A Formal Approach to Rounding

Geoff Barrett
Oxford University Programming Research Group

Abstract

This paper presents a formal description of rounding, as specified in the IEEE Standard, and an algorithm to perform the task along with its proof of correctness.

Introduction

The main aim of a standard is that "conforming" implementations should behave in the manner specified – it is, therefore, necessary that they should be proved to do so. It has long been argued that natural language specifications can be ambiguous or misleading and, furthermore, that there is no formal link between specification and program. This paper sets out to formalise the definition of rounding given in [IEEE] and to present an algorithm, with proof of correctness, which performs this task.

The notations used in this paper are Z (see [Abrial,Hayes,Z]) and occam (see [inmos]). The meaning of each new piece of notation is explained in a footnote before an example of its use.

Using a formal specification language bridges the gap between natural language specification and implementation. Natural language specifications have two disadvantages: they can be ambiguous; and it is difficult to show their consistency. The first problem is considered to be an important source of software and hardware errors and is eliminated completely by a formal specification. Further, it is important to show that a specification is consistent (i.e. has an implementation) for obvious reasons.

Of course, it could be argued that an implementation of a solution provides a precise specification of a problem. While this is true, no one likes to read other peoples' code and the structure of a program is designed to be read by machine and not by humans. Moreover, any flexibility in the approach to the problem is hampered by the need to make concrete design decisions. Specification languages are structured in such a way that they can reflect the structure of a problem or a natural language description or even of a program. But, above all, they can also be *non-algorithmic*. This means that one can formalise what one has to do without detailing how it is to be done.

A formal development divides the task of implementing a specification into four well-defined steps. The first is to write a formal specification using mathematics. In the second, this specification is decomposed into smaller specifications which can be recombined in such a way that it can be shown formally that the decomposition is valid. Third, programs are written to satisfy the decomposed specifications. And, lastly, program transformations can be applied to make the program more efficient or, possibly, to adapt it for implementation on particular hardware configurations.

The example presented here is part of a large body of work which has been undertaken to formally develop a complete floating-point system. A formalisation of most of the IEEE Standard and proofs of the non-exceptional arithmetic routines can be found in [Barrett]. This work has been taken further by David Shepherd to transform the resulting routines into a software model of the inmos T800 processor, and so specify its functions. Thus, the development process has been carried through from formal specification to silicon implementation.

The structure of this paper is as follows. Section 1 presents a formal definition of floating-point numbers and how they are used to approximate real numbers. Section 2 states some theorems and decomposes the definition to give a specification of a rounding module. Section 3 presents the algorithm with its proof and a brief description of the techniques used in the proof.

1 Specification.

This section presents a formal description of floating-point numbers and how they are used to approximate real numbers. The description serves as a specification for a rounding procedure.

First, floating-point numbers and their representation are described. Each number has a format. This consists of the exponent and fraction widths and other useful constants associated with these – the minimum and maximum exponent and the bias: [1]

Format

$$\begin{array}{|l} wordlength, expwidth, fracwidth : \mathbb{N} \\ EMin, EMax, Bias : \mathbb{N} \\ \hline wordlength = expwidth + fracwidth + 1 \\ EMin = 0 \\ EMax = 2^{expwidth} - 1 \\ Bias = 2^{expwidth-1} - 1 \end{array}$$

[1] The variable names which are used are declared in a signature (the upper part of the box) and any constraints on these are described by the predicates in the lower part.

CH2419-0/87/0000/0247$01.00 © 1987 IEEE

Four formats are specified – the exponent width and wordlength are constrained to have particular values:

$$\begin{array}{lll} Single & \widehat{=} Format \mid & expwidth = 8 \wedge \\ & & wordlength = 32 \\ Double & \widehat{=} Format \mid & expwidth = 11 \wedge \\ & & wordlength = 64 \\ SingleExtended & \widehat{=} Format \mid & expwidth \geq 11 \wedge \\ & & wordlength \geq 43 \\ DoubleExtended & \widehat{=} Format \mid & expwidth \geq 15 \wedge \\ & & wordlength \geq 79 \end{array}$$

