A Numeric Error Algebra

W.S. Brown
C.S. Wetherell

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Wetherell recently described an algebra of error values that could be added to the ordinary arithmetic of a programming language. Along with ordinary arithmetic values, error values were included in the set of computational quantities. The error values could participate in all arithmetic operations and return meaningful results. Unfortunately, the definitions of the error values were not precise enough. Using Brown's model of computer arithmetic, we supply precise definitions for the error values, define the fundamental arithmetic operations on the new values, comment on their properties, and discuss briefly how they might be used and implemented. We also compare our model to the error handling features of the proposed IEEE floating point standard.

1. Introduction

Wetherell recently described the use of error values in the data-flow language VAL[1]. VAL has, in addition to the ordinary values of every data type, some extraordinary data values used to report various error conditions. For the arithmetic data types, these error values participate in an algebra that sometimes allows the effect of an error to be undone by later calculations. In VAL, however, the algebra was something of a byproduct; the underlying language demanded that errors not cause sudden halts and hence the error values were invented as a way to avoid interrupts or exceptions.

But an error value algebra may have merit outside the confines of VAL and VAL's special needs. In particular, error values allow an erroneous computation to be handled in one of three graceful ways, depending on the external environment:

1. The error may be reabsorbed because it does not affect later computations. This is the effect of an underflow swallowed in the roundoff error of a later computation.
2. The error may be output from the computation but ignored. For example, full matrices are often computed when only a few elements are desired; if an error value shows up as one of the unneeded elements, no one cares.
3. The error may be tested for explicitly. Because the error is propagated, the test may come at a critical point of the computation and there need not be tests for validity of every unimportant partial result.

These virtues are worth much less, however, if their presence causes the loss of conventional numerical accuracy or portability of numerical software. In the remainder of this paper, we propose a way to build an error algebra compatible with a strong model of numerical computation.

2. The Propagation of Errors

The most important feature of an error data value algebra is that the error values must be full fledged members of the class of data elements. Thus, error values must be allowed to participate as operands in data operations without causing the operations to fail. It may be, of course, that the participation of an error value means that the operation has no reasonable result; in such a case, the operation presumably must construct another (possibly different) error value as result. This is the *propagation* of error values and it is exactly equivalent to saying that the algebra of normal and error values is closed under all operations.

But error value propagation must not be uncontrolled if we are to reap all its benefits without damaging other properties of our numerical types. In particular, a conservation principle must be heeded:

> No operation may generate a result that has more information than is implied by its operands.

So, for example, if we include an overflow value as one of the error values, adding two (positive) overflows should result in an overflow again. However, we must take care when subtracting an ordinary (positive) number from an (positive) overflow; if the ordinary number is large enough (that is, close to the overflow threshold), the result may no longer be overflow. The reason is that the ordinary number might be able to bring the overflow value back below the threshold. In this case, the result of the subtraction is neither overflow nor an ordinary number; it is a positive value of unknown magnitude. (We shall introduce a symbol for this value later.)

A second principle must also be heeded in the design of the algebra:

> Operations on the new data items must not confuse the error analysis provided for the rest of the arithmetic values.

That is, expansion of an arithmetic system with error values must not disrupt the properties and behavior of the system in normal cases. In addition, the error values must behave in an intuitively natural manner wherever possible and it would be desirable to be able to extend theorems smoothly over the new values.

The proposed IEEE floating point standard[4] also provides mechanisms to handle arithmetic errors; obviously the two proposals compete. Although they are not compatible with each other, each can be described in terms of Brown's model of computer arithmetic[2]. We can roughly characterize the differences by saying that our proposal encodes all errors as intervals of values while the IEEE proposal includes denormalized numbers, some less defined error values, and exceptions. In the rest of the paper, we will remark on the differences and relative advantages of the two schemes where appropriate.

3. Definition Of An Error Algebra

For the purposes of description, we will assume that the error values are to be added to a floating point number system that is a

CH1892-9/83/0000/0086$01.00 © 1983 IEEE

simple instantiation of Brown's model. It is clear to us that simply adding the algebra that Wetherell proposed to Brown's model arithmetic would not be wise; the definitions of Wetherell's values are not precise enough.[1] Our new approach, by contrast, permits extension of computation into the domain of error values without fear of anomalies. We will define the new values directly in terms of the parameters of Brown's arithmetic.

3.1 Review Of The Brown Model

The relevant portions of the model can be described by a few lines from Brown's paper[2].

