NPL Report CISE 11/97

February 1997

Testing Ada fixed point operations

B A Wichmann
Information Systems Engineering Branch
Centre for Mechanical and Optical Technology
National Physical Laboratory
Teddington
Middlesex
United Kingdom
TW11 OLW

Abstract

Since errors have been noted in Ada 83 compilers in handling fixed point operations, a study has been undertaken to ensure such errors can be located easily. The report describes an automatic test case generator written in Prolog. Results of applying the generator to three Ada compilers are given, none of which were entirely satisfactory.

NPL Report CISE 11/97

ISSN 1361–407X

National Physical Laboratory
Teddington, Middlesex TW11 0LW, United Kingdom

Approved on behalf of the Managing Director of NPL by
Dr. D. Rayner, Head, Information Systems Engineering Branch CMOT

Contents

1 Executive Summary

It has been reported [2] that some Ada compilers have significant errors in the handling of Ada fixed point operations. Since Ada fixed point is commonly used in safety-critical applications, such errors must be detected. Ada fixed point requires compiler support in a manner that does not arise with floating point as floating point is usually provided by the underlying hardware. Hence it is evident that good tests are required for Ada fixed point which can be used to ensure critical applications are free from such errors.

In Ada 83, the definition of fixed point has several implementation options which makes testing difficult. The Ada validation facility cannot even assume 2's complement arithmetic which reduces the effectiveness of the testing undertaken during validation.

In this report a *test generator* is described. This allows tests to be generated involving the numeric types used in a specific application, taking into account the implementation characteristics.

This report shows that the test generator (written in Prolog[4]) is effective in generating the test cases and in revealing errors in implementations of Ada 83. For all the systems that have been tested using the tool, problems have been found.

2 Introduction

In the study of the determinism of SPARK [2], it was noted that errors remain in Ada compilers over the handing of fixed point operations. These problems arise because of the complexity of the fixed point model in the language and that the Ada Compiler Validation Capability (ACVC) is comparatively weak in this area.

This situation is made worse by the ability to specify the attribute `'Small` which plays a key role in the accuracy of operations, since this value gives the smallest positive value of the type, for which all other values are a multiple. If the attribute is not specified, then the default is a power of two, which implies that all type conversion operations are logically equivalent to arithmetic shifts and therefore straightforward to implement.

This report demonstrates a method of providing a better test facility for fixed point which can then be used to provide higher assurance for critical applications. In fact, it appears that many safety critical applications do currently use fixed point.

In theory, it would be possible to add the proposed testing method to the Ada Program Test Generator (APTG) [1]. This does not seem the best option, since the testing proposed here does make some assumptions which cannot be made for Ada systems in general. Also, it would be difficult to extend the APTG (which is written in Ada) to undertake the level of testing proposed here (since Ada does not have unbounded arithmetic). Hence a separate testing tool is described here.

3 Design requirements

Many safety critical systems for which the accuracy of fixed point is an issue, use the SPARK subset. In some situations, the Ada compiler is used in a mode which does not support the full language. In consequence, it is highly desirable that the tests are written in the SPARK subset, or can be easily converted into that subset.

Except in the case of gross errors, the requirement is to check the rounding and truncation with extreme care, which implies not relying upon other operations within the systems being tested. This gives rise to an implementation problem. For instance, checking 32-bit operations may require rather more than 32-bits for the checking code, if excessive complexity in the code is to be avoided.

An additional consideration is that the test programs should be capable of being proved correct by suitable tools which might be developed in the future. We resolve this by producing a generator which merely produces Ada Boolean expressions whose value should be True. These expressions can then be incorporated into a test program (with the relevant numeric type definitions) in numerous ways.

Another issue is the number of potential test cases. The fixed point multiplication and divide operations involves three types, and each type can use the default value for `'Small`, or specify that value. However,

for any specific application, the fixed point types will be known and hence it is feasible to concentrate testing on those.

>From the above considerations, the preferred method of testing is to *generate* self-checking, correct programs in a similar manner to the APTG [1]. However, to avoid the complexities of multi-length arithmetic, the generator should be written in a language which supports unbounded integers. Initially, a choice was made to use Haskell [3] since it has direct support for rational arithmetic as well as unbounded integers. However, Haskell is a research tool for functional programming, and as such, is not stable. Changes have been made to the I/O system which does not make the language appropriate for a research *contract*. After reviewing the situation, it was decided to use Prolog, since many implementations support unbounded integers, and in other respects, it provides the facilities required. The use of Prolog has proved a success.