Once the format is known, the sign, exponent and fraction can be extracted from the integer in which they are stored: [2] (See *Fields*, figure 1)

Some of the elements of *Fields* are considered to be error codes, or non-numbers. These will be denoted by *NaNF*:

$$NaNF \widehat{=} Fields \mid frac \neq 0 \wedge exp = EMax$$

Now, there are enough definitions to give a definition of the value. This is only specified in single or double formats when the number is not a non-number: ("infinite" numbers are given a value to facilitate the definition of rounding) (See *FP*, figure 2)

To facilitate further descriptions, *FP* is partitioned into five classes depending on how its value is calculated from its fields: (non-numbers; infinite, normal, denormal numbers; and zero)

$$\begin{array}{lll} NaN & \widehat{=} FP \mid frac \neq 0 \wedge exp = EMax \\ Inf & \widehat{=} FP \mid frac = 0 \wedge exp = EMax \\ Norm & \widehat{=} FP \mid EMin \leq exp < EMax \\ Denorm & \widehat{=} FP \mid frac \neq 0 \wedge exp = EMin \\ Zero & \widehat{=} FP \mid frac = 0 \wedge exp = EMin \end{array}$$

$$Finite \widehat{=} Norm \vee Denorm \vee Zero$$ [3]

The essential ingredients of rounding are as follows:

- the number to be approximated;
- a set of values in which the approximation must be;
- a rounding mode;
- a set of preferred values in case two approximations are equally good.

Because the number to be approximated may be outside the range of the approximating values, two values, *MaxValue* and *MinValue*, are introduced which are analogous to $+\infty$ and $-\infty$. The set of *Preferred* values is restricted to ensure that when two approximations are equally good, at least one of them is preferred. To ensure that rounding to zero is consistent, 0 must be in the approximating values.

$$Mode ::= ToNearest \mid ToZero \mid ToNegInf \mid ToPosInf$$

(See *Round_Signature*, figure 3)

The following schemas describe the closest approximations from above and below. If, e.g., the number is smaller than *MinValue*, then the approximation from below is *MinValue*: (See *Above*, figure 4) (See *Below*, figure 5)

Finally, we are in the position to define rounding in its various different modes. Rounding toward zero gives the approximation with the least modulus:

RoundToZero

Round_Signature

$mode = ToZero$
$(r \geq 0 \wedge Below$
$\vee$
$r \leq 0 \wedge Above)$

Rounding to positive or negative infinity returns the approximation which is respectively greater or less than the given number:

RoundToPosInf

Round_Signature

$mode = ToPosInf$
$Above$

RoundToNegInf

Round_Signature

$mode = ToNegInf$
$Below$

When rounding to nearest, the closest approximation is returned, but if both are equally good, a member of the set *Preferred* is returned: [4] (See *RoundToNearest*, figure 6) These specifications can be disjoined to give the full specification as follows.

$$Round \widehat{=} RoundToNearest \vee RoundToZero \vee RoundToPosInf \vee RoundToNegInf$$

[2] This form is equivalent to declaring the variables of *Format* in the signature and conjoining the constraints with the new constraint.

[3] Logical operators between schemas have the effect of merging the signatures and performing the logical operation between the predicates.

[4] Decorating the name of a schema with, e.g., $_1$, $'$ has the effect of decorating the names of the variables in the signature of that schema throughout.

Fields

$$
\begin{array}{l}
\textit{Format} \\
\textit{sign} \quad : 0..1 \\
\textit{exp}, \textit{frac}, \textit{nat} : \mathbb{N} \\
\hline
\textit{nat} = \textit{sign} \times 2^{\textit{wordlength}-1} + \textit{exp} \times 2^{\textit{fracwidth}} + \textit{frac} \\
\textit{exp} < 2^{\textit{expwidth}} \\
\textit{frac} < 2^{\textit{fracwidth}}
\end{array}
$$