> The model includes a few basic parameters and a few derived parameters for each floating-point system that is supported by the host computer. If a computer supports two or more such systems (*e.g.*, single and double precision), then each has its own parameters. The basic parameters, all integers, are
>
> (1) the *base*, b;
>
> (2) the *precision*, p,
>
> (3) the *minimum exponent*, $e_{\min}$;
>
> (4) the *maximum exponent*, $e_{\max}$.
>
> These define a system of *model numbers* consisting of zero and all numbers of the form
>
> $$x = fb^e \tag{1}$$
>
> where
>
> $$f = \pm(f_1 b^{-1} + \cdots + f_p b^{-p}),$$
> $$f_1 = 1, \cdots, b-1$$
> $$f_2, \cdots, f_p = 0, \cdots, b-1 \tag{2}$$
>
> and
>
> $$e_{\min} \leqslant e \leqslant e_{\max}. \tag{3}$$
>
> The parameters must be chosen so that these model numbers are exactly representable in the machine...
>
> . . .
>
> We do not assume that the computer actually uses a normalized sign-magnitude representation for floating-point numbers. In fact, the details of the hardware representation are of no concern to us. What we do require is simply that the model numbers be possible values for program variables, and that arithmetic operations be at least accurate enough to satisfy the axioms [*not presented here—Authors*].
>
> . . .
>
> Since the model numbers with a given exponent e are equally spaced on an absolute scale, the relative spacing decreases as the magnitude of the fraction-part f increases. For error analysis, the maximum relative spacing
>
> $$\epsilon = b^{1-p} \tag{4}$$
>
> is of critical importance. Also of interest · · · are the smallest positive model number
>
> $$\sigma = b^{e_{\min}-1} \tag{5}$$
>
> and the largest model number
>
> $$\lambda = b^{e_{\max}}(1-b^{-p}). \tag{6}$$

1. It is also the case that VAL's algebra has some errors and omissions in the propagation laws. These were known to Wetherell, but the warts were left as is so that the paper accurately reflected the VAL language at the time of publication.

The four parameters of the model are linked by several inequalities of which two are of interest in this paper:

$$e_{\min} \leqslant 2 - 2p \quad \text{and} \quad e_{\max} \geqslant 2p - 1$$

These inequalities force a range of at least $4p-3$ in the exponent. Thus a value near the middle of the range is much less than one *unit in the last place* (*ulp*) of a value near the large end of the range. In addition, we use one more parameter for easy description of the error values.

λ^* is one ulp more than λ and equals $b^{e_{\max}}$.

Although $e_{\min}$ and $e_{\max}$ can be chosen arbitrarily in Brown's model, we will require here that they meet the condition

$$\sigma\lambda^* = 1$$

This choice makes the definition of the error values much easier and simplifies the resulting algebra. If this condition is violated, the error values representing overflow and underflow are no longer exact reciprocals; complications then grow quickly in the error algebra.

Brown's model permits additional machine numbers that are either *extra-precise* or *out-of-range*. A machine that conforms to the proposed IEEE floating-point standard (for this paper, an *IEEE machine*) conforms *a fortiori* to Brown's model. However, on such a machine, σ is the smallest positive *normalized* number and the denormalized numbers are out-of-range. IEEE extended precision number may conveniently be viewed as extra-precise, or if the user controls when the extended precision numbers are used (by a language data type, say), the extended precision system can be modeled separately from the standard system.

3.2 New Arithmetic Error Values

Each of our new numbers represents an interval of the real line with the projective infinity added. The first two values are their own negatives.

- ∞ is the *projective infinity*. If we need to make the distinction, the ends of the ordinary real number line will be marked as $-\infty$ and $+\infty$.
- Ω is the entire set of numbers $(-\infty, +\infty) \cup \{\infty\}$. It represents the state of no knowledge whatsoever.

We also add three signed numbers.

- δ is the positive open interval from zero to σ, *i.e.*, $(0, \sigma)$; this may be thought of as the result of positive underflow.[2]
- ρ is the open interval $(\lambda^*, +\infty)$; this is the exact reciprocal of δ and may be thought of (to within rounding) as positive overflow.
- ω is the open interval $(0, +\infty)$ containing all the positive numbers; this is the positive *unknown* value that is created when all but sign information is lost.

These error values all have corresponding negative values $-\delta$, $-\rho$, and $-\omega$. If we let $\Re$ stand for the representable real numbers in the interval $[\sigma, \lambda]$, then the representable real line in our algebra is

$$\infty, -\rho, -\Re, -\delta, 0, \delta, \Re, \rho, \infty$$

Notice that projective infinity ∞ stands at both ends.