3.1 Semantic model of fixed point operations

The proposed testing would use the model proposed in [2]. This implies that the freedom for providing extra precision in Ada 83 would be regarded as an error. It does not appear that existing implementations exploit this freedom and therefore the assumption would have no effect.

It does appear that the attribute `'Machine_Rounds` is not correctly set for some implementations, and hence testing must take into account the value of this attribute (i.e. for every operation in which this attribute would have an effect, we would check the attribute has the correct value). Of course, an implementation could always set the value to False, since this places the weakest constraint on the values of numeric operations.

A significant problem arises with overflow for several reasons:

- The SPARK subset assumes the code is exception-free;
- The minimal range that must be supported without overflow depends upon the derivation mechanism which is not part of SPARK;
- The actual range supported can depend upon optimization, as allowed in section 11.6 of the RM (Reference Manual for Ada 83);
- Ada 9X and Ada 83 are different here, since for Ada 9X, the predefined floating point types are regarded as unconstrained.

The conclusion here is to test execution that should be overflow-free, and that the range of the testing can be specified by effectively giving the attributes `'Upper` and `'Lower` as defined in [2]. This values are effectively `'Base'Last` and `'Base'First` respectively.

3.2 Test program specification

The generator should take as input a number n of numeric type definitions. Note that integer and floating point types must be included in order to test the conversion operations.

The output is a test program, in a simple subset of Ada, which tests the fixed point operations involving the specified types, and the conversions to/from the other numeric types. As noted above, we just produce Boolean expressions, which are then processed to incorporate into an Ada program.

We would not restrict ourselves to the SPARK subset in the resulting Ada test program for the following reasons:

1. To ensure that `Constraint_Error` and `Numeric_Error` are not raised, we need to insert a handler for these exceptions.
2. The absence of floating point equality makes it awkward to check operations which produce a floating point result.

However, in practice, these two departures from the SPARK subset should make little difference. Some hand modifications to the generated tests should provide strict compliance with the SPARK subset.

4 Functional Specification

For each type, one needs the following information

Integer. `'First`, `'Last`. It is also necessary to distinguish type `Integer`, due to its use with fixed point multiply and divide;

Fixed point. `'Small`, `'First`, `'Last`, `'Upper`, `'Lower`, `'Machine_Rounds`. `'Small` would be rational, the rest an integer multiple of `'Small` (except `'Machine_Rounds`, of course). However, in order to generate the type definition, one needs `'Delta` and whether `'Small` is fixed by default;

Floating point. It is best to assume that the exponent range will be large enough. Then all that is required is the `'Machine_Radix`, `'Machine_Rounds`, and `'Machine_Mantissa`.

For each fixed type, take the limiting values and a few intermediate values such as 0.0 and ± 1.0. Use these values as operand values for the operations and compute the resulting model interval using rational arithmetic. Output corresponding testing code, taking into account the special case of the singleton interval. Special tests are needed to ensure the result is in range.

One issue is that of subtypes. Components of a composite type in which the size of the component is constrained imply that the compiler must generate different code, and this should be checked. Optimizing compilers might cut off the bits, rather than implement the check. However, since it is assumed the code is exception-free, it does not seem this can be checked. Hence, with reluctance, we conclude that checks on subtypes would not be useful.

Given an operation and operand values, it is easy to compute the exact (rational) result. From that, it is not difficult to compute the model interval for the result type. This model interval could overflow, but if it does not, then checking code can be issued for this particular case.

One possibility would be to store test cases on a data file, so that the program cannot be optimized by the presence of constant expressions. However, one would wish to avoid input and output of real data which might be suspect due to rounding problems. Hence the data should be integers, which in the case of fixed point types, can be just the multiple of `'Small`. This solution is not ideal, since systems on bare boards cannot be tested. The choice made here is to generate explicit tests and not use a data file.

One issue that must be addressed is that of the rounding of literals. We assume they will be rounded and test for this. An assumption must be made that a literal giving a model number exactly will be converted exactly.