Figure 1: *Fields*

FP

$$
\begin{array}{l}
\textit{Fields}; \textit{value} : \mathbb{R} \\
\hline
(\textit{Single} \vee \textit{Double}) \wedge \neg \textit{NaNF} \Rightarrow \\
\quad \textit{exp} = \textit{EMin} \wedge \textit{value} = (-1)^{\textit{sign}} \times 2^{\textit{exp}-\textit{Bias}} \times 2 \times \textit{frac}_0 \\
\quad \vee \\
\quad \textit{exp} \neq \textit{EMin} \wedge \textit{value} = (-1)^{\textit{sign}} \times 2^{\textit{exp}-\textit{Bias}} \times (1 + \textit{frac}_0) \\
\quad \textbf{where} \quad \textit{frac}_0 = 2^{-\textit{fracwidth}} \times \textit{frac}
\end{array}
$$

Figure 2: *FP*

Round_Signature

$$
\begin{array}{l}
r : \mathbb{R}; \textit{mode} : \textit{Modes} \\
\textit{ApproxValues}, \textit{Preferred} : \mathbb{P}\mathbb{R} \\
\textit{MinValue}, \textit{MaxValue} : \mathbb{R} \\
\textit{value}' : \mathbb{R} \\
\hline
\textit{Preferred} \cup \{\textit{value}'\} \subseteq \textit{ApproxValues} \cup \{\textit{MinValue}, \textit{MaxValue}\} \\
0 \in \textit{ApproxValues} \\
\forall \textit{value}_1, \textit{value}_2 : \textit{ApproxValues} \cup \{\textit{MinValue}, \textit{MaxValue}\} \mid \textit{value}_1 > \textit{value}_2 \bullet \\
\quad \exists p : \textit{Preferred} \bullet \textit{value}_1 \geq p \geq \textit{value}_2 \\
\forall \textit{value} : \textit{ApproxValues} \bullet \textit{MinValue} \leq \textit{value} \leq \textit{MaxValue}
\end{array}
$$

Figure 3: *Round_Signature*

Above

$$
\begin{array}{l}
\textit{Round\_Signature} \\
\hline
r > \textit{MaxValue} \Rightarrow \textit{value}' = \textit{MaxValue} \\
r \leq \textit{MaxValue} \Rightarrow \textit{value}' \geq r \\
\qquad \forall \textit{value} : \textit{ApproxValues} \cup \{\textit{MaxValue}\} \mid \textit{value} \geq r \bullet \\
\qquad\quad \textit{value} \geq \textit{value}'
\end{array}
$$

Figure 4: *Above*

Below

Round_Signature

$$
\begin{array}{l}
r < MinValue \Rightarrow value' = MinValue \\
r \geq MinValue \Rightarrow value' \leq r \\
\qquad \forall value : ApproxValues \cup \{MinValue\} \mid value \leq r \bullet \\
\qquad\quad value \leq value'
\end{array}
$$

Figure 5: *Below*

RoundToNearest

Round_Signature

$$
\begin{array}{l}
mode = ToNearest \\
\exists Above_1; Below_2 \mid r_1 = r = r_2 \bullet \\
\quad value_1 - r < r - value_2 \quad \wedge \quad Above \\
\qquad\qquad \vee \\
\quad value_1 - r > r - value_2 \quad \wedge \quad Below \\
\qquad\qquad \vee \\
\quad value_1 - r = r - value_2 \quad \wedge \\
\qquad (value_1 = value_2 \quad \wedge \quad Above \wedge Below \\
\qquad\qquad \vee \\
\qquad\quad value_1 \neq value_2 \quad \wedge \quad value' \in Preferred \wedge (Above \vee Below))
\end{array}
$$

Figure 6: *RoundToNearest*

FP_Round1

Round $\wedge$ *FP'*

$$
\begin{array}{lcl}
ApproxValues & = & \{Finite \mid Format = Format' \bullet value\} \\
Preferred & = & \{Finite \mid Format = Format' \wedge frac \bmod 2 = 0 \bullet value\} \\
MinValue & \in & \{Inf \mid Format = Format' \wedge sign = 1 \bullet value\} \\
MaxValue & \in & \{Inf \mid Format = Format' \wedge sign = 0 \bullet value\}
\end{array}
$$