The proposed IEEE floating-point standard provides an infinite value corresponding to our ∞ when the *projective* mode is selected. If the *affine* mode is selected instead, the IEEE standard has $-\infty$ and $+\infty$ at the ends of the number line. It also provides a large number of NaN's (*NaN* ≡ *N*ot *a N*umber) which correspond roughly to our Ω. The IEEE standard does not provide intervals like δ, ρ, and ω, but it does provide *denormalized numbers* that are equally spaced throughout our interval δ.

Unary Negation	
x	$-x$
0	0
δ	$-\delta$
R	$-R$
ρ	$-\rho$
ω	$-\omega$
∞	∞
Ω	Ω

TABLE 1. Definition Of Unary Negation

3.3 Operations On The New Error Values

The arithmetic algebra for the expanded system can be stated in a few tables. The structure of the algebra relies on the interpretation of the error values as intervals; from this observation, most operator results can be derived by inspection from the represented intervals. In all the tables, R stands for a value from the representable positive real numbers $\Re$. The first rule, covering unary negation, is shown in Table 1. It is constructed directly from the definitions of the error values themselves.

Absolute Value	
x	$\lvert x \rvert$
$-\omega$	ω
$-\rho$	ρ
$-R$	R
$-\delta$	δ
0	0
δ	δ
R	R
ρ	ρ
ω	ω
∞	∞
Ω	Ω

TABLE 2. Definition Of The Unary *Abs* Operator

Similarly, the definition for the absolute value operator can be found in Table 2. One might expect that $|\Omega|$ would be ω because all the negative numbers that are members of Ω have their signs changed to positive by the absolute value operation. However, ω does not include zero or $+\infty$ and Ω does; this keeps $|\Omega|$ from being contained in ω. If ω were defined as $[0, +\infty) \bigcup \{\infty\}$, then $|\Omega|$ would be ω, but some more important multiplication results would be less natural.

The revised definition of addition is fairly simple. First, Table 3 defines addition for all combinations of positive and signless quantities. Table 4 likewise defines subtraction for all differences of positive quantities. Using these two tables, the definition of negation from Table 1, the ordinary laws of signs, and particularly the equation

$$x - y = x + (-y)$$

all sums and differences can be computed.

Several points need to be made about addition.

- Addition is still commutative.
- Zero is still the additive identity.
- Projective infinity ∞ is a kind of additive eraser; the unknown value Ω is a slightly less powerful eraser.

2. One of the problems with Wetherell's earlier paper was that it identified the error values specifically with underflow, overflow, and the like. These terms are too vague for analytical use, although they may be of some use in explanation. Care should be taken when using these intuitive but vague terms.

Addition							
+	0	δ	R	ρ	ω	∞	Ω
0	0						
δ	δ	ω					
R	R	(7)	(8)				
ρ	ρ	ρ	ρ	ρ			
ω	ω	ω	ω	ρ	ω		
∞	∞	∞	∞	∞	∞	∞	
Ω	Ω	Ω	Ω	Ω	Ω	∞	Ω

TABLE 3. Definition Of Addition
See text for equations (7) and (8).

- The entries for the sum or difference of a representable number R and δ are not immediately obvious. However, remember that δ is a very small value, so small that it may be lost in the rounding error when added to R. On the other hand, if R is very near σ, then δ could change it by an unknown amount and it is impossible to return an accurate approximation to the sum. This gives rise to the rule

$$R \pm \delta = \begin{cases} R & \text{if } R \geqslant \sigma/\epsilon \\ \omega & \text{otherwise} \end{cases} \tag{7}$$

In the proposed IEEE standard, the availability of denormalized numbers throughout the underflow interval δ allows an addition rule that is much more attractive than (7). If R is normalized and d is denormalized, then

$$R \pm d = \begin{cases} R & \text{if } R \geqslant \sigma/\epsilon \\ R' & \text{otherwise} \end{cases} \tag{7a}$$

where R' is within an ulp of the exact value of $R \pm d$. Furthermore, the sum or difference of two denormalized numbers is always at least approximately correct on an IEEE machine, whereas in our model, $\delta+\delta$ loses all information except sign and $\delta-\delta$ loses all information.

