) * (midv + leftv) * 0.5
```

all operations inside parentheses can execute simultaneously.

This general notion of an expression is enhanced significantly in VAL so that expressions and functions look and act very much alike. Normally, a function displays the following characteristics: it receives inputs through parameters, defines some environment for execution (e.g., makes new function definitions or creates new identifier-value bindings), and then computes some expression which is its result. The environment defined by the function is only available during the execution of the function, and it disappears when the function returns. Several types of VAL expressions also have this ability to extend temporarily the current execution environment by creating new identifier-value bindings. As with functions, the scope of these bindings is limited to the expression that makes them.

Environment expansion within an expression is specified in VAL by giving a list of identifier-value bindings in a header to the actual expression. Both the type and value must be specified for each identifier; these bindings then remain in force until the final expression is computed. Using this view, it should be clear that each binding must be made to a different identifier (i.e., single assignment). The simplest example of an expression that can extend the execution environment is the **let-in** expression. The example in Figure 2 makes three temporary bindings before the result is generated in the **in** clause. For each binding, the identifier name, type, and value are given. This expression constructs two interval records (*left_interval* and *right_interval*), combines them into one array (*two_intervals*), and, finally, produces one value of type *Result_info* which holds the array under the **tag** "*more*." All of the bindings made at the beginning of this expression disappear as soon as the result value has been computed.

Environment expansions may contain substantial amounts of concurrency. Generally, not all expressions can execute simultaneously, because an expression may use identifiers bound earlier. However, the only limit to concurrency is data dependency. In Figure 2, both record structures can be constructed in parallel before they are entered in the array. While this amounts to very little concurrency, several examples in Appendix A illustrate high concurrency (e.g., the *Compute_Quads* function).

VAL functions and expressions have also been extended to allow them to return more than one value. In conventional languages a function can only return one value. If others are needed, they come back through the parameters or global data. Since both are side effects (unexpressible in VAL), a function is permitted to specify multiple results that it returns on every call. The most common way of specifying multiple results is by writing a comma-delimited list of expressions (other forms are introduced in the discussion of parallel expressions). Multiple results can then be used in a program anywhere that a list of the particular type is expected, such as in multiple binding definitions and as parameters to another function call. The former is illustrated below. The *Compute_Quads* function returns three values: an area calculation, a sequence of intervals that need further processing, and a flag that signals if problems were encountered during the call.

```
new_area    : real,                                    (Int, 12-15)
result_data : Result_list,
abort_info  : boolean
            := Compute_Quads(list);
```

Since the types and order of the return parameters match the types and order of the target identifiers, this is a valid use of a multiple expression.

3.4 Parallel Expressions

One of the most important forms of concurrency available in VAL comes from the **forall** expression. In conventional languages, looping constructs are often employed to compute iteratively values that could be computed in parallel. Sequencing is imposed only because the language and target machine operate sequentially. Array construction is often done this way; each pass through a loop computes and then stores one value within the array. If the calculations do not depend on each other, the array could be built in one highly parallel, nonlooping expression. Another type of array operation can also be sped up by eliminating loops—namely, summing all elements of a vector. In conventional languages vector sums are performed by having each pass of a loop add one element to a running sum. This approach requires linear time. Faster execution can be achieved by eliminating the loop and organizing the calculation as a binary tree of partial sums. This approach requires log time. Both of these types of concurrency are representable in VAL through one language feature called **forall**.

Parallel expressions in VAL have three basic parts: range specification, environment expansion, and result accumulation. In any parallel calculation some operation or set of operations needs to be executed over a range of indices that represent the width of the parallelism. The range specification of a **forall** is a contiguous integer interval or a cross product of contiguous integer intervals that define the scope of the parallelism. The body of the **forall** expression contains an

```
forall i in [array_liml(list), array_limh(list)]                    (CQ, 2-25)
  left:   real := list[i].x_low;
  leftv:  real := list[i].Fx_low;
  right:  real := list[i].x_high;
  rightv: real := list[i].Fx_high;

  mid:    real := (left + right)/2.0;
  F_mid:  real := Evaluate_Function(mid);
  midv:   real := if under(F_mid) then 0.0 else F_mid endif;

  old_area: real := (right - left) * (rightv + leftv) * 0.5;
  new_area: real := (right - mid) * (rightv + midv) * 0.5
                  + (mid - left) * (midv + leftv) * 0.5;

  done: boolean := Stop_Condition(old_area, new_area, right - left);
  abort: boolean := is_error(new_area) | is_error(done);

  eval plus if    abort then old_area
            elseif done then new_area
                        else 0.0
            endif
  construct if    abort then make Result_info[none: nil]
            elseif done then make Result_info[none: nil]
                        else Build_two_intervals(left,  leftv,
                                                 mid,   midv,
                                                 right, rightv)
            endif
  eval or abort
endall
```

Fig. 3. A **forall** expression that generates all elements of an array simultaneously.

environment expansion section identical to the one used earlier in the **let-in** expression. This expansion executes once *for each* element in the **forall** range. At this point N parallel paths of execution can be in progress (where N is the range size), and now the results must be accumulated into a unified object. One form of accumulation, **construct**, allows each parallel path to generate a value which becomes an array element, the position corresponding to the range index associated with that path. In the second form of accumulation, **eval**, values generated from each path are immediately merged into one result using an associative dyadic operator (like **plus**). This merging takes the form of a binary tree evaluation; the leaves contain the values from the various paths, the internal nodes apply the binary operator, and the root node produces the merged result. To avoid inappropriate merge operators, VAL only permits six operators to be used in the merge system. Programmers may use **plus**, **times**, **max**, **min**, **and**, and **or**. For added convenience, a **forall** may produce multiple result values using these two result-accumulation strategies any number of times in one expression.

Figure 3 illustrates one use of the **forall** expression. This expression receives as input a list of intervals requiring further processing using the adaptive quadrature algorithm. During the processing a decision is made whether an interval has an adequate approximation or not. The expression produces three results: the sum of area approximations for intervals where processing is complete, an array of interval pairs requiring further processing, and a flag that is set to **true** if arithmetic errors were encountered during interval processing. The first line of

the expression is the range specification; this expression uses the lower and upper limit of the input array (*list*) as the processing range. The environment expansion section binds values to eleven identifiers. This section executes once for each element of the range, using the range identifier *i* to allow the various paths to access different interval descriptions. The first result accumulation adds together (using **eval plus**) the results of an expression that is computed once for each range path. That expression immediately follows the reserved word **plus**. Informally, then, each path evaluates the **if** expression; if the interval examined by a particular path has an adequate approximation, the **if** expression returns that value. Otherwise, the expression produces a zero. The merging operation then adds these results from all of the paths to produce one **real** value that becomes the **forall**'s first result. The second result accumulation produces an array of values, one element for every parallel path executed. The value of each array position is produced by the expression following **construct**, which is evaluated once for each parallel path. In this case, the value in each array position has a discriminated union type; if an interval has completed processing, then the value has a tag indicating no further processing is needed. Otherwise, the value holds the description of two new intervals that require further processing. The final result merges abort flags from the various paths using **eval or**, thus transmitting any internal processing errors back to the calling function.

The concurrency embedded in the **forall** expression illustrates its expressive power. This expression describes a particular algorithm without imposing any extra sequencing constraints, thus exploiting the parallel nature of the computation. The expression does not impose a particular form of run-time concurrency. Conceivably, this program segment could be translated into graphs using lock-step, vector-type processing; all paths of the parallel expression could execute together moving down through the environment expansion portion together. Alternately, the program could be compiled to treat the environment expansion section as a pipeline, feeding the various range indices into the front of the pipe and letting the various paths overlap execution in that manner. However, the most general execution view of this expression is that each path represents a separate and independent computation that can proceed at its own rate until results need to be merged at the end. Within each of these separate paths, independent operations can further increase the concurrency. In Figure 3, for example, the calculations for *old_area* and *new_area* can proceed together within one path evaluation.

3.5 Sequencing Expressions

While most of VAL is designed to promote concurrency, sequential execution is enforced in some places. Sequencing is imposed either to insure logical correctness or to avoid initiating operations whose results will never be used. These situations can arise with all three VAL conditional expressions. **If-then-else** expressions permit basic selection of expression results. The **tagcase** expression provides a means of interrogating values having a **oneof** type while maintaining type-correctness. The **for-iter** expression implements loops that cannot execute in parallel because values produced in one pass must be used in the next. In all of

these expressions, some conditional clause (**if** Boolean expression, **tag** of a **oneof** type, or the loop restart test) controls activities that will follow. VAL imposes the explicit constraint that no result operation can begin execution until the conditional clause selects the appropriate action.

The **if** expression is akin to the standard **if** statement found in most languages—only in VAL each result clause must be an expression (or multiple expression). A Boolean expression provides selection between a **then** and an **else** expression. To insure static type consistency, VAL requires that the expressions defined by each result option be type-equivalent. If a programmer needs to return two different structures depending on the test condition, he can employ the **oneof** type constructor. An **if** expression with a **oneof** result can be found in Figure 3. The **construct** clause produces an array, where each element is of type *Result_info*. The **then** clause of the **if** produces a *Result_info* value having a "*none*" **tag**; the **else** clause uses another function that produces a *Result_info* value having the "*more*" **tag**. Hence, regardless of the result clause chosen at execution, the type of the result is certain to be *Result_info*.[5]

The **tagcase** expression permits access to information bound to an identifier whose type is **oneof** without violating type-correctness. The strategy is to interrogate the **tag** field to discover the value's true type and then use a multiway branch to select the appropriate action. It is illustrated below:

```
tagcase interval_data := result_data[loc]                    (Bl, 9-13)
  tag none: new_list
  tag more: new_list
            || [1: interval_data[1]]
            || [1: interval_data[2]]
endtag
```

This expression examines a value of type *Result_info* (namely, the value bound to *result_data*[*loc*]), and the result of the **tagcase** depends on the **tag** associated with that value. The assignment in the header is a semantic maneuver to preserve type-correctness. Notice that *result_data*[*loc*] has a well-defined type (**oneof**[*none*: **null**; *more*: *Interval_list*]) established outside the scope of this expression, and that type is different from either of its component types. By binding the **oneof** object to a *local* identifier, the type of that local identifier can change in each **tag** arm. In each **tag** arm the identifier *interval_data* takes on the type associated with the corresponding **tag** field. This permits the information within a **oneof** object to be accessed properly for each possible case. The result of the above **tagcase** is a list of intervals, either the existing list (if the *result_data* had no intervals) or the existing list with the two intervals catenated at the end (each new interval is first transformed into a one-element array to establish type compatability for the catenate operator). As in the **if** expression, all arms of a **tagcase** must generate the same type of expression.

The last major expression in VAL, **for-iter**, permits sequential looping. Its form is an adaptation of LUCID's approach to loops [8] with changes to allow

[5] An alternate solution is to have the **if** expression return an array of intervals. The **then** clause would return an array with zero elements, and the **else** clause would contain two elements. This is legal since type checking does not involve array ranges.

```
for                                                    (Int, 2-29)
  area : real := 0.0;
  list : Interval list
       := (initial interval info.);
  abort code: boolean := false;
do
  let
    new area    : real,
    result data : Result list,
    abort info  : boolean
                := Compute Quads(list);
    new intervals: Interval list
                := Build_list(result_data);
  in
    if array_size(new intervals) = 0
      then area + new area,
           abort code | abort info
      else iter
           area := area + new_area;
           list := new intervals;
           abort_code := abort_code | abort_info;
           enditer
    endif
  endlet
endfor
```

Fig. 4. Illustration of the **for** loop.

the loop to yield a value. In this expression values can be transmitted from one pass through the loop to the next. This transmission can *only* occur by defining loop parameters and then binding new values to them just prior to the beginning of the next pass. Notice that the conventional method of retaining information from one pass to the next, assigning to global variables, is not permitted in VAL. Within a **for** loop, objects can only be bound to identifiers defined *inside* the loop. The program segment in Figure 4 illustrates the use of **for**.

A **for** expression has two separate structural parts: loop initialization and loop body. Loop initialization appears between the reserved words **for** and **do**. All loop parameters (identifiers that can carry information from one pass to the next) must be declared and given initial values here. This portion of the expression executes only once for each evaluation of a **for**. The loop body appears between the **do** and **endfor**. It is an expression that is repeatedly evaluated until a final result can be computed. The decision concerning loop iteration takes place inside some form of conditional expression (in this case, **if**). Normally, all result arms of a conditional must be expressions. Inside a loop body some of these expressions may be replaced by **iter** clauses that cause reevaluation of the entire loop body with new bindings to the loop parameters. If the result arm selected by a conditional contains an expression, then the loop terminates yielding that expression as its result. Otherwise, the result arm must be an **iter** clause that initiates another loop pass.

The **for** loop in Figure 4 provides overall control for the adaptive quadrature algorithm. Three pieces of information must be carried from one iteration

to the next:

area	total area in intervals having acceptable approximations;
list	a list of intervals needing better approximations;
abort_code	a flag that is set to **true** if any interval encounters overflow problems.

The initial values for these identifiers are given in the header (between **for** and **do**). The body of the **for** evaluates a **let** expression. This expression calls *Compute_Quads* to do one more level of analysis on the remaining intervals and then *Build_List* to restructure the remaining intervals into a new list. The value of the **let** expression is a conditional. If the new list of intervals is empty, the loop terminates, returning the total area (previously accumulated area plus the new amount found in the last call to *Compute_Quads*) and the final abort status. If some intervals remain, the **for** body is reevaluated with new values for the loop parameters as specified inside the **iter** clause.

In all of the conditional expressions, concurrency is limited by the controlling Boolean expression. Only the required arm of an **if** or **tagcase** expression executes. Similarly, the next pass in a **for** loop will not begin until an **iter** clause is evaluated. This strategy for handling conditionals may cost some time in terms of execution speed (since evaluation of result clauses cannot be overlapped with the controlling conditional), but it almost certainly saves time because only usable computations are begun.

3.6 Error Handling

An unfortunate result of emphasizing concurrency at the operation level is that it becomes difficult to stop a computation in midstream (the program can be concurrently executing in many places). This problem suggests the need for a different error-handling mechanism from those found in conventional languages. A VAL function can never abort in the middle of its calculation; it must either terminate by generating legal type-correct results or run forever (due to an infinite loop). This characteristic is possible because every data type contains error values in addition to the values commonly associated with the type. When an operator cannot carry out its assigned task (e.g., multiplication underflows or an undefined array element is accessed), it returns the appropriate error value. All operators are defined over these error values so the error information can propagate. Language-defined functions permit testing for the presence of error values so that a programmer can provide alternate results when an anticipated error arises.

The error values defined in VAL attempt to provide accurate information about the cause of a particular problem. Eight specific error values are recognized and handled by VAL:

pos_over, **neg_over**	represent numbers known to surpass the magnitude capability of the machine;
pos_under, **neg_under**	represent numbers known to require greater precision than the machine can supply;

zero_divide is the result of a zero division;

miss_elt is the result of any attempt to access an array element not previously defined;

unknown represents any number whose value cannot be accurately determined due to previous magnitude or precision errors;

undef is produced for all other error conditions.

These error values belong to each of the appropriate basic language types. For example, **undef** is a member of all types, but **pos_over** is only a member of the types **integer** and **real**. This design implies the use of three-valued logic in all Boolean operations; the result of any test can be **true**, **false**, or **undef**. Error values belonging to multiple types are treated as distinct entities to insure type-correctness properties.

All operations defined over the basic language types include provisions for handling these error values should they be given as input. The following operation rules illustrate the actions taken by various operators when error values are received as inputs.

(1) **pos_over + pos_over** yields **pos_over**;
(2) **−pos_over** yields **neg_over**;
(3) **pos_over + neg_over** yields **unknown**;
(4) **pos_over − 1** yields **unknown**;
(5) **unknown** * *X* yields 0 if *X* = 0, otherwise **unknown**;
(6) **miss_elt** / *Y* yields **undef**;
(7) **if undef then** 5 **else** 6 **endif** yields **undef**.

VAL's error propagation rules provide absolutely accurate information, sometimes at the cost of providing less information. The third rule above illustrates how two conflicting error values can be resolved by an operation. The fourth rule demonstrates the hard line taken regarding accurate information; while unlikely, the **pos_over** value may stand for a value one greater than the machine's accuracy; so VAL cannot guarantee that the result of a decrement operation will fit the definition of **pos_over**. The last rule shown illustrates that the language defines the actions of all basic operations over all possible inputs.

In addition to automatic error propagation, VAL provides specific functions that can test for the presence of errors. The special tests are required because under three-valued logic the expression "**undef = undef**" produces **undef**, not **true**. This combination of default propagation and programmer-controlled testing is used in the adaptive quadrature program.

```
F_mid: real := Evaluate_Function(mid);                    (CQ, 8-14)
midv:  real := if under(F_mid) then 0.0 else F_mid endif;

old_area: real := (right − left) * (rightv + leftv) * 0.5;
new_area: real := (right − mid) * (rightv + midv) * 0.5
                + (mid − left) * (midv + leftv) * 0.5;