Hence the following tests are required, where `FX` is a fixed point type, `FP` is a floating point type, `Int` an integer type and `Integer` the predefined type:

round: Rounding of literals for type `FX`; (We assume that the result is rounded, but test for this.

addx: `FX + FX`; (Operations **addx** .. **mulix** are exact, but overflow could occur.)

subx: `FX - FX`;

negx: `-FX`;

absx: `abs FX`;

mulxi: `FX*Integer`;

mulix: `Integer*FX`;

convix: `Int(FX)`; (Required to round.)

convxi: `FX(Int)`; (Rounding depends upon the value of `FX'Machine_Rounds`.)

convfx: `FP(FX)`; (Rounding depends upon the value of `FP'Machine_Rounds`.)

convxf: `FX(FP)`; (Rounding depends upon the value of `FX'Machine_Rounds`.)

convxx: `FX1(FX2)`; (Rounding depends upon the value of `FX1'Machine_Rounds`.)

mulpxx: `FP(FX1*FX2);` (Rounding depends upon the value of `FP'Machine_Rounds`.)

mulixx: `Int(FX1*FX2);` (Required to round.)

divpxx: `FP(FX1/FX2);` (Rounding depends upon the value of `FP'Machine_Rounds`.)

divixx: `Int(FX1/FX2);` (Required to round.)

divxxx: `FX1(FX2/FX3);` (Rounding depends upon the value of `FX1'Machine_Rounds`.)

mulxxx: `FX1(FX2*FX3);` (Rounding depends upon the value of `FX1'Machine_Rounds`.)

The first seven cases involve one fixed point type, the next five cases are type conversions involving a fixed point type and another numeric type, and the last six cases involves two fixed point types and one numeric type for the result.

There is a problem with a test in which the computed result is within the range `'Lower..'Upper`, but not within `'First..'Last`. Ideally, the computed expression would be adjusted to give a value within `'First..'Last`, before any other processes can proceed (without the danger of overflow). Here, we test the full range, and hence the types must be defined so that `'Lower='First` and `'Upper='Last`.

Clearly, the generation program needs to be able to output Ada code, which includes literals without any loss of precision.

Note that the relational operators and membership tests will be used in the checking code. Indeed, checks should be made in a redundant fashion, using all these operations. In this generator, only `in` and = are used.

The generation of tests can be simplified if variables are not used. However, it is essential to ensure that literals are treated as being of the type for which the test is designed. Hence to test `Int*FX`, we cannot generate `3*4.0`, but produce `Int(3)*FX(4.0)` instead. Even then, a compiler could optimize the execution of the expression by undertaking the evaluation in the compiler. Clearly, the best option is to be able to test the case in which this optimization can be performed *and* those cases in which it cannot. This can be arranged by generating: `Ident_Int(3)*Ident_FX(4.0)`, where `Ident_`*type* is the name of an identity function for a type. This function can be an in-line function which can clearly be optimized, or a function whose body is not available at the point of the compilation of the test code (and hence cannot be optimized).

An interesting question arises about rounding and producing a result which does not overflow. Given a fixed point type with `'Small` = 1.0 and 7 bits in the unsigned value, then the range of the type is −128.0 .. 127.0. What happens if the mathematical result of an operation is 127.1? If the type rounds, then should the rounded result of 127.0 be produced, or can the result be overflow? We use the 'overflow before rounding' rule which implies that the result overflows, and therefore this case is not tested (another testing assumption).

5 Results obtained

Three different systems have been used to try out the generator, two at NPL and one at BAeSEMA. Problems noted have been logged and these have been resolved or documented. The results in each case have been as follows:

NPL Sun^{TM}/3 system. This system ran on old hardware that was not under any maintenance agreement, but use was made of this system, since there is no charge made. Tests were run which located a fault with the **in** membership test. When a complete set of tests were generated, the compiler failed to process the tests in one compilation due to insufficient memory. Hence it was decided to abort the use of this system.

NPL Digital^{TM} Ada compiler version 2.3, under VMS^{TM} 5.5-2. A complete set of tests was generated using the standard types plus a 'binary' fixed point type. The only reported 'error' was that the compiler does not round fixed point literals to the nearest model number. This error was checked using the Unix^{TM} calculator `bc`. Of course, this feature of VAX^{TM} Ada is not an error in the

implementation, but arises from the assumptions made in the testing, see page 7. The 'error' has been reported to Digital^{TM}.