- The sum or difference of two representable real numbers may underflow, overflow, or be representable depending on the magnitude of the result. The magnitude of the result is given by the rule in the equation and the sign follows the ordinary rules:

$$R \pm R' = \begin{cases} \delta & \text{if the difference underflows} \\ \rho & \text{if the sum overflows} \\ R'' & \text{otherwise} \end{cases} \tag{8}$$

On an IEEE machine, this rule is replaced by

$$R \pm R' = \begin{cases} d & \text{if the difference underflows} \\ \infty & \text{if the sum overflows} \\ R'' & \text{otherwise} \end{cases} \tag{8a}$$

where d is a denormalized number within an ulp of the exact value of $R \pm R'$. Note that the identification of overflow with ∞ has no mathematical foundation.

- The value of $\delta-\delta$ is, unfortunately, Ω because the magnitudes of the quantities going into the computation are unknown. The mathematical result could be anywhere in the open interval $(-\sigma, \sigma)$; we have no value to represent this interval.
- The difference $\rho-R$ in Table 4 is analogous to the addition entry for $R+\delta$:

$$\rho - R = \begin{cases} \rho & \text{if } R \leqslant \epsilon\lambda^* \\ \omega & \text{otherwise} \end{cases} \tag{9}$$

On an IEEE machine, the analogous rule is $\infty - R = \infty$. Hence, if $R+R'$ overflows, we have $(R+R')-R=\infty$, which illustrates the vagueness of the IEEE ∞. In our proposal, the value of this expression would be ω, which accurately reveals the machine's inability to do the desired computation, while retaining some useful information.

- The value of $\rho-\delta$ is ρ because of the inequalities governing e_{min} and e_{max} mentioned above.

Subtraction							
-	0	δ	R	ρ	ω	∞	Ω
0	0						
δ	δ	Ω					
R	R	(7)	(8)				
ρ	ρ	ρ	(9)	Ω			
ω	ω	Ω	Ω	Ω	Ω		
∞	∞	∞	∞	∞	∞	∞	
Ω	Ω	Ω	Ω	Ω	Ω	∞	Ω

TABLE 4. Definition Of Subtraction
See text for equations (7), (8), and (9).

The definition for multiplication is similar and can be found in Table 5. Division can be read off the multiplication table by use of the equation

$$x/y = x \times (1/y)$$

where Table 6 gives the reciprocals of all the algebraic values. Some comments about multiplication and division also need to be made.

- The definition of $R \times \delta$ depends on the exact value of R; the turnover point is one.

$$R \times \delta = \begin{cases} \delta & \text{if } R \leqslant 1 \\ \omega & \text{otherwise} \end{cases} \qquad (10)$$

On an IEEE machine, if R is normalized and d is denormalized, the corresponding rule is

$$R \times d = \begin{cases} d' & \text{if } R \leqslant 2 \\ NaN & \text{otherwise} \end{cases} \qquad (10a)$$

where d' is a denormalized number (or possibly normalized if $R>1$) within an ulp of the exact value of $R \times d$, while the *NaN* is one that accompanies an *invalid operation* exception.

- The product of two representable real numbers can result in δ, a representable number, or ρ, depending on whether the product underflows, is normal, or overflows. The rule is

$$R \times R' = \begin{cases} \delta & \text{if the product underflows} \\ \rho & \text{if the product overflows} \\ R'' & \text{otherwise} \end{cases} \qquad (11)$$

On an IEEE machine, the analogous rule is

$$R \times R' = \begin{cases} 0 & \text{if the product underflows below } d_{min}/2 \\ d & \text{if the product underflows a larger value} \\ \infty & \text{if the product overflows} \\ R'' & \text{otherwise} \end{cases} \qquad (11a)$$

where d_{min} is the smallest positive denormalized number and d is a denormalized approximation to the exact value of $R \times R'$ (with at least one significant bit). Note that $(\sigma \times \sigma)/\sigma = 0$ and $(\lambda \times \lambda)/\lambda = \infty$. In our proposal, both of these expressions would have the value ω, which accurately reveals the machine's inability to perform the requested computation.

- Similarly, $\rho \times R$ depends on the value of R going the other direction.

$$\rho \times R = \begin{cases} \rho & \text{if } R \geqslant 1 \\ \omega & \text{otherwise} \end{cases} \qquad (12)$$

On an IEEE machine, the analogous rule is $\infty \times R = \infty$. Hence if R'/R overflows, we have $(R'/R) \times R = \infty$, which again illustrates the vagueness of the IEEE infinity. In our proposal, the value of this expression would be ω, which accurately reveals the machine's inability to perform the desired computation.