Figure 7: *FP_Round1*

So far, the specification is suitable for describing rounding into any format – be it integer or floating-point. To adapt *Round* specifically for floating-point format, all that is necessary is to fill in the definitions of *ApproxValues*, *Preferred*, *MinValue* and *MaxValue*. This inevitably involves the format of the destination, so FP' must be conjoined with *Round*. Once the definitions are filled in, they are no longer needed outside the specification and can be hidden (by existential quantification). It is not difficult to show that the definition of *Preferred* is consistent with the constraint in *Round_Signature*, but this will be left until section 2 where a result is proved which makes it even simpler. It is also simple to verify that 0 is an element of *ApproxValues* and that *MinValue* and *MaxValue* satisfy the constraint of *Round_Signature*. (See *FP_Round1*, figure 7)

$$FP\_Round2 \mathrel{\widehat=} FP\_Round1 \setminus \{ApproxValues, Preferred, MinValue, MaxValue\}$$

The resulting error-conditions have not yet been specified. The conditions resulting in overflow and underflow exceptions are specifically related to a floating-point format and can be described as follows:

$$\begin{array}{lcl} Errors & ::= & inexact \mid overflow \mid underflow \\ Error\_Signature & \mathrel{\widehat=} & r : \mathbb{R};\ errors' : \mathbb{P}\, Errors;\ FP' \end{array}$$

(See *Error_Spec*, figure 8) (The two alternative conditions under which *underflow* is included in the set *errors'* mean that there is a choice about which condition to implement.)

Finally, the whole specification is:

$$FP\_Round \mathrel{\widehat=} FP\_Round2 \wedge Error\_Spec$$

2 Towards an Algorithm.

The aim of this section is to specify the relation between **R** and its representations in the program.

First, notice the simple result that the order on the absolute value of a number is the same as the usual order on the less significant bits of its representation as a word:

$$\begin{array}{l} FP; FP' \mid Format = Format' \wedge \neg(NaN \vee NaN') \\ \vdash \quad \text{abs } value \leq \text{abs } value' \\ \qquad\qquad \Leftrightarrow \\ nat \bmod 2^{wordlength-1} \leq nat' \bmod 2^{wordlength-1} \end{array}$$

This can be used to see that the number of least modulus with modulus greater than a given finite number is obtained by incrementing its representation as a word: (See *Succ*, figure 9)

$$FP; FP_0 \mid Finite \wedge nat_0 = nat + 1 \vdash Succ$$

From this result, the consistency of *Preferred* in section 1 can be deduced.

In turn, this means that if the approximation of less modulus is known only enough extra information to determine the four predicates in *RoundToNearest* is needed to return the correct value. This is, of course, the familiar *guard* and *sticky* bits defined below:

Bounds

$$\begin{array}{l} Succ; guard, sticky : 0..1;\ r : \mathbb{R} \\ \hline r > 0 \Rightarrow sign = 0 \wedge Below[value/value'] \\ r = 0 \Rightarrow Zero \\ r < 0 \Rightarrow sign = 1 \wedge Above[value/value'] \\ guard = 0 \Leftarrow r - value < value_0 - r \\ sticky = 0 \Leftarrow r - value = value_0 - r \vee r = value \end{array}$$

$$\begin{array}{l} Bounds \vdash \\ \exists Above_1; Below_1 \mid r_1 = r = r_2 \bullet \\ value_1 - r < r - value_2 \Leftrightarrow guard = 0 \wedge sticky = 1 \\ value_1 - r > r - value_2 \Leftrightarrow guard = 1 \wedge sticky = 1 \\ value_1 - r = r - value_2 \Leftrightarrow sticky = 0 \\ value_1 = value_2 \Leftrightarrow guard = 0 \wedge sticky = 0 \end{array}$$

This is, however, not quite enough information to return the correct overflow condition. If $r \geq 2^{EMax'-Bias'}$, this information is lost. Conversely, it is not possible to determine the overflow condition before rounding as the condition *Inf'* cannot be tested until the final result is calculated. Thus, it is necessary to divide Error_Spec into two parts. The *inexact* and *underflow* conditions can be determined before or after rounding. The design decision is made that so many error conditions as possible will be determined after rounding in order that the precondition of the module is simpler. Thus, the following decomposition is valid (the validity is demonstrated by the theorem):