Alsys (TeleSoft) TeleGen2 VAX^{TM}/E386 v3.2.5 cross compiler. This compiler runs under VMS^{TM} on the VAX^{TM} generating code for a bare target Intel 80386 or 80486 processor. It was tested using a fixed point type with a `'Small` of 0.01, several binary fixed point types and the predefined types. The first attempt at testing revealed an error in the handling of the largest value of the non-binary type. This error required that the tests be re-generated to avoid repeated reporting of the error (the error caused `Constraint_Error` to be raised and the rest of the tests to be aborted). Problems with this type with a `'Small` of 0.01 could not be avoided entirely. Firstly, the type definition with a representation clause giving a size of 16 was rejected by the compiler. When this representation clause was removed, the type was allocated 32 bits, but `Constraint_Error` was raised in some cases, which made further testing awkward. Problems were also encountered with some type conversions, which were rejected by the compiler. No errors were located in the generated tests.

The following conclusions arose from the testing of the generator:

To undertake testing on a properly controlled basis, such as would be necessary for accreditation by UKAS, would require a detailed manual. Application of the generator is not straightforward and needs care.

2. The fact that the literal giving the value of `FX'First` can cause `Constraint_Error` to be raised is annoying. The generator is not designed to handle this, and hence the best route is to edit the output and replace the literal by a named number.

3. The rounding of literals is clearly an issue, since a program will (in general) produce different results with different rounding algorithms and there is no requirement in the Ada standard to specify the rounding undertaken. Since static expressions are required to be handled to arbitrary precision by the language, it seems there is little reason for compilers not to round literals to the nearest machine number (when in range).

4. The generator is effective in locating faults, as illustrated by the results above.

6 Conclusions

The test generator presented here shows that, given reasonable simplifying assumptions, it is possible to produce tests largely automatically. However, this system requires that operand test values are chosen by the user, and some skill must be exercised in selecting such values. It is thought that the test generator has been much less work than that which has been required to produce the corresponding tests in the ACVC. (This is not a criticism of the ACVC, since the underlying assumptions are different.)

The use of the Prolog language has proved ideal, since an essential aspect of the tests is to produce only those satisfying specific conditions. However, the program is quite long, and therefore to understand the logic, one should select just one of the 18 specific classes of tests. This then provides about 2 pages of text to understand.

Differences have been found in Prolog implementations, and the manual for the SICStus implementation used did not give the semantics for integer division for negative operands (truncation towards zero is assumed). The generator also assumes unbounded integers.

In applying this generator to the VAX^{TM} Ada compiler, a significant problem arose. We are assuming a 2's complement machine, which implies that the most negative value has no corresponding positive counterpart. If the most negative value is printed as a literal, then this value will involve a unary minus of an out-of-range positive value. Hence a compiler which does not optimize the application of the unary minus will fail by raising `Constraint_Error`. This situation arose on the VAX^{TM} for which no simple solution is available within the context of the Prolog generator using the current design. However, by editing the generator output, such negative values can be replaced by a reference to a number declaration, which overcomes the problem. In consequence, no change to the Prolog program seems necessary.

7 Acknowledgements

This work would not have been possible without the support of Dr Colin O'Halloran, via a contract from DRA-Malvern (Contract No CSM/089). Assistance at NPL was provided by Simon Ashford and Dr Tony Mansfield.

BAeSEMA provided very useful assistance in running some tests on their system, which showed some weaknesses in the generator which have been corrected. Michael Pickett followed this work and provided useful insights into the issues faced.

References

[1] S M Austin, D R Wilkins and B A Wichmann. An Ada Program Test Generator. TriAda Conference Proceedings. ACM. October 1991.

[2] B A Wichmann. Is SPARK faithful? Report to Program Validation Ltd. NPL. June 1994.

[3] Haskell Special Issue. ACM SIGPLAN Notices, Vol 27, No 5. May 1992.

[4] SICStus Prolog User's Manual. Swedish Institute of Computer Science, January 1993.

A Assumptions for testing

Here we list the assumptions made to produce the testing system.