- The representable value σ has a surprising reciprocal—ρ. Unfortunately, its natural reciprocal, λ^*, is not representable without special effort. This does not seem a major gap. However, a representation for λ^* could be added specially to the number system to close the gap; the implementation cost is likely to be higher than any gain.

Multiplication							
$\times$	0	δ	R	ρ	ω	∞	Ω
0	0						
δ	0	δ					
R	0	(10)	(11)				
ρ	0	ω	(12)	ρ			
ω	0	ω	ω	ω	ω		
∞	Ω	∞	∞	∞	∞	∞	
Ω	Ω	Ω	Ω	Ω	Ω	Ω	Ω

TABLE 5. Definition Of Multiplication
See text for equations (10), (11), and (12).

4. Other Error Values

Once error values have been added to the real numbers, they must also be added to other data types as well. For example, what is the result of converting the real value ρ to an integer value? Surely it should not be an overflow signal; why rid the algebra of exceptional conditions in one area only to retain them in another? The integers will require their own ρ, ω, ∞, and Ω values to match those of the reals (but not δ, of course).

Similarly, Boolean operations need to be extended. What, for example, is the result of the comparison

$$\omega < 7$$

The answer can be neither true nor false because ω represents an interval of values that includes 7.[3] Rather than simply failing on such an unordered relation, we choose to add a Boolean value Ω that stands for the set {*true*, *false*}, in other words, for the state of no knowledge about the result of a comparison (or other Boolean operation). Once this value is added, the Boolean operations are easy to extend.

Other basic data types may also need new values. The values may be as simple as an undefined value (as Boolean Ω) or may be much more complicated, depending on the use of the data type. The Ada ®[4] language's fixed point data type[3], for example, might have use for nearly as complete a set of error values as the real error values proposed here. Compound data types may also be in need of extension. For example, what is the effect of indexing an array with the integer value Ω? Should the program simply halt with an exceptional condition—perhaps with the message "Array index out of bounds"? In VAL, such an indexing operation generated the new error value *Out-of-bounds-reference* which in turn propagated to Ω (or its equivalent) in all further operations. Depending on the use of the error algebra, it may be desirable to include these extra compound

3. The relation $-\omega < 7$ is true, however, because every element of the interval comprising $-\omega$ is less than 7.

4. Ada is a Registered Trademark of the U.S. Government—Ada Joint Program Office.

Reciprocal	
x	$1/x$
0	∞
δ	ρ
R	R
ρ	δ
ω	ω
∞	0
Ω	Ω

TABLE 6. Definition Of The Reciprocal Operation

data values. VAL includes a rationale and proposal for such values

Finally, it may be necessary to have a completely undefined error value for each data type representing the result of a meaningless computation that goes outside the set. Real square root, applied to a negative argument, produces such a value. If all operations result in a reasonable value, then the need for automatically raised exceptions is eliminated. A considerable software and hardware simplification may result.

4.1 Testing Predicates

In VAL, some error values always propagate to the most undefined error value in every operation; in particular, when they appear in relationals, they always generate the Boolean Ω. Thus VAL includes explicit predicates to test for each of the possible error values. Such a predicate might be written

$$is\_\Omega(x)$$

This predicate would return true if its argument had the value Ω and false otherwise. Under VAL's propagation rules, such a predicate can not be built from simple relations. Even if a language did not have such a stringent propagation regimen, however, error value predicates might be worth inclusion.

4.2 More And More Error Values

One of the rules that may seem strange is that $\delta+\delta$ sums to ω in spite of the fact that the sum is known to be very small. Why not add another error value, say Δ, that covers the interval $(0, \sigma/\epsilon)$? Then the smallness of the problem sum is captured. However, this leads to an indefinite regress because we need to have a rule for the sum $\delta+\Delta$, yet another small (but not quite so small) number. And if we add a special symbol for this sum, what about the next? Instead, we provide one graceful chance to catch an error and then subside into ω.

5. Examples

A typical iterative scheme, such as Newton's method, forms a new value by adding some computed term to an old value. The process terminates when the increment becomes smaller than some limit. Using the error values, if the added term happens to underflow, it will automatically terminate the iteration because δ is less than any representable positive quantity. On the other hand, if the term overflows or becomes anomalous in any other way, the program will report the erroneous condition as an error value and the condition will not go unnoticed. Thus, naive programs are likely either to work correctly or to report the presence of a problem even if they do not correct it.