Error_Before

$$\begin{array}{l} Error\_Signature \\ \hline overflow \in errors' \Leftrightarrow \text{abs } r \geq 2^{EMax'-Bias'} \end{array}$$

Error_After

$$\begin{array}{lcl} Error\_Signature; errors : \mathbb{P}\, Errors & & \\ \hline overflow & \in errors' \Leftrightarrow & \text{abs } r \geq 2^{EMax'-Bias'} \\ inexact & \in errors' \Leftrightarrow & r \neq value' \\ overflow & \in errors' \Leftrightarrow & overflow \in errors \vee Inf' \\ underflow & \in errors' \Leftrightarrow & Denorm' \end{array}$$

$$\vdash Error\_Spec \sqsubseteq (Error\_Before; Error\_After)^5$$

[5] If a schema is thought of as a function from its unprimed to its primed components, the sequential composition (;) is analogous to the right composition of the two functions. The symbol $\sqsubseteq$ is used to indicate that a design decision has been made.

Error_Spec

Error_Signature

$$
\begin{array}{lll}
inexact & \in errors' \Leftrightarrow & r \neq value' \\
overflow & \in errors' \Leftrightarrow & Inf' \vee \exists Inf \bullet \text{abs } r \geq \text{abs } value \\
(underflow & \in errors' \Leftrightarrow & 0 \neq \text{abs } r < 2^{EMin'-Bias'} \\
 & \vee & \\
underflow & \in errors' \Leftrightarrow & Denorm')
\end{array}
$$

Figure 8: *Error_Spec*

Succ

$FP; FP_0$

Finite
abs *value* < abs $value_0$
$\forall FP' \mid \text{abs } value < \text{abs } value' \bullet \text{abs } value_0 \leq \text{abs } value$

Figure 9: *Succ*

Because real numbers cannot be represented in a machine, the specification which has been produced so far is not implementable. But, given the conditions of ***Bounds***, it is possible to write an implementation. The specification of the module that will be presented here is:

$$Round\_Proc \mathrel{\widehat=} \exists r : \mathbb{R}; FP_0 \bullet FP\_Round2 \wedge Bounds \wedge Error\_After \mid sign = sign'$$

(Since the sign of 0.0 often depends on factors not available to the rounding procedure, the sign of the number is maintained.)

3 The Algorithm.

There are two things to notice about the specification:

- the specification of errors is conjoined in such a way that the unprimed variable, *errors*, upon which it depends is not restricted by the the other conjuncts; thus the specification decomposes into a sequential composition of a specification on *FP* and a specification on *errors*;
- *Round*, and hence *FP_Round*, is a disjunction of specifications and thus may be implemented by a conditional.

The first observation can be formalised as:

$$\vdash Round\_Proc = (\exists r : \mathbb{R}; FP_0 \bullet FP\_Round2 \wedge Bounds); Error\_After$$

And the second observation can be formalised as:

$$
\begin{array}{c}
\vdash \quad FP\_Round2 \wedge Bounds \\
= \\
FP\_Round2 \mid mode = ToNearest \wedge Bounds \\
\vee \\
FP\_Round2 \mid mode = ToPosInf \wedge Bounds \\
\vee \\
FP\_Round2 \mid mode = ToNegInf \wedge Bounds \\
\vee \\
FP\_Round2 \mid mode = ToZero \wedge Bounds
\end{array}
$$

The first observation has the obvious implication that the module can be implemented as the sequence of two smaller programs, the first of which sets the correct approximation and the second of which returns the correct error conditions.

The second observation leads to a decomposition because each of the disjuncts is disjoint (i.e. the conjunction of any two is not satisfiable). Thus, a conditional can be formed in which the guards discriminate according to the rounding mode.