A.1 Extra precision for fixed point not provided

Detailed description. According to RM 4.5.7, any real operation can provide more accuracy than that required by the type definition. However, in the case of a representation specification for `'Small`, stored values cannot have such excess accuracy. It appears that excess accuracy is not provided by current implementations of Ada 83.

Reason. Excess accuracy makes the testing very much more awkward, and is a major contributing factor to the weakness of ACVC tests in this area, since they must allow of this. The simplified semantic model proposed for SPARK does not provide excess accuracy, and hence the reason for this assumption.

Example. Although existing implementations may not compute excess precision, an implementation could provide some optimisation based upon the existing permission. Consider:

```
type FX is delta 0.1 range -1.0    1.0;

X: FX := FX'Small


if X/2*2 = X then
   -- The above expression could be optimized to True.
   -- Will be False in this case with our assumptions, since
   -- X/2*2 will be either 2*FX'Small or 0.0.
end if;
```

A.2 Fixed point literals are rounded

Detailed description. The implicit conversion from *universal_real* to a fixed point type follows the same rules as other conversions, although it is implemented by the compiler (see RM 4.5.7 and 4.6(15)).

Reason. Since there is no run-time overhead, there is no reason not to round. Most existing implementations are thought to round literals. We will test for this assumption, but if those tests fail, then the subsequent tests could report incorrect results.

Example.

```
type FX is delta 0.1 range -1.0 .. 1.0
for FX'Small use 0.0625;

X: FX := FX'Delta;


if X = 0.125 then
   -- The above expression must be True
end if;
```

A.3 2's complement integer and fixed point arithmetic

Detailed description. Ada explicitly allows 1's complement arithmetic for both integer and fixed point. For type `Integer`, this determines the first and last values, given the value Bits above.

Reason. This avoids lots of special cases which are virtually impossible for us to check, since we have no access to a 1's complement machine.

Example.

```
type FX is delta 0.1 range -1.0 .  1.0;
for FX'Small use 0.0625;

-- FX'First = -1.0, FX'Last = 0.9375.
end if;
```

A.4 1's complement floating point arithmetic

Detailed description. Ada implicitly allows 2's complement arithmetic for floating point, although the model is 1's complement.

Reason. This avoids lots of special cases which are virtually impossible for us to check, since we have no access to a 2's complement floating point machine. The only 'significant' 2's complement floating point machine appears to be the 1750A (A non-commercial architecture used primarily by US defence companies).

Example.

```
if -Float'First /= FX'Last then
   -- will never be executed.
end if;
```

A.5 Rounding after overflow

Detailed description. Testing is not undertaken if the resulting interval overflows, even if the rounded result would not overflow.

Reason. In Ada 83, if the result interval overflows, then the result is not defined. An implementation would need to appeal to 11.6 to produce a result.

Example.

```
type FX is delta System.Fine_Delta range -1.0 .. 1.0;
X: FX;
...
X := FX(Float(1.0 - 0.75*System.Fine_Delta));
-- Since we assuming a 2's complement machine, the value of FX'Last is
--   1.0 - System.Fine_Delta. We are assuming that the expression above
--   is computed exactly in type Float.

-- The above example is not tested, but may well execute to produce the
-- correctly rounded result.
```

A.6 Full range of type used

Detailed description. The values of `'First` and `'Last` are not used in the test generator, and hence if these values do not cover the full range of the type, a constraint check could fail.

Reason. This is just a simplifying assumption. By this means, the values of the attributes `'First` and `'Last` need not be known to the test generator. Effectively, it means the testing is on types rather than subtypes.

Example.

```
type FX is delta 0.1 range -1.0 .. 1.0;
X: FX;
-- The tests are generated from information which states that Bits =31,
--  and hence the range provided for FX is much greater than that declared.
--  This large range will be fully used for testing to ensure the total
--  range of operations are supported.
...
X := -2.0;  -- will raise Constraint_Error
```

B Information required for testing

This section consists of the text sent to those who wish their numeric types to be tested by means of the generator. It is implicit in this testing that the assumption above are satisfied.

Information is required for each of the integer, fixed point and floating point types used in the critical application. The values of the attributes can either be determined by consulting the compiler documentation, or by running a program to print them out.

Integer types. For each integer type, in which one must be denoted as `Integer`:

- The Ada name of the type;
- The value of `'First` and `'Last`.