done: boolean := Stop_Condition(old_area, new_area, right − left);
abort: boolean := is_error(new_area) | is_error(done);
```

If the *Evaluate_Function* produces an underflow value, the second line substitutes a zero to permit continued processing. If any other error condition is generated, the *new_area* calculation will produce some error value. The last line sets the *abort* flag if any significant error arises during either the *new_area* calculation or the *Stop_Condition* evaluation.

For more information on VAL's error value system see [3] or [51].

4. ADAPTIVE QUADRATURE IN VAL

Programming in VAL can best be illustrated by writing and analyzing an example. Adaptive quadrature has been selected because it contains several levels of concurrency, the algorithm is relatively simple to understand, and the solution in VAL gives a balanced view of the language. Three basic parts of the programming process are of interest here. Algorithm selection requires a thorough understanding of how the algorithm works and where the potential concurrencies and "slow spots" reside. Program design stresses correctly mapping the algorithm onto VAL and at the same time insuring that the algorithm's concurrencies are still visible in the data flow graph. Finally, analysis of the program is critical for recognizing changes that would improve accuracy or speed.

4.1 Review of Adaptive Quadrature

Adaptive quadrature is an algorithm for computing the integral of a function, $\mathcal{F}$, over a specified interval. $\mathcal{F}$ is assumed to be continuous over the interval; however, the only way to gather specific information on the function is to evaluate it at various points within the interval. Adaptive quadrature uses a simple approximating function APPROX (e.g., the trapezoidal rule) and a scheme for subdividing intervals (e.g., bisection). The algorithm begins by applying APPROX to the interval as a whole to get one integral approximation and then subdividing the interval and applying APPROX again to each subinterval, summing the results, to get a second integral approximation. The two approximations are compared, and, if they are sufficiently close, the latter one is taken as the integral of the function. Otherwise, the entire algorithm is applied *independently* to each subinterval until acceptable results are found, and then all of the partial results are added for a final result.

The effectiveness of adaptive quadrature in computing an integral relies on its sparing use of calls to $\mathcal{F}$. In most applications, function calls are likely to be very expensive, often completely dominating the cost of the integration calculation. Because each split interval is treated independently, most intervals are likely to converge to a good approximation quickly, without using many function calls. Hence, the cost of function calls is only incurred where the information is likely to make substantial improvement in the approximation. For more information on this algorithm see [19].

One of the open questions regarding this algorithm is the choice of an acceptance criterion for deciding when a particular approximation should be taken. Rice [41–43] proposes a myriad of options that trade accuracy for execution time, the principal differences being in the amount of information required in order to make the accept/reject decision. Simple information, such as the interval's size

and the overall acceptable error tolerance, is sufficient to construct a converging solution; extra information, such as error estimates for completed intervals, may lead to a faster solution.

The important concurrency in adaptive quadrature rests in the independent handling of each of the subintervals. Once the process has iterated through several subdivisions, there are many intervals that can be analyzed simultaneously. In particular, each interval analysis requires one invocation of $\mathscr{F}$ (at the midpoint, when using bisection and the trapezoidal rule); so all of those function calls can execute concurrently. Since these calls dominate the cost of execution, they are almost certain to control overall speed of this algorithm on a multiprocessor.

4.2 Program Design

The goal in implementing an adaptive quadrature program on a data flow machine is to arrange the computation so that all of the operations on intervals can be performed concurrently. One approach for achieving this goal is to use recursion. As the interval splits occur, the independent function invocations can proceed simultaneously. Unfortunately, VAL does not currently permit recursion; so another design scheme must be used.[6] The design implemented in Appendix A uses a list structure (represented in VAL as an array) to keep track of the intervals that require further processing. This structure was proposed by Rice [42] with the idea that in a multiprocessor environment the processors would be constantly taking intervals from the front of the list and adding the split intervals to the rear. The implementation in this paper differs from his design by processing the entire list simultaneously and then constructing a new list with all of the split intervals. The issue of an appropriate interval acceptance criterion is resolved by having the user supply a stopping condition routine in addition to $\mathscr{F}$.

All intervals on the list are processed concurrently using a **forall** expression that returns an array of results (one result for each interval processed). The code for this part of the program is in the *Compute_Quads* function. The range of the **forall** is the entire list, and its body computes the interval analysis for one interval (which includes invocation of the $\mathscr{F}$ function). The result of each interval analysis is returned in an array entry, as specified in a **construct** clause. Concurrency is achieved by the algorithm because the critically expensive function calls can be evaluated in parallel.

The only weakness of this solution is the return of the new list of intervals for the next pass. From the dividing nature of the algorithm we know that the new list may contain anywhere from zero to twice the number of intervals that were input. Unfortunately, the **construct** arm of a **forall** must return an array that has exactly the range of the loop control identifier (i.e., the same length as the input array). To compensate, elements of the output array are **oneof** structures whose contents can vary among array elements. Each entry has either no intervals or two new intervals. Since this structure is not the same as the list of intervals input to the **forall**, a restructuring step must take place. This is done in *Integrate* by calling *Build_list*.

[6] This language constraint may change in the future. There is further discussion of this point later in the paper.

This program also provides a minimum capability for error handling. Trouble can occur in two ways: the user-supplied functions could fail to work properly (affecting an individual interval analysis), or the total area summation could exceed the machine's magnitude capability. The former case is handled by having the *Compute_Quads* function return a flag (*Abort_info*) set to **true** if any interval has problems; the analysis of that interval is stopped, reporting the last valid approximation as the result. Only an underflow error from *Evaluate Function* is corrected without further notice (by assuming the value should be zero). If the total area exceeds the machine's capability, no specific action is taken; *Integrate* simply returns the appropriate error value. Further precautions against errors could be taken if a programmer desired. Here the calling function does not know which interval(s) had to abort due to an error; with further programming this type of information could be returned to the caller.

4.3 Program Analysis

This solution has both good and bad points with respect to concurrency. The program does permit many simultaneous evaluations of the target function. Because $\mathcal{F}$ is assumed to be the most expensive step in the evaluation, this should save execution time. On the negative side, the list restructuring (in *Build_list*) is a sequential operation and, hence, time-consuming. One possible remedy for the future is to add another form of result generator to the **forall**, one which would permit the construction of arbitrary length arrays in parallel. For now, though, we must settle for the position that the sequential portion of the program, while inelegant, is unlikely to dominate the cost of the computation.

Program and language flexibility can be examined by considering how the various acceptance criteria proposed by Rice would affect the concurrency of the program. Rather than discussing each option in detail, we state several principles that should help the reader understand the nature and use of VAL more clearly. The primary concern is to identify those pieces of information that can and cannot be passed to some "*Stop_Condition*" function. As the program is currently structured, the decisions about interval splitting are made during the processing of each interval; thus, they are done in parallel. At that point, only information on the current interval and information from the previous passes can be used. For example, instead of providing the interval's size, any or all of the following information could be passed (assuming appropriate changes in the program to accumulate these statistics):

(1) interval endpoints;
(2) partial area accumulation as of last pass;
(3) partial error estimate as of last pass;
(4) number of intervals remaining in the list.

These values could be useful in designing different stopping conditions. Notably absent from this list is any information about analysis on the current pass through the intervals. It may be that almost all of the intervals in the current pass get very accurate estimates, and so the few remaining ones can afford to be less precise. That information will not be available until the next pass, because the analysis of all intervals is concurrent.

An alternate solution design would permit information about the current pass results to be available immediately but at the expense of some speed at execution. The decision on whether to split an interval or quit could be moved to the *Build_list* routine. This routine is already basically sequential; so the move would not drastically change the execution time. The **for** loop could be changed to include some loop variables that keep information on the latest error and area estimates. The cost of this approach is that the decisions on whether or not to split each interval would now be done sequentially rather than simultaneously. Notice, however, that this cost is directly related to the type of algorithm that is desired. A stopping condition that needs absolutely current information about the state of the approximation imposes more constraints on the solution.

In summary, one of the main concerns of a programmer using VAL is to understand the timing constraints of various solutions and to find an implementation that runs as quickly as possible. In the adaptive quadrature problem, the critical concurrency must be the multiple function evaluations. The proposed solution permits this concurrency with only a modest amount of effort. The program variations to accommodate different stopping conditions can be done *with* the corresponding concurrencies available in each choice. The only sequencing constraint introduced by programming in VAL is the list restructuring at the end of each pass.

5. LANGUAGE CRITIQUE

With the preceding description and examples of VAL, a rather thorough analysis of the language can be given. This section analyzes VAL using three separate perspectives. The most important view examines the language definition—its computational power and limitations, its ability to represent concurrency, and its formal semantics. The second view of VAL discusses programmer use—how this environment influences the task of writing effective programs. The third viewpoint considers translation of VAL to various machines. These different perspectives give a balanced analysis of VAL.

Any analysis of VAL must take into account the motivations and goals of the designers. VAL was intended to be a small research language that would give everyone an opportunity to experiment with data flow computing. To keep the language small, the only features placed in the language were ones deemed critical for expressing algorithms with their inherent concurrency. Features such as abstract data types and array processing were omitted from the initial definition pending further evidence that their inclusion would significantly enhance VAL. Other features were omitted because their normal definitions interfered with the basic language structure and more appropriate methods were being sought. Input/output is one feature in this category; it is discussed in more detail later in the analysis. For these reasons VAL should be viewed as an evolving language; with more experience the design will grow and improve. This analysis addresses the current state of VAL and indicates where changes may occur.

5.1 Language Definition

The VAL language definition contains some interesting points from both a practical and theoretical viewpoint. The VAL definition is complete (including

provisions for coping with machine arithmetic); rarely, if ever, does a language definition include such detail. The use of expression-based features and implicit concurrency in VAL provides substantial power for writing programs intended for highly concurrent execution environments. Strict adherence to functions without side effects significantly reduces the complexity of formal language definition.

The reference manual for VAL specifies the result of every program that meets all syntactic and semantic rules. Since these rules can all be checked at compile time, any program that begins execution has a well-defined result. This characteristic provides a measure of uniformity in implementations. No aspect of the language is left "undefined" and hence open to different interpretations by compiler writers. The only semantic differences that can arise in the execution of a VAL program occur when running on machines with different arithmetic capabilities. One execution may produce a numeric result while another yields one of the special error values. However, the arithmetic rules try to prevent two machines from producing two distinct numeric results when executing the same program.

The use of implicit concurrency and implicit synchronization gives VAL its expressive power. VAL has no features for forcing any kind of concurrent activity; it simply does not require a programmer to add sequencing constraints beyond those imposed by the algorithm. This aspect is shown best in **forall**; its semantics clearly demonstrate that the expression is not a loop, and hence it contains substantial concurrency. A compiler can decide what forms of concurrency to exploit. This is a distinct advantage over languages like LRLTRAN [34] (LLNL's version of FORTRAN for CDC 7600s, CDC Stars, and Cray I's) and GLYPNIR [31] (a language for the Illiac IV) because they focus exclusively on vector concurrency to the exclusion of all other forms.[7] VAL also has implicit synchronization; among other features, the **eval** and **construct** clauses in **forall** imply significant synchronization. In most other concurrent languages, notably Concurrent PASCAL [11] and MODULA [53], programmers must use explicit tools (e.g., monitors or semaphores) to coordinate this interaction. The net result for VAL is a small language that is easy to use when writing concurrent programs.

Relative to other data flow language efforts, VAL goes further to support implicit concurrency and machine independence. The ID language [7] defines program concurrency in terms of an unraveling interpreter model. That model describes program execution by first translating programs to graphs and then using a highly concurrent graph execution model. A programmer must know how translation proceeds to identify where concurrency is possible. For example, VAL's **forall** expressions would be written in ID as loops. These loops unravel into highly concurrent activities, but a programmer may not recognize this fact due to the loop structure. The definition of VAL is completely independent of any machine structure (real or model). The reference manual does not rely on graphs or data flow machine execution. It happens (by design) that VAL is highly suitable for a data flow architecture, but it is certainly not limited to implementation on that type of machine.

[7] This weakness is not limiting in their current environment because the machines mentioned only have the power to exploit "vector/array" concurrency.

Although the original definition of VAL uses informal semantics, the language is amenable to all of the current techniques for formal semantic definition. Gehani and Wetherell [22] have written a complete denotational semantic definition for VAL. This work includes a thorough definition of all arithmetic operators (parameterized for different machine precisions). This application of denotational semantics was easy to write (according to the authors) because the language definition prevents the expression of any side effects. Concurrency never enters the semantic definition because only implicit forms of concurrency are used in the language and determinate behavior insures that all different execution patterns must generate the same result. Brock [12] has applied operational semantics to a subset of the VAL language. He used the obvious computational model (data flow graph execution) and provided a translation algorithm for generating graphs from the various language features. The language subset includes basic and iteration-type expressions; extensions to include the remaining VAL features do not appear to pose serious problems.

Finally, Ackerman [1] has applied axiomatic semantics to a VAL-like toy language. This application of semantics is interesting because axiomatic definitions normally employ state descriptions that represent an environment before and after the execution of each statement. Since VAL uses expressions instead of statements, this may appear to be a mismatch of ideas. Actually, many ideas remain the same; VAL binds values to identifiers (which is a restrictive form of assignment), and it contains looping expressions (which provide power similar to that provided by loops in conventional languages). Ackerman shows that the additional VAL-like restrictions simplify assertions, thus reducing complexity for potential users. As one example, assignment statements like $j := j + 1$ force axioms to recognize two different values bound to j at different times. VAL disallows this difficult form through the single-assignment rule, thus eliminating the need for axioms to represent timing conditions.

VAL also has interesting connections to the research on functional programming (FP) systems exemplified by Backus' work [9]. These efforts are similar in that both exclusively use functions and expressions and neither allows side effects. They differ in at least two major ways. VAL explicitly deals with types: each function is defined only over the domain of the input types. FP systems do not have types; so functions are defined over the universe of inputs. Also, FP systems include *functional* operators (operations that produce functions as their result); VAL does not have this power. The interesting connection is that many of the "Laws of the Algebra of Programs" identified by Backus hold true for VAL programs. One simple case illustrates the similarity. Backus [9] proves that in FP systems,

> for all functions F, G, and H and all objects x:
>
> $$([F, G] \circ H) : x \equiv [F \circ H, G \circ H] : x.$$

The equivalent "law" for VAL is the following:

> for all type-compatible functions F, G, and H and all type-compatible objects x:

$$\left.\begin{array}{l}\textbf{let}\\ \quad temp\text{: ``}type\text{''} := H(x);\\ \textbf{in}\\ \quad F(temp), G(temp)\\ \textbf{endlet}\end{array}\right\} \equiv F(H(x)), G(H(x)).$$

Note that in FP systems we normally reason at the function level (i.e., we can omit all references to objects like x) and prove the same result. In VAL we must reason at the object level, considering the inputs and their type compatibilities. Nonetheless, this tie between VAL and FP systems implies that proof techniques developed for FP systems may be directly applicable to VAL as well.

An interesting issue was recently brought to light when one of the theorems in FP systems was found *not* to hold for VAL. In FP systems composition distributes across conditionals; the formal notation is

$$H \circ (p \rightarrow F; G) \equiv p \rightarrow H \circ F; H \circ G.$$

The corresponding law in VAL,

$$H(\textbf{if}\ p(x_1)\ \textbf{then}\ F(x_2)\ \textbf{else}\ G(x_3)\ \textbf{endif})$$
$$= \textbf{if}\ p(x_1)\ \textbf{then}\ H(F(x_2))\ \textbf{else}\ H(G(x_2))\ \textbf{endif}$$

(where x_i is any arbitrary sequence of parameters),

does not always hold. Differences arise when the call to p returns an error value. In the first VAL expression, H is still invoked and has the opportunity to test for an error input value and take corrective actions. The second expression always returns an error value if the function p yields an error value. At issue here is the exact meaning of an error value. VAL's definition of **if** on an **undef** predicate treats the error value like "bottom" in logic (the result of a function on an undefined input must be undefined); however, VAL's ability to test for error values treats them like any other values in the language. Several plausible responses to this issue have been suggested: leave the definition alone and live without this "law"; equate error values with **false** in **if** evaluations; or define a trap "otherwise" clause for **if** that is not optional and that is evaluated when the predicate produces an error value. At this time it is not clear which choice is best.

As with most languages, VAL does have a few weaknesses in its definition. Most of these problems are due to insufficient experience with the target machine environment, namely, data flow computers.

The most serious weakness in the VAL definition is an omission of general input/output facilities. Under the current definition each function receives all of its input values and then produces some result or results. This setup does permit the most primitive I/O, namely, batch I/O. When a VAL function is called, it can receive a file that contains all input it will need, and the function can produce a file or files containing all of the output it wants to generate. Notice that no I/O is actually done within the function itself. Presumably, the system environment would accomplish all reads before invoking a VAL function and generate all output on termination. This design has several serious limitations. First, requiring all input prior to execution prohibits a user from interacting with a program while

it is running. Second, no output prior to program termination precludes overlapping computation with I/O, thus increasing the need for space during execution and increasing the time before a user can see the final results. This design also precludes the use of VAL on any database applications because the program cannot continually update any external files.

Input and output were omitted from VAL because their standard definitions employ side effects and this would conflict with the basic language structure. Input is not functional because repeated invocations with the same parameters produce potentially different results. Output is not functional because each invocation modifies some global environment (the output file). The problems become clear if we assume VAL contains conventional read and write routines. Referring to the adaptive quadrature program, what would happen if the *Evaluate_Function* function tried to read or write? Since this function can have many independent simultaneous invocations, what reasonable order can be placed on the distribution of input data and the accumulation of output results? None is reasonable without imposing some well-defined semantic execution sequencing on all operations that otherwise have substantial concurrency. Rather than immediately surrendering to a bad choice, VAL's designers included no specific I/O features, and work is continuing to find a more suitable answer.

A second serious weakness in VAL is the lack of recursion. In the data flow environment recursion is a natural and powerful programming tool. Most recent languages support recursion because programmers can write shorter and clearer programs without incurring much (if any) run-time penalty. The overhead associated with recursion (developing reentrant code and managing a stack environment) is absorbed by the compiler and run-time support system. In a highly concurrent environment recursion provides another potential advantage by increased simultaneous activity. Many divide-and-conquer algorithms (e.g., quicksort) divide a problem into two parts and recursively call the function on each half. Concurrency accrues because both calls can proceed together, leading to significant concurrency as the calls branch into a binary tree structure. The program in Appendix B illustrates the significance of prohibiting recursion in VAL. This recursive function could replace all functions declared in *Adaptive Quadrature*. It is computationally equivalent to the original, obviously shorter, and potentially faster because this version need not complete all analysis at one level before any computation begins at the next level. In fairness, however, it is important to notice that a recursive version of adaptive quadrature imposes more limitations on the type of information that could be given to a "*Stop Condition*." Only information on the current interval (e.g., size, bounds, and area estimates) and initial constraints (e.g., desired error bounds for the entire problem) can be used in a recursive version.

The designers of VAL chose to omit recursion because of problems envisioned in implementation, not because of any interference with the remainder of the language. The initial target machine environment has no support for dynamic operations like recursion; the graph language does not allow expansion of the graph during execution (there is no "call" operator that can replace a graph node with a function graph). It may have been possible to find a way to implement recursion in a static system using techniques comparable to using limited-depth

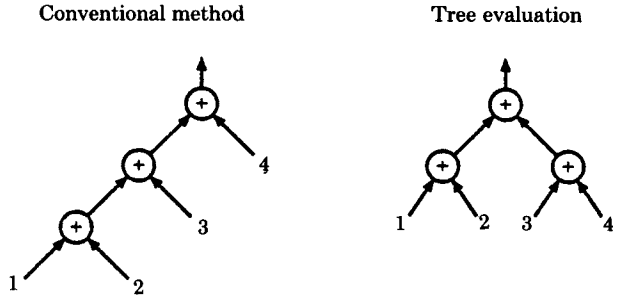

Fig. 5. The sum 1 + 2 + 3 + 4 can be accumulated in two ways. The left figure uses the standard method, summing left to right. The right figure uses a tree structure to decrease evaluation time by one time unit.

stacks in most current compilers; however, all of the proposed solutions fail to reflect the natural concurrency. Since no reasonable implementation was known, the designers chose to omit it and see how much it would hamper programming. As the adaptive quadrature example illustrates, the cost is significant.

Another relatively minor problem in the VAL definition can be found in the description of the **eval plus** operation that is used to accumulate results inside **forall**. The idea behind this form of result accumulation is that, if a sequence of values needs to be merged into one result (such as adding all elements of the sequence) and if the merge operation is associative, ordering the accumulation in a tree fashion would speed up execution without changing the result. Figure 5 illustrates the difference. Unfortunately, in machine arithmetic addition is not associative because of possible overflows; the linear evaluation may produce a valid numeric result while the tree evaluation produces **unknown** (due to an intermediate overflow), or vice versa. The same problem holds for **eval times**. The VAL reference manual does not specify which method of evaluation is used because different compilation strategies could benefit from both. However, as long as the two forms can produce different results, the language definition fails to be sufficiently precise.

Another observation (that some would feel is a weakness in VAL) is that expressions and function calls are not always interchangeable. At issue here is the precise method of evaluation used for every expression. *Strict* evaluation requires that all inputs to an expression be evaluated before the result is produced (even if some inputs are not used). *Lazy* evaluation forces each parameter to be evaluated after it is certain the value will be used. VAL uses both of these evaluation forms in different situations. The **if**, **tagcase**, **forall**, and **let** expressions use lazy evaluation. The first two are done this way to avoid starting unnecessary calculations; the latter two are done this way to simplify graph translation and to avoid extra synchronization at the beginning of an expression. On the other hand, the **for** expression and function calls use strict evaluation (the latter implied by the use of call by value). These choices were made to simplify translation to a particular type of data flow machine that allows several sets of tokens to pass through the same graph. At the language definition level, the significance of these different evaluation schemes is in one minor type of

situation where apparently equivalent expressions can produce different results. The following function illustrates this point.