The following annotated program constitutes a proof of correctness. The predicates in braces, e.g. $\{\phi\}P\{\psi\}$, mean that if P is executed in a state satisfying ϕ, then it is guaranteed to terminate in a state satisfying ψ. Some of the conjuncts of the assertions are omitted for the sake of clarity. The first assertion is called the precondition of the program – if this does not hold on entry to the program, neither is it guaranteed to terminate nor, if it does, to terminate in any sensible state. The rules relating the program to the assertions are described in [Gries], [Dijkstra] and

[Hoare]. A brief description follows:

Rule 1 *The program* SKIP *does nothing but terminate:*

$$\vdash \{\phi\}\texttt{SKIP}\{\phi\}$$

Rule 2 *If the expression* e *can be evaluated correctly (i.e. there is no division by zero etc.), then if the state is required to satisfy ϕ after termination, it must satisfy ϕ with* e *substituted for* x *before:*

$$\vdash \{D\mathtt{e} \wedge \phi[\mathtt{e}/\mathtt{x}]\}\mathtt{x} := \mathtt{e}\{\phi\}$$

Rule 3 *If* P *starts in state ϕ and terminates in state ψ and* Q *starts in state ψ and terminates in state χ, then* P *followed by* Q *starts in state ϕ and terminates in state χ:*

$$\{\phi\}\mathtt{P}\{\psi\} \wedge \{\psi\}\mathtt{Q}\{\chi\} \vdash \{\phi\} \begin{array}{c}\texttt{SEQ}\\ \mathtt{P}\\ \mathtt{Q}\end{array} \{\chi\}$$

Rule 4 *The rule for conditionals is that if* P *starts in a state satisfying ϕ and its guard and terminates in state ψ and similarly for* Q *then the conditional composition can start in a state which satisfies one or other of the guards and ϕ and terminate in a state satisfying ψ:*

$$\{b_P \wedge \phi\}\mathtt{P}\{\psi\} \wedge \{b_Q \wedge \phi\}\mathtt{Q}\{\psi\} \vdash \begin{array}{c}\texttt{IF}\\ b_P\\ \mathtt{P}\\ b_Q\\ \mathtt{Q}\end{array} \{(b_P \vee b_Q) \wedge \phi\}\ \{\psi\}$$

Rule 5 *The precondition of a program may be strengthened:*

$$(\chi \Longrightarrow \phi) \wedge \{\phi\}\mathtt{P}\{\psi\} \vdash \{\chi\}\mathtt{P}\{\psi\}$$

Rule 6 *The postcondition of a program may be weakened:*

$$(\chi \Longleftarrow \psi) \wedge \{\phi\}\mathtt{P}\{\psi\} \vdash \{\phi\}\mathtt{P}\{\chi\}$$

The most obscure line is the following: `nat := nat + (guard ∧ (sticky ∨ nat))` . This is derived from: `nat := nat + ((guard ∧ sticky) ∨ (guard ∧ (nat ∧ 1)))` . Using `guard = guard ∧ 1` and the commutativity and associativity of ∧, the last part of the expression reduces to `guard ∧ nat`. Now, ∧ distributes through ∨ to give the optimised expression.

The original expression can be seen to be correct by studying the inequalities used to define *RoundToNearest.*

```
{overflow ∈ errors ⟺ r ≥ 2^(EMax−Bias)}
{r ≥ 0 ⟹ Below[FP/FP']}
{r ≤ 0 ⟹ Above[FP/FP']}
SEQ
  IF
    mode = ToZero
      SKIP
      {FP_Round2[FP/FP']}
    mode = ToNegInf
      IF
        sign = 0
          SKIP
        sign ≠ 0
          nat := nat + 1
          {FP_Round2[FP/FP']}
    mode = ToPosInf
      IF
        sign = 0
          nat := nat + 1
        sign ≠ 0
          SKIP
          {FP_Round2[FP/FP']}
    mode = ToNearest
      nat := nat +
             (guard ∧ (sticky ∨ nat))
      {FP_Round2[FP/FP']}
  {overflow ∈ errors ⟺ r ≥ 2^(EMax−Bias)}
  errors := errors ∩ {overflow}
  {underflow, inexact ∉ errors}
  {overflow ∈ errors ⟺ r ≥ 2^(EMax−Bias)}
  IF
    Inf
      errors := errors ∪ {overflow}
    ¬ Inf
      SKIP
  {overflow ∈ errors ⟺ Inf ∨ r ≥ 2^(EMax−Bias)}
  {underflow ∉ errors}
  IF
    Denorm
      errors := errors ∪ {underflow}
    ¬ Denorm
      SKIP
  {underflow ∈ errors ⟺ Denorm}
  {inexact ∉ errors}
  IF
    (sticky ∨ guard) ≠ 0
      errors := errors ∪ {inexact}
    (sticky ∨ guard) = 0
      SKIP
  {inexact ∈ errors ⟺ r ≠ value}
  {FP_Round[FP/FP']}
```