Note that we assume that the machine is 2's complement and hence `'First = -'Last - 1`. All tests are conducted on the base type of types, and hence if one had `type Digit is range 0..9;` the testing would be on `Digit'Base` which is likely to be byte arithmetic.

Fixed point types. For each fixed point type:

- The Ada name of the type;
- The value of `'Small`, `'Delta`, `'Base'First` (which I call `'Lower`), `'Base'Last` (which I call `'Upper`);
- The value of `'Machine_Rounds`;
- Any representation specification, such as `'Size`.

Note that the tests are effectively undertaken on base types, and hence the type declaration should not imply a physical range check.

Floating point types. For each floating point type:

- The Ada name of the type;
- The value of `'Machine_Radix`, `'Machine_Mantissa` and `'Machine_Rounds`

C Specification of test cases

For each of the 18 test cases noted above, we need to define the method of test selection. For each numeric data type, a set of values is defined, denoted by $S(FX)$ (for type FX). The general method of testing is to generate test cases for each operand in the appropriate set. The test sets should be chosen to include boundary values, to ensure input boundary values will be tested.

The Prolog program contains the data values for each type. Apart from boundary values, other values can be relevant to output boundaries which would not necessarily be tested unless care is taken with the data value selection. For instance, if one was testing scaled fixed point multiplication, then it would be advisable to have a test case which produced the largest value to the resulting type. The Prolog program does not ensure such cases are generated. Another internal boundary is that of conversion to integers in which values having a fractional part of 0.5 are a special case.

Another problem with the generation of the test cases is the Ada form in which they should be produced. The form chosen here is to produce a Boolean which should be True on a line of output. Apart from comment lines separating test batches, these lines need to be included into an Ada program for testing. The simplest way to produce executable code is to call a procedure with a single Boolean parameter.

The APTG does *not* necessarily generate procedure calls to check the execution, since the parameter passing mechanism itself could cause different code to be produced than that in ordinary computational sequences. Hence this test system, leaves the choice open to the user.

If we wish to check addition for a fixed point type FX, we clearly need to generate a Boolean which ensures that the literals are converted to type FX. However, generating:

```
FX(1.0) + FX(1.0) = 2.0
```

is unlikely to be effective, since many compilers will evaluate the expression to True without generating code. To avoid this, all the generated code uses identity functions, denoted by $Type\_1$, and $Type\_2$. Hence the above case produces:

```
FX(1.0) + FX_1(FX(1.0)) = FX_2(2.0)
```

The user can edit the output to remove such function calls, or provide an implementation (and such an implementation could be in-lined, or compiled after compilation of the test without the use of `Pragma InLine`).

The test generator is highly modular, and can be easily used to generate tests for specific operations. Since the process is largely automatic, it is feasible to produce a large number of tests.

Notes below explain the test case generation in those cases in which the above strategy does not cover the situation.

round: Rounding of literals for type `FX`; (We assume that the result is rounded, but test for this.)

For each fixed point type, check that the adding or subtracting `0.25*'Small` to the literal value does not change the actual value (assuming in range). The addition and subtraction are handled as separate subcases in the Prolog generator.

addx: `FX + FX;`

No additional notes.

subx: `FX - FX;`

No additional notes.

negx: `-FX;`

No additional notes.

absx: `abs FX;`

No additional notes.

mulxi: `FX*Integer;`

No additional notes.

mulix: `Integer*FX;`

This test case is combined with the previous one since the conditions for test case generation are the same.

convix: `Int(FX);` (Required to round.)

Note that the rounding does not define a unique result when the value is half way between two integers — in that case, the result cannot be tested by strict equality, so use the form `in Int N..N+1`.

convxi: `FX(Int);` (Rounding depends upon the value of `FX'Machine_Rounds`.)

If `FX'Machine_Rounds` is false, or the true result is midway between two model numbers, the result of the operation is not uniquely determined, and hence strict equality cannot be used. In this case, check the upper or lower bound on the result using a relational operator.

convfx: `FP(FX);` (Rounding depends upon the value of `FP'Machine_Rounds`.)

If `FP'Machine_Rounds` is false, or the true result is midway between two model numbers, the result of the operation is not uniquely determined, and hence strict equality cannot be used. In this case, check the upper or lower bound on the result using a relational operator.

convxf: `FX(FP);` (Rounding depends upon the value of `FX'Machine_Rounds`.)