```
function if_fun(B: boolean; P, Q: real returns real)
  if B then P else Q endif
endfun
```

This function is not equivalent to VAL's **if** expression because, if either P or Q should generate an infinitely looping computation, the function is guaranteed to run forever while the simple **if** expression has a chance to halt (B may select the halting computation).

This difference has both good and bad points for VAL. The negative aspect is that in special cases like the one above VAL lacks *referential transparency* (function calls cannot be used interchangeably with expressions). For the most part, functions and expressions can be interchanged safely, and it would be nice to have that property uniformly true. The positive aspect of providing both strict and lazy evaluation is that programmers can often select which form they want (in those few cases where it might make a difference in execution speed). For example, the above function, *if_fun*, can improve execution speed if B, P, and Q all take a long time to execute and if extra processors are free to do the work.

One final observation about the VAL definition should be made. VAL was designed to implement deterministic algorithms. This means that time-dependent algorithms (such as an airline reservation system) and strictly nondeterministic algorithms (such as a truly random walk across some grid space) cannot be written in VAL. The rationale for this limitation is twofold. First, mixing massive concurrency with nondeterministic programs greatly compounds problems for users. Insuring program correctness (both formally and through testing) is certain to be more difficult. Second, the class of algorithms precluded by this restriction, while interesting, does not appear to be crucial for solving highly computational numeric problems. Since the driving force in the search for faster computers is the need to solve these problems, nondeterminism has not received much attention in VAL.

5.2 Language Use

The VAL language influences almost all facets of the programming process. In the design phase programmers must adapt to a structured approach; the only program interfaces are at function calls, which limits the types of interaction that various pieces of code can have. Also, as illustrated in the adaptive quadrature program, the design phase must begin to address the issue of concurrency; the critical concurrency must be identified and carefully mapped into the program. In the writing phase programmers need to adjust to a different set of language tools that may be uncomfortable at first. Once beyond these initial problems, programmers do have a better mind-set for writing concurrent programs. In the debug phase programmers must insure both correct and fast execution. Here, adequate language support is only beginning to emerge.

Programming in VAL requires users to adapt to a new writing style that emphasizes values, expressions, and functions instead of variables, statements, and procedures. The difficulty in transition depends heavily on each individual's

background; long-time FORTRAN programmers are likely to spend more effort in a transition than more recently trained persons with varied language experience. Very few, if any, of the language features are totally new; the expression-based features are similar to LISP and BLISS, the **forall** is a common vector operation, and **tagcase** came directly from CLU. The **for-iter** expression, which has a strong resemblance to a similar feature in LUCID, probably poses the most serious learning problem for beginners. Separating the iteration control from the expression header hampers both the understandability of the expression and the overall readability of the program. Fortunately, our limited experience with students and a few career programmers shows that the transition to this programming style can be made and that the obstacles are short-lived.

Once programmers adjust to this style of programming, they have substantially better tools for writing highly concurrent programs. The most important tool is **forall**. This feature tends to replace "do loops" in most other languages, with significant gains in concurrency. An unclassified version of a large and heavily used program at LLNL (called *Simple* [16]) has been translated from FORTRAN to VAL. The original version contains 76 do loops, most of which are in pairs to scan across two-dimensional arrays. The VAL version uses 19 **forall**s and only 8 **for-iter**s (the **forall**s absorb most of the do's, because each one can range across both dimensions). These **forall** expressions generate massive amounts of concurrency, while the programmer tends to write somewhat less code. Some of the concurrency in the VAL version represents vector/array parallelism that an optimizing compiler can find in the FORTRAN version; however, the VAL version goes significantly beyond this level. Moreover, programmers can be certain that the VAL compiler recognizes all available concurrency where they may be unsure about an optimizing compiler's results. Obviously, one sample FORTRAN-to-VAL translation cannot be considered "proof" of VAL's superiority, but its success on a *real* problem is significant.

One language component not presently in VAL which will enhance its usability is a set of features for array processing. Array processing was originally left out of VAL because **forall** covers almost all of the concurrencies array processing can provide, but at a more general level. Also, there are many unanswered questions about the best way to provide such a system within a language. In translating programs like *Simple*, the program readability could have been improved and the length shortened with even simple array features. As an illustration, compare expressions for summing all elements of a two-dimensional array *Beta* done in APL and VAL:

APL	VAL
+/ +/ *Beta*	**forall** *i* **in** [**array_liml**(*Beta*), **array_limh**(*Beta*)] *j* **in** [**array_liml**(*Beta*[*i*]), **array_limh**(*Beta*[*i*])] **eval plus** *Beta*[*i*, *j*] **endall**

Some computer scientists would argue that APL is too cryptic, but most would probably agree that VAL is overly wordy in this case. Efforts are in progress to establish a reasonable set of array processing features. The APL manual [25] and

a paper by Wetherell [52] on this subject are likely to have a significant influence on the design.

Possibly the most significant unknown in the VAL design is the usefulness of the error value system in helping programmers write and debug programs. The initial motivation for using error values was to make the language complete; the actual choice of error values and propagation rules attempts to address programmer concerns about errors and error recovery. As illustrated in the adaptive quadrature program, a very few explicit tests for error values and automatic error propagation allow minimal error-handling capability. However, the utility of this particular design to users is unknown. The strict adherence to a meaning for errors (e.g., **pos_over** − 1 = **unknown**) may negate the benefits of having different error values because propagation tends to quickly transform them into **undef**.

The general strategy for handling run-time errors is still only of marginal use when a programmer is faced with an incorrect program. It is unreasonable to insert program text to test the validity of every operator, yet failing to do so may obscure where an error actually arises. One proposal (which is not yet officially in VAL) is to have the language define a special audit trail associated with each error value. This trail would provide information about where an error originated and what transformations it went through during propagation. Since this audit trail would constitute an unusual side effect if it were accessible within the program, one constraint on this information is that it could only be dumped to some system file for later access.

Even the traceback log has limits to its usefulness. It only identifies the source operation that caused the initial error; a programmer may need to trace back several steps prior to the error to actually determine the cause of a problem. Traceback logs cannot provide this level of help. One possible solution for more thorough debugging is to define a canonical program execution for sequencing all operations. Running the canonical version on a sequential machine (admittedly very slow) would allow programmers to breakpoint execution at meaningful places. Because VAL has completely deterministic behavior, any errors that occur in the concurrent version must also appear in the sequential one.

The last critical programmer concern is execution speed. In VAL programmers can identify concurrency by applying the simple rule that two operations can proceed together unless they are related by a data dependency or if either is limited by a control dependency (i.e., **if**, **for**, or **tagcase**). Furthermore, **forall** expressions automatically imply independence across range elements. In practice these rules may be sufficient for studying local concurrency potential, but for complete program evaluation this seems inadequate. An interesting research project would be to develop a means for doing automatic program analysis and presenting the results to programmers in a meaningful way.

5.3 Language Translation

Translation of VAL programs to various machines depends heavily on the target system. For data flow machines, most of the basic VAL statements can be mapped across with no difficulty. Depending on the machine's capabilities, some of the more dynamic aspects of VAL could pose problems. Another interesting option

for VAL is translation to conventional machines. While not designed for these architectures, the available language concurrency may be able to improve the machine's usage.

Most parts of the VAL language can be translated easily into graphs that can be executed on almost any proposed data flow machine. As mentioned earlier, all type checking can be done at compile time. Translation can use either LR or recursive descent parsing techniques. The simplest approach to graph generation is to have the translator bind each program identifier to the output arc that produces its value. By the language rules there can be only one such binding; so no conflicts can arise here. As an expression is translated, each use of an identifier can be mapped to its source arc and used as input to the expression's graph. Higher level expressions like **if**, **let**, **tagcase**, and **for-iter** can be mapped into graphs using simple graph patterns that are available but often different among data flow machines. This approach not only accomplishes the translation; it also automatically exposes the available concurrency. Going back to Figure 1, where a small VAL program and its graph are shown, this translation is produced with no special concurrency analysis.

The difficult parts of translation arise in VAL's dynamic features. A VAL translation can go smoothly if the machine supports *dynamic graphs* [18] (the machine graph language contains some "call" operation that can be replaced by a function subgraph during execution) or if the machine supports *token labeling* [7, 50] (multiple function invocations can simultaneously execute within a single copy of the function's graph without delay or confusion). Even though VAL has no recursion, other aspects of the language are highly dynamic. For example, the range of the **forall** in Figure 3 is only computable at execution, and, furthermore, that range varies drastically over different invocations. Either of the dynamic machine features mentioned above can represent this aspect of VAL in a simple and natural way. The disadvantage with these machine designs is the significant possibility for deadlock created by the dynamic allocation of graph (or token) space. The adaptive quadrature program in this paper has a highly dynamic nature that is data dependent; it could easily consume all resources on any reasonably sized machine and then deadlock. If translation fails to address this issue, then an extraordinary burden must be placed on an architecture to resolve this conflict.

On a static architecture [15, 21] the VAL dynamics must be addressed solely by the translator. Pipelined executions of the **forall** would be required unless program analysis could compute the range during compilation (in which case the body of the **forall** could be replicated, effectively vectorizing the calculation). The cost of pipelined execution is reduced execution speed; although all range elements can logically proceed at the same pace, they do not do so in the piped version. One partial solution to this problem is to have the translator set up several parallel pipes, thus combining the vector and pipe solutions. This kind of work naturally leads to the general question of optimization.

Optimization techniques for the efficient use of a data flow architecture will probably require entirely new approaches. Brock and Montz [13, 36] describe some simple optimizations for data flow graphs. However, there is much more work to do. In VAL expressions like **forall** the translator is faced continually