Conclusions

It is often heard said that formal methods can only be applied to practically insignificant problems, that development costs in large products are too high, and that the desired reliability is still not achieved. Al-

though the problem presented here may seem insignificant, it is only a small part of a large body of work which has been undertaken to implement a provably correct floating-point system. This work develops the system from a Z specification to silicon implementation – an achievement which cannot be considered insignificant. The formal development was started some time after the commencement of an informal development and has since overtaken the informal approach. The reason for this was mainly because of the large amount of testing involved in the intermediate stages of an informal development – a process which becomes less necessary with a formal development.

As for reliability, that remains to be seen. However, the existence of a proof of correctness means that mistakes are less likely and can be corrected with less danger of introducing further mistakes. Errors can arise in two ways: first, a simple mistype in the program; or a genuine error in the proof. Because of the steps in the development, the effect of this can be limited. Either, a fragment of program is wrong and can be corrected without affecting any larger scale properties of the program; or, the initial decomposition was at fault, in which case most of the development may have to be reworked. If the latter scenario seems a little dire, remember that decomposition is a prerequisite of any structured programming methodology but errors at this stage are more likely to be discovered in a formal development. Furthermore, there are now two ways to discover bugs and a way to show that they are not present. The possibility of automatic proof-checkers gives some hope that programmers will be able to guarantee the quality of a program more reliably than an architect can guarantee the robustness of a house.

This example, however, does demonstrate some of the advantages which can be gained from a formal specification. Specifications often become modified – either the customer changes her mind or the original description of the problem is found to be at fault. Trying to modify a badly documented system is disastrous. Trying to modify a well documented system is, at best, error prone. Using a formal specification, it is possible to determine which parts of the system to change and, moreover, how to change them without affecting unmodified parts. For instance, if the specification of error conditions were to change, it would be possible to prove that only the second part of the rounding module and, perhaps, its precondition need be changed. The modification can take place without having to resort to various levels of scale. Likewise, in the development stage, the specification exists to reason about how proposed modules will fit together. Moreover, modules may be reused with greater confidence because there is a precise description of what each one does.

The advantages of a non-algorithmic formalism speak for themselves. The language used here bears a formal relation to its implementation and can be transformed to emulate the structure of a program. On the other hand, the high-level specification can be written to bear a close relationship to a natural language description – there are many mathematical idioms which already exist to formalise seemingly intractable descriptions. This paper has assumed some familiarity with the IEEE Standard, but it is desirable to use the formalism as a supplement to a natural language specification to which reference can be made in case of ambiguity.

References

[Abrial] Abrial, J-R., Schumann, S.A. & Meyer, B. Specification Language Z. *Massachusetts Computer Associates, Inc. 1979.*

[Barrett] Barrett, G., Formal Methods applied to a Floating-point Number System. *Internal Report. Programming Reasearch Group, Oxford University, 1987.*

[Dijkstra] Dijkstra, E.W. A discipline of programming. *Prentice-Hall, 1976*

[Gries] Gries, D. The science of programming. *Springer-Verlag, 1981*

[Hayes] Hayes, I. (ed.) Specification Case Studies. *Prentice Hall, 1987*

[Hoare] Hoare, C.A.R. An axiomatic Basis for Computer Programming. *CACM 12 (1969). pp.576-580,583.*

[IEEE] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985, New York. August, 1985.*

[inmos] inmos, ltd. The occam Programming Manual. *Prentice Hall. 1984*

[Z] Sufrin, B.A., Sørensen, I.H., Saunders, J.W., Woodcock, J.C.P. et al The Z Handbook. *Programming Research Group, Oxford University. To appear.*