If `FX'Machine_Rounds` is false, or the true result is midway between two model numbers, the result of the operation is not uniquely determined, and hence strict equality cannot be used. In this case, check the upper or lower bound on the result using a relational operator.

convxx: `FX1(FX2);` (Rounding depends upon the value of `FX1'Machine_Rounds`.)

This test is only performed when `FX1` and `FX2` are distinct types. If `FX1'Machine_Rounds` is false, or the true result is midway between two model numbers, the result of the operation is not uniquely determined, and hence strict equality cannot be used. In this case, check the upper or lower bound on the result using a relational operator.

mulpxx: `FP(FX1*FX2);` (Rounding depends upon the value of `FP'Machine_Rounds`.)

If `FP'Machine_Rounds` is false, or the true result is midway between two model numbers, the result of the operation is not uniquely determined, and hence strict equality cannot be used. In this case, check the upper or lower bound on the result using a relational operator.

mulixx: `Int(FX1*FX2);` (Required to round.)

Note that the rounding does not define a unique result when the value is half way between two integers — in that case, the result cannot be tested by strict equality, so use the form `in Int N..N+1`.

divpxx: `FP(FX1/FX2);` (Rounding depends upon the value of `FP'Machine_Rounds`.)

If `FP'Machine_Rounds` is false, or the true result is midway between two model numbers, the result of the operation is not uniquely determined, and hence strict equality cannot be used. In this case, check the upper or lower bound on the result using a relational operator.

divixx: `Int(FX1/FX2);` (Required to round.)

Note that the rounding does not define a unique result when the value is half way between two integers — in that case, the result cannot be tested by strict equality, so use the form `in Int N..N+1`.

divxxx: `FX1(FX2/FX3);` (Rounding depends upon the value of `FX1'Machine_Rounds`.)

If `FX1'Machine_Rounds` is false, or the true result is midway between two model numbers, the result of the operation is not uniquely determined, and hence strict equality cannot be used. In this case, check the upper or lower bound on the result using a relational operator.

mulxxx: `FX1(FX2*FX3)`; (Rounding depends upon the value of `FX1'Machine_Rounds`.)

If `FX1'Machine_Rounds` is false, or the true result is midway between two model numbers, the result of the operation is not uniquely determined, and hence strict equality cannot be used. In this case, check the upper or lower bound on the result using a relational operator.

D The Prolog generator

The NPL quality system has specific requirements for software development in which an item of software is classified into one of four levels. Level 3 corresponds to that appropriate for fully supported 'commercial' software. Since this software has been produced to demonstrate the feasibility of the proposed testing method, and there is not necessarily a commitment by NPL to fully support the software, this Prolog program is put at Level 2. Hence the following disclaimer applies:

> *This software has not been fully subjected to NPL's Quality Assurance procedures. No warranty or guarantee applies to this software, and therefore any users should satisfy themselves that it meets their requirements.*

This program was written and developed using SICStus Prolog on a Sun-SPARK machine.

The program is in two parts: the description of the Ada numeric types which the user of the generator needs to set for any specific application, and the main generator itself.

The numeric types used to undertake preliminary tests on the Sun^{TM}/3 VADS compiler is listed below. The details of each numeric type are given as a list of rationals, or integers (in the case of integer types). The last parameter is a list of operand test values for each type. In the case of floating point or fixed point types, these are all given with the same denominator (and for convenience, are in numeric order).

```
/* File of VADS test types */

/* Integer types, Ada name, 'First, 'Last , [List of test values] */

i_type('Integer', -2147483648, 2147483647,  -2147483648,-1,0,1,2147483647]).

i_type('Short_Integer', -32768, 32767, [-32768, -1, 0, 1, 32767]).

/* Fixed point types:
   Ada name, 'Small, 'Lower, 'Upper, 'Machine_Rounds, [List values] */

/* 'Machine_Rounds is True encoded as =0 */

fx_type('FX1', 1/2147483648, -2147483648/2147483648, 2147483647/2147483648, 0,
         [-2147483648/2147483648, -1073741824/2147483648, -1/2147483648, 0/2147483648,
           1/2147483648, 1073741824/2147483648, 2147483647/2147483648]).

fx_type('Duration', 1/1024, -2147483648/1024, 2147483647/1024, 0,
         [-2147483648/1024, -1/1024, 0, 1/1024, 2147483647/1024]).

/* Floating point types:
   Ada name, 'Machine_Radix,  'Machine_Mantissa)**'MR, 'Machine_Rounds, [List values] */

fp_type('Float', 2, 16777216, 0, [-1/1, -1/2, 0/1, 1/2, 1/1]).
```