```
X: array[real] := forall i in [1, N]
                    construct F(i)
                  endall;
Y: array[real] := forall j in [1, N]
                    construct G(X[j])
                  endall
```

Fig. 6. Two VAL expressions that can have substantial overlap at execution if translated carefully.

with the question of whether to sequence, vectorize, pipeline, or independently compute all elements in the range (this applies to both static and dynamic architectures). The trade-off is between space and speed. While speed is the driving force in this work, it gains nothing to allow orders of magnitude more concurrency than a machine can exploit. Such situations can easily arise when writing large programs (like *Simple*) that operate on arrays containing several thousand elements. In these cases the translator should recognize the lack of speed gain and save space. At other times, an optimizer may need to stretch to find more concurrency to keep the system busy. Woodruff [54] identifies one potential slow spot in using VAL that need not be there. Consider the two **forall** expressions in Figure 6. Because the second expression uses values calculated by the first one, VAL seems to impose sequencing by having the entire array X calculated prior to any evaluations for Y. In fact, each range element, j, in the second expression only uses one value of X; so each value for $Y[j]$ can begin execution as soon as $X[j]$ is known, thus allowing significant overlap between the two expressions. For completeness, it should be noted that the example in Figure 6 can be optimized by folding the two expressions into one **forall** that produces two arrays. However, the general case of this problem does not have such a simple answer.

Finally, while the VAL language was designed for a data flow architecture, translation to a conventional machine seems reasonable. A VAL program describes some calculation with a minimum amount of sequencing that is imposed by the algorithm. Since most machines could not exploit all of this concurrency, a translator would need to organize execution to suit the particular machine. For efficient organization a translator requires substantial information about the program (e.g., which pieces of code are independent of each other) to best arrange the calculation. In most languages this information comes at great cost after substantial analysis and usually produces incomplete results. In VAL the necessary information is explicit in the graph form. Furthermore, the various forms of concurrency are visible, and they can be exploited on architectures designed for these purposes. At LLNL, we are looking at possibilities for experimentally putting VAL on a Cray and on the S-1 (a general multiprocessor system). Neither of these machines is well matched to data flow concepts, but the extra concurrency information a VAL translator could extract from programs (over translators for FORTRAN or PASCAL) might lead to surprising execution speeds.

The two major problems with translation to a conventional machine involve dynamic arrays and the error-handling system. VAL's liberal use of dynamic objects requires an efficient run-time memory management system for the allocation and recovery of space. Very few machines handle this task well. To make matters worse, the overall machine performance almost always depends on a careful layout of the data in memory to reduce memory contention. Automatic

memory management systems will be hard pressed to handle this problem well. The second problem, implementing error values, poses an even more difficult problem. We assume that a data flow machine will handle error values within the hardware. Since current machines do not have this capability, it would need to be done in software: a very expensive strategy.

6. SUMMARY

The VAL language is part of a response to a general need for more computing power and more effective use of computers. We have passed the time when cheap orders-of-magnitude speedup can be achieved solely through hardware technology. At least partial responsibility for greater speed must fall on software and users. The data flow approach to high-speed computing provides a viable architectural alternative; however, it requires highly concurrent data flow graphs in order to achieve reasonable performance. VAL gives programmers a new and comfortable way of expressing many forms of concurrency. The language takes responsibility for mapping all available program concurrency into the graphs but gives to users the ultimate responsibility for selecting highly concurrent algorithms.

Several important characteristics of VAL differentiate it from other languages. The expression-based features prohibit all forms of side effects and simplify the task of identifying concurrency. Also, VAL has no special features for explicitly specifying concurrency; all concurrency and synchronization are implicit. These two traits separate VAL from most non-data-flow languages. The extensive error-handling system, which embeds special error values in each basic data type, and the **forall** expression (with its unique **eval** clause) give VAL distinct advantages over other data flow languages. Programmers have a greater ability to see and express concurrency in algorithms, and they have more flexibility in dealing with errors.

At the present time, VAL is a small research language. It demonstrates that data flow execution concepts can be mapped into a high-level language, giving programmers an opportunity to exploit this new type of computing machine effectively. Even with "minimal" language features, significant computational algorithms like adaptive quadrature (presented here) and two-dimensional hydrodynamics (*Simple*) have clear and highly concurrent program representations in VAL.

For VAL to evolve into a truly useful and practical language tool, several critical areas of work need to be carried out. First, VAL needs more general facilities for managing input and output operations. Most data flow languages have stalled on this issue, but current research efforts at M.I.T. and LLNL are nearing a solution. Second, translation and optimization techniques require further study. VAL could be useful on conventional as well as data flow machines. To insure effective system use the mapping between the language and any target machine must be well planned. Finally, further program analysis tools need to be developed to help programmers evaluate the performance of their programs on the machines. Efforts are under way in all of these areas, with the ultimate goal that VAL or a descendant of VAL will provide the language component of a solution for high-speed computing needs.

APPENDIX A. ADAPTIVE QUADRATURE PROGRAM