Quite often, of course, the additive term takes the form of a quotient. The quotient can sometimes be arranged so that the numerator is never greater than one and the denominator is never less than one. Then either of two numeric errors might occur: first, the numerator may underflow; second, the denominator may overflow. In either case, convergence has been achieved, but a conventional machine would probably require special trap coding to recognize the convergence. In the error algebra, the possible converging cases are

$$q = \begin{cases} R/R', & R \ll R' \\ \delta/R', & R' > 1 \\ R/\rho, & R < 1 \\ \delta/\rho \end{cases}$$

In each case, the quotient q can be tested by the relation

$$|q| < \epsilon$$

because q will be either a representable real number or δ. If the program can be arranged so that one of these four conditions is guaranteed for the quotient, then no checks for computation errors need be made explicitly; all errors will be swallowed by the division operation.

As a second example, consider the summation of a series. If all the elements of the series are representable numbers, there still may be an underflow or overflow during the operation. However, if some partial sum underflows, the next term is quite likely to be big enough to absorb the δ so generated and the summation can go to completion with a reasonable result. If the next term is not big enough, then the result will be ω or Ω and will at least signal that a problem occurred. Similarly, if an overflow occurs, ρ will be generated and signal a problem. Notice that this naive approach is reasonable even if there are error values (particularly δ) in the series.

On the other hand, a result of ω need not mean that the series can not be summed. It may need to be reordered or summed more carefully. If the series happens to have all positive terms, a δ can only arise because there are δ values in the series itself. Instead of adding them directly, count them, and when the representable values have been summed, see if the net effect of all the δ values could make the entire sum uncertain. If so, then produce ω as a result; otherwise, ignore the contribution of the δ values. This algorithm, of course, is slightly more complicated than a naive summation; it is, however, just the first step towards scaled summation, necessary in general to preserve precision.

6. Implementation

The implementation of error values in hardware is not too difficult. One or more special exponent values will probably be reserved for the error values. The configuration of the significand, since it carries no numeric information, can be used to code the particular value selected. Most operations involving error values can be trapped very early in the arithmetic processor and pushed down special (and simple) paths; for example, if an Ω appears in a multiplication, the result is always an Ω.

In a few cases, the result of a computation is dependent on conditional tests applied either to the input values or to the output values However, whenever the result depends on input conditions (for example, $R \times \delta$), the test is always a simple one on the exponent value of the ordinary value (here, is the exponent less than one?). Similarly, conditional output only arises in situations where ordinary units would signal overflow, underflow, or another error condition; replacing the signal with a value is straightforward.

Because the significand of an error value is mostly unused, it can be utilized as storage for an error trace. If every operation that propagates an error value carries the old trace information forward, it may be fairly simple to pin down the spot where a bad computation first appeared. Since a major part of debugging is finding the genesis of the erroneous output, error traces are likely to be of significant help. The details of an error trace must depend on the particular hardware implementation.

7. Conclusions

The inclusion of error values in an arithmetic algebra is a simple extension. It does not unduly complicate numerical analysis, hardware implementation, or software. It does, however, pay considerable benefits in error handling; the programmer may choose to handle errors in a variety of ways and at a variety of times. Some algorithms become simpler because potential errors are gracefully swallowed or reported only when appropriate. Error values also makes possible programming languages with fewer side effects, theoretically and practically a desirable end. Naturally, good language design is necessary to incorporate error values cleanly; in particular, it may be difficult to add them on top of current languages in a tidy way. A complete review of the advantages of an error algebra can be found in Wetherell's paper[1].

We do note that the error algebra allows graceful handling of the first error in a computation; normally such an error will generate either δ or ρ. The second error, however, will usually result in ω or Ω, losing most of the information available in the computation. Some other arithmetic models attempt to retain more information when a succession of errors occurs[4]; however, we feel they do this at a some loss in the precision with which the values can be characterized and in the sharpness with which portability arguments may be made. In particular, our case by case comparison with the proposed IEEE standard suggests that our error values are likely to be more informative except in the one case of addition of very small values.

Bibliography

[1] C.S. Wetherell. Error Data Values in the Data-Flow Language VAL. *TOPLAS*, **4, 2**, pp. 226-238. April, 1982.

[2] W.S. Brown. A Simple but Realistic Model of Floating-Point Computation. *TOMS*, **7, 4**, pp. 445-480. December, 1981.

[3] J. Ichbiah *et al. Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815 A. United States Department of Defense. January, 1983.

[4] IEEE Task P754. A Proposed Standard For Binary Floating Point Arithmetic, Draft 8.0 *IEEE Computer*. 1981.