```
function Adaptive_Quadrature(low, high: real returns real, boolean)
  % Adaptive_Quadrature computes the integral of a function, F, on the interval
  % low .. high (the inputs). If any interval calculations blow up, the previous best
  % interval estimate is used, and the second result (a boolean) is set to true. The user
  % must provide two functions; their characteristics are described in the external
  % declarations below. Most design and implementation details are in Section 4 of the
  % paper. Comments at the beginning of each function explain the operation within that
  % function.
    type Interval = record[x_low, Fx_low, x_high, Fx_high: real];
      % An interval is represented by its endpoints and the values of the function at
      % those endpoints.
    type Interval_list = array[Interval];
      % A list of intervals is represented by an array of intervals. The list always begins
      % at the 1 index position and extends as high as necessary.
    type Result_info = oneof[none: null; more: Interval_list];
      % The result of analyzing an interval may be that no new intervals are generated
      % (none tag) or that two new intervals are generated (more tag, list has 2 entries).
    type Result_list = array[Result_info];
      % A result list holds all results from the processing of one interval list. Notice
      % that each array entry is a oneof type, and the tags need not match between
      % entries.
    external Evaluate_Function(x: real returns real);
      % This must be a user-provided function for computing F(x), where x is in the
      % range low ≤ x ≤ high.
    external Stop_Condition(area1, area2, interval_width:
                                    real returns boolean);
      % This must be a user-defined function for deciding whether area1 and area2 are
      % close enough to permit the latter to be used as the approximation for an interval
      % of width interval_width. The program could easily be modified to allow this
      % function access to other information.
      %    To improve readability, each function is presented separately, at the same
      % syntactic nesting level. This structure is not legal in VAL because functions are
      % not permitted to call other functions declared at the same level. These functions
      % could easily be reorganized into a correct VAL program, if necessary.
    function Integrate(low, lowv, high, highv: real returns real, boolean)
      % The adaptive quadrature algorithm is implemented by keeping a list of intervals
      % needing a better approximation, a partial area sum for completed intervals, and
      % a flag that is set if any interval encounters computational problems.
      %    This function loops until the interval list contains no more intervals. On each
      % pass through the loop, Compute_Quads analyzes each interval, reporting an
      % area sum for completed intervals, a new set of (subdivided) intervals for those
      % needing further processing, and a flag that is set if any interval analysis encoun-
      % ters problems. The Build_list function restructures the divided intervals into
      % a new list. When all intervals are done, the total area and error flag are returned
      % as the results.
      for
        area: real := 0.0;
        list : Interval_list
             := [1: record[x_low:  low;
                           Fx_low:  lowv;
                           x_high:  high;
                           Fx_high: highv]]
        abort_code: boolean := false;
      do
```

```
        let
          new_area      :  real,
          result_data   :  Result_list,
          abort_info    :  boolean
                        :=  Compute_Quads(list);
          new_intervals :  Interval_list
                        :=  Build_list(result_data);
        in
          if array_size(new_intervals) = 0
            then area + new_area,
                 abort_code | abort_info
            else iter
                 area := area + new_area;
                 list := new_intervals;
                 abort_code := abort_code | abort_info;
                 enditer
          endif
        endlet
    endfor
endfun

function Compute_Quads(list: Interval_list returns real,
                       Result_list, boolean)
  % Compute_Quads receives a list of intervals for which acceptable area approxi-
  % mations have not been found. For each interval (all are done in parallel) the
  % midpoint and function value at the midpoint are computed. The trapezoidal rule
  % is applied separately to each half-interval and once to the interval as a whole. If
  % the two approximations are acceptable (as defined by the Stop_Condition),
  % that area is added into the other acceptable areas for this list. Insufficient
  % approximations return two subintervals to be operated upon later.
  %    The initial assignments below (left . . . rightv) are purely for readability. Areas
  % are only accumulated for acceptable approximations (done in eval plus). Due
  % to strong type-checking, the list of results must be elements of a oneof type
  % representing either
  %
  % (1) no new intervals (none option) or
  % (2) two new intervals (more option).
  %
    forall i in [array_liml(list), array_limh(list)]
      left:    real := list[i].x_low;
      leftv:   real := list[i].Fx_low;
      right:   real := list[i].x_high;
      rightv:  real := list[i].Fx_high;
      mid:     real := (left + right) / 2.0;
      F_mid:   real := Evaluate_Function(mid);
      midv:    real := if is under(F_mid) then 0.0 else F_mid endif;
      old_area:  real := (right - left) * (rightv + leftv) * 0.5;
      new_area:  real := (right - mid) * (rightv + midv) * 0.5
                       + (mid - left) * (midv + leftv) * 0.5;
      done:  boolean := Stop_Condition(old_area, new_area, right - left);
      abort: boolean := is error(new_area) | is error(done);
      eval plus if      abort then old_area
                elseif done then new_area
                            else 0.0
                endif
```

```
        construct if      abort then make Result_info[none: nil]
                  elseif done then make Result_info[none: nil]
                               else Build_two_intervals(left,  leftv,
                                                        mid,   midv,
                                                        right, rightv)
                  endif
        eval or abort
      endall
    endfun

    function Build_list(result_data: Result_list returns Interval_list)
      % Build_list takes a list where each element is either
      %
      % (1) empty or
      % (2) a pair of interval descriptions
      %
      % and returns a list whose elements are the nonempty intervals from the input.
      %    This implementation examines each of the input elements sequentially, and,
      % for each pair of intervals found, it catenates them one at a time to the list that
      % will eventually be returned. The actual catenate operation looks unusual because
      % each element must be converted to a one-element array in order to insure type-
      % correctness for "||", which requires that all operands be arrays.
        for
          new_list: Interval_list := empty[Interval_list];
          loc: integer := array_liml(result_data);
        do
          if loc > array_limh(result_data)
            then  new_list
            else  iter
                  new_list := tagcase interval_data := result_data[loc]
                                tag none: new_list
                                tag more: new_list
                                          || [1: interval_data[1]]
                                          || [1: interval_data[2]]
                              endtag
                  loc := loc + 1;
                  enditer
          endif
        endfor
    endfun

    function Build_two_intervals (left, leftv, mid, midv, right, rightv:
                                  real returns Result_info)
      % This is a utility function which takes information on the analysis of an interval
      % and builds a list consisting of the two subintervals.
        let
          left_interval  :  Interval
                         :=  record[x_low:  left;
                                    Fx_low: leftv;
                                    x_high:  mid;
                                    Fx_high: midv];
          right_interval :  Interval
                         :=  record[x_low:  mid;
                                    Fx_low: midv;
                                    x_high:  right;
                                    Fx_high: rightv];
```

```
            two_intervals : Interval_list
                          := [1: left_interval;
                              2: right_interval];
        in make Result_info[more: two_intervals]
        endlet
    endfun
  % The body of the main procedure is simply responsible for computing the value
  % of the function at the endpoints and then invoking the Integrate routine to
  % complete the work.
    Integrate(low, Evaluate_Function(low), high, Evaluate_Function(high))
endfun  % end of Adaptive_Quadrature
```

APPENDIX B. RECURSIVE AQ

```
function Integrate_2 (left, leftv, right, rightv: real returns real, boolean)
  % If VAL were to allow recursion, this function could replace four functions in the
  % program found in Appendix A: Integrate, Compute_Quads, Build_list, and
  % Build_two_intervals. It is computationally equivalent to the previous version, but
  % only because the first one did not use any information about error estimates for
  % completed intervals. That type of information would not be available in a recursive
  % algorithm. On the other hand, this version is potentially faster because there is no
  % constraint to finish one list of intervals before starting the next one.
    let
      mid:      real := (left + right) * 0.5;
      F_mid:    real := Evaluate_Function(mid);
      midv:     real := if under(F_mid) then 0.0 else F_mid endif;
      old_area: real := (right - left) * (rightv + leftv) * 0.5;
      new_area: real := (right - mid) * (rightv + midv) * 0.5
                      + (mid - left) * (midv + leftv) * 0.5;
      done: boolean := Stop_Condition(old_area, new_area, right - left);
      abort: boolean := is_error(new_area) | is_error(done);
    in
      if    abort then old_area, true
      elseif done then new_area, false
                  else
                    let
                      left_area: real,
                      abt_left : boolean
                               := Integrate_2(left, leftv, mid, midv);
                      rgt_area : real,
                      abt_rgt  : boolean
                               := Integrate_2(mid, midv, right, rightv);
                    in
                      left_area + rgt_area, abt_left | abt_rgt
                    endlet
      endif
    endlet
endfun
```

ACKNOWLEDGMENTS

The author gratefully acknowledges the help of Charles Wetherell, Jack Dennis, and Steve Skedzielewski and the constructive criticisms of the referees during the preparation of this paper.

REFERENCES

(Note. References [5, 6, 20, 24, 27] are not cited in the text.)

1. ACKERMAN, W.B. Axiomatic verification in single assignment languages. Computation Structures Group Memo, Laboratory for Computer Science, M.I.T., Cambridge, Mass., Sept. 1980.
2. ACKERMAN, W.B. Data flow languages. In *Proc. 1979 Nat. Computer Conf.*, New York, N.Y., June 4–7, 1979. AFIPS Conf. Proc., vol. 48. AFIPS Press, Arlington, Va., 1979, pp. 1087–1095.
3. ACKERMAN, W.B., AND DENNIS, J.B. VAL—A value-oriented algorithmic language: Preliminary reference manual. Tech. Rep. TR-218, Computation Structures Group, Laboratory for Computer Science, M.I.T., Cambridge, Mass., June 1979.
4. ADAMS, D.A. A computation model with data flow sequencing. Tech. Rep. CS 117, Computer Science Dep., Stanford Univ., Stanford, Calif., Dec. 1968.
5. ALLAN, S.J., AND OLDEHOEFT, A.E. A flow analysis procedure for the translation of high level languages to a data flow language. In Proc. 1979 Int. Conf. Parallel Processing, O.N. Garcia (Ed.), Aug. 1979, pp. 26–34.
6. ARVIND AND BRYANT, R.E. Parallel computers for partial differential equation simulation. Computation Structures Group Memo 178, Laboratory for Computer Science, M.I.T., Cambridge, Mass., June 1979.
7. ARVIND, GOSTELOW, K.P., AND PLOUFFE, W. The (preliminary) Id report. Tech. Rep. TR 114a, Dep. Information and Computer Science, Univ. Calif. Irvine, Irvine, Calif., May 1978.
8. ASHCROFT, E.A., AND WADGE, W.W. Lucid, a nonprocedural language with iteration. *Commun. ACM 20*, 7 (July 1977), 519–526.
9. BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.
10. BOEKELHEIDE, K. A high-level, graphical, data-driven language. In Proc. Workshop Data Driven Languages and Machines, J.C. Syre (Ed.), Toulouse, France, Feb. 12–13, 1979, sec. XV.
11. BRINCH HANSEN, P. The programming language Concurrent Pascal. *IEEE Trans. Softw. Eng. SE-1*, 2 (June 1975), 199–207.
12. BROCK, J.D. Operational semantics of a data flow language. Tech. Rep. TR-120, Laboratory for Computer Science, M.I.T., Cambridge, Mass., Dec. 1978.
13. BROCK, J.D., AND MONTZ, L.B. Translation and optimization of data flow programs. In Proc. 1979 Int. Conf. Parallel Processing, O.N. Garcia (Ed.), Aug. 1979, pp. 46–54.
14. COMTE, D., DURRIEU, G., GELLY, O., PLAS, A., AND SYRE, J.C. Parallelism, control, and synchronization expression in a single assignment language. *SIGPLAN Notices* (ACM) *13*, 1 (Jan. 1978), 25–33.
15. CORNISH, M., HOGAN, D.W., AND JENSEN, J.C. The Texas Instruments Distributed Data Processor. In Proc. Louisiana Computer Exposition, Lafayette, La., March 1979, pp. 189–193.
16. CROWLEY, W.P., HENDRICKSON, C.P., AND RUDY, T.E. The *Simple* code. Rep. UCID-17715, Lawrence Livermore National Laboratory, Livermore, Calif., Feb. 1978.
17. DAVIS, A. DDNs—A low level programming schema for fully distributed systems. In Proc. Workshop Data Driven Languages and Machines, J.C. Syre (Ed.), Toulouse, France, Feb. 12–13, 1979, sec. XVI.
18. DAVIS, A. The architecture and system method of DDM1: A recursively structured data driven machine. In Proc. 5th Ann. Symp. Computer Architecture, *Comput. Archit. News 6*, 7 (April 1978), 210–215.
19. DE BOOR, C. On writing an automatic integration algorithm. In *Mathematical Software*, J.R. Rice (Ed.). Academic Press, New York, 1971, pp. 201–209.
20. DENNIS, J.B. Data flow supercomputers. *Computer 13*, 11 (Nov. 1980), 48–56.
21. DENNIS, J.B., MISUNAS, D.P., AND LEUNG, C.K.C. A highly parallel processor using a data flow machine language. Computation Structures Group Memo 134, Laboratory for Computer Science, M.I.T., Cambridge, Mass., Jan. 1977.
22. GEHANI, N.H., AND WETHERELL, C.S. Denotational semantics for the data flow language VAL. Internal Memo., Bell Laboratories, Murray Hill, N.J., July 1980.
23. GLAUERT, J. A Single Assignment Language for Data Flow Computing. M.Sc. thesis, Dep. Computer Science, Univ. Manchester, England, Jan. 1978.
24. HOARE, C.A.R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.

25. IVERSON, K. *A Programming Language.* Wiley, New York, 1962.
26. KARP, R.M., AND MILLER, R.E. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM J. Appl. Math. 14,* 6 (Nov. 1966), 1390–1411.
27. KELLER, R.M., LINDSTROM, G., AND PATIL, S.S. Data flow concepts for hardware design. In *Proc. COMPCON Spring 80,* San Francisco, Calif., Feb. 1980. IEEE, Piscataway, N.J., 1980, pp. 105–111.
28. KELLER, R.M., LINDSTROM, G., AND PATIL, S. A loosely-coupled applicative multi-processing system. In *Proc. 1979 Nat. Computer Conf.,* New York, N.Y., June 4–7, 1979. AFIPS Conf. Proc., vol. 48. AFIPS Press, Arlington, Va., 1979, pp. 613–622.
29. KUCK, D.J. Parallel processing of ordinary programs. In *Advances in Computers,* vol. 15. Academic Press, New York, 1976, pp. 119–179.
30. LANDRY, S., AND SHRIVER, B. A data flow simulation research environment. In Proc. Workshop Data Driven Languages and Machines, J.C. Syre (Ed.), Toulouse, France, Feb. 12–13, 1979, sec. X.
31. LAWRIE, D.H., LAYMAN, T., BAER, D., AND RANDAL, J.M. Glypnir—A programming language for Illiac IV. *Commun. ACM 18,* 3 (March 1975), 157–164.
32. LISKOV, B.H., ET AL. CLU reference manual. Memo 161, Computation Structures Group, Laboratory for Computer Science, M.I.T., Cambridge, Mass., July 1978.
33. MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, pt. I. *Commun. ACM 3,* 4 (April 1960), 184–195.
34. MARTIN, J.T., ZWAKENBERG, R.G., AND SOLBECK, S.V. LRLTRAN language used with the CHAT and STAR compilers. Livermore Time-Sharing System, ed. 4, chap. 207, Lawrence Livermore National Laboratory, Livermore, Calif., Dec. 11, 1974.
35. MILLER, R.E., AND COCKE, J. Configurable computers: A new class of general purpose machines. Rep. RC 3897, IBM T. J. Watson Research Center, Yorktown Hts., N.Y., June 1972.
36. MONTZ, L.B. Safety and optimization transformations for data flow programs. Tech. Rep. TR-240, Laboratory for Computer Science, M.I.T., Cambridge, Mass., 1980.
37. OXLEY, D. Private communication on the Distributed Data Processor, March 30, 1979.
38. PETERSON, J.L. Petri nets. *Comput. Surv.* (ACM) *9,* 3 (Sept. 1977), 223–252.
39. PETRI, C.A. Concepts of net theory. In *Proc. Symp. and Summer School on Mathematical Foundations of Computer Science,* High Tatras, Czechoslovakia, Sept. 3–8, 1973. Math. Inst. Slovak Academy of Science, 1973, pp. 137–146.
40. PLAS, A., COMTE, D., GELLY, O., AND SYRE, J.C. LAU system architecture: A parallel data driven processor based on single assignment. In Proc. 1976 Int. Conf. Parallel Processing, P.H. Enslow (Ed.), Aug. 1976, pp. 293–303.
41. RICE, J.R. Parallel algorithms for adaptive quadrature III: Program correctness. *ACM Trans. Math. Softw. 2,* 1 (March 1976), 1–30.
42. RICE, J.R. A metalgorithm for adaptive quadrature. *J. ACM 22,* 1 (Jan. 1975), 61–82.
43. RICE, J.R. Parallel algorithms for adaptive quadrature—Convergence. In *Proc. IFIP Congress 74,* Stockholm. Elsevier North-Holland, New York, 1974, pp. 600–604.
44. RODRIGUEZ, J.E. A graph model for parallel computation. Rep. MAC-TR-64, Project MAC, M.I.T., Cambridge, Mass., Sept. 1969.
45. SEEBER, R.R., AND LINDQUIST, A.B. Associative logic for highly parallel systems. In AFIPS Conf. Proc., vol. 24, 1963, pp. 489–493.
46. SHAPIRO, R.M., SAINT, H., AND PRESBERG, D.L. Representation of algorithms as cyclic partial orderings. Rep. CA-7112-2711, Applied Data Research, Wakefield, Mass., 1971.
47. TESLER, L.G., AND ENEA, H.J. A language design for concurrent processes. In *Proc. 1968 Spring Joint Computer Conf.,* Atlantic City, N.J., April 30–May 2, 1968. AFIPS Conf. Proc., vol. 32. Thompson Book Co., Washington, D.C., 1968, pp. 403–408.
48. TRELEAVEN, P.C., FARRELL, E.P., GHANI, N., JONES, S.B., RANDELL, B., AND SMITH, P.J. The design of highly concurrent computing systems. Tech. Rep. TR126, Computing Laboratory, Univ. Newcastle upon Tyne, England, July 1978.
49. TURNER, D. *SASL Language Manual.* Tech. Rep., Univ. Kent, Canterbury, England, 1979.
50. WATSON, I., AND GURD, J. A prototype data flow computer with token labelling. In *Proc. 1979 Nat. Computer Conf.,* New York, N.Y., June 4–7, 1979. AFIPS Conf. Proc., vol. 48. AFIPS Press, Arlington, Va., 1979, pp. 623–628.

51. WETHERELL, C.S. Error data values in the data flow language VAL. Internal Memo., Bell Laboratories, Murray Hill, N.J.; to appear in *ACM Trans. Program. Lang. Syst. 4,* 2 (April 1982).
52. WETHERELL, C.S. Array processing in Fortran. Rep. UCID-30175, Rev. 1, Lawrence Livermore National Laboratory, Livermore, Calif., Jan. 1980.
53. WIRTH, N. Modula: A language for modular multi-programming. *Softw. Pract. Exper.* 7 (Jan. 1977), 3–35.
54. WOODRUFF, J.P. Scientific application coding in the context of data flow. Rep. UCRL-81922, Lawrence Livermore National Laboratory, Livermore, Calif., Dec. 15, 1978.

Received November 1979; revised January and June 1981; accepted June 1981