The main Prolog program, naturally split into 18 parts is listed below. The following points should be noted about the program:

1. For each of the test cases, the relevant types and operand values are found; then any numerical conditions are checked in the clause starting `satisfy`; then the test operands are output; then the bounds on the correct result computed; finally the resulting condition is output.
2. The / operator is not evaluated, since the generator must not depend upon the accuracy of the floating point operations of the Prolog implementation. Hence the operator is merely used as a notational convenience (it is very helpful in this respect).
3. The operator `..` is defined also for notational convenience. This is used to define the one value or range of values for the result.

4. The expected results are computed as a rational, but this is held as an expression tree (in general). Hence the real computation of the result is that needed to output the values in decimal.

```
/* For selecting a specific numeric type */

pick_fx(N, S, Lu, Up, MR, V) :-
        fx_type(N, S, Lu, Up, MR, L),
        member(V, L).

pick_i(N, Fi, La, V) :-
        i_type(N, Fi, La, L),
        member(V, L).

pick_fp(N, Ra, Ma, MR, V) :-
        fp_type(N, Ra, Ma, MR, L),
        member(V, L).

member(X, [X|_])
member(X, [_|L])  - member(X, L)

/* Service routines */

write_x(A/B)
                B < 0 ->
                write_x1((-A)/(-B))
                write_x1(A/B).

write_x1(A/B) :-
        (       A >= 0 ->
                write_xp(A/B)

                write('(-'),
                Z is -A,
                write_xp(Z/B),
                write(')')


write_xp(A/B)  :- Z is A//B,
                write(Z), write('.'),
                F is A - (Z*B),
                G is (10*F) // B,
                write(G), write_dp(((10*F) - G*B)/B)

write_dp(A/B) :-
        (       A =:= 0 ->
                write('')

                G is (10*A) // B,
                H is ((10*A) - G*B),
                write(G), write_dp(H/B)


write_i(I) :-
        A is I,
        (       A >= 0 ->
                write(A)

                write('('
```

```
                write(A),
                write(')'


:- op(450, xfx,

round_int(A/B, 1, X .. Y )          /* undefined rounding */
      I is A // B,
      Rem is A mod B,
      (   Rem =:= 0 ->
          (X is I, Y is I)

          Rem > 0 ->
          (X is I, Y is I+1)

          Rem < 0 ->
          (X is I-1, Y is I)


round_int(A/B, 0, X .. Y ):-        /* rounded unless half-way */
      I is A // B,
      Rem is A mod B,
      (    Rem =:= 0 ->
           (X is I, Y is I)

           Rem > B // 2 ->
           (X is I+1, Y is I+1)

           Rem =:= B // 2 ->
           (X is I, Y is I+1)
      ;
           Rem > 0 ->
           (X is I, Y is I)

           Rem > -B // 2 ->
           (X is I, Y is I)

           Rem =:= -B // 2 ->
           (X is I-1, Y is I)

           Rem < 0 ->
           (X is I-1, Y is I-1


round_fx(A/B, C/D, MR, X/D .. Y/D) :-
        round_int( (A*D)/(B*C), MR, U    V),
        X is U*C,
        Y is V*C.

round_fp(A/B, Ra, Ma, MR, (X/D) .. (Y/D)
        (    A =:= 0 ->
             (D is 1, X is 0, Y is 0)

             B < 0 ->
             round_fp((-A)/(-B), Ra, Ma, MR, (X/D) .. (Y/D))

                A < 0 ->
                Sign is -1
```

```
            Sign is +1
        ),
    Mant is Sign*A,
    normalise(Mant/B, 1/1, Ra, Ma, ER1/ER2, Rx/Ry),
    !,
    round_int(Rx/Ry, MR, U .. V),
    D is ER2,
            Sign =:= 1 ->
            (X is U*ER1, Y is V*ER1)

            (X is -V*ER1, Y is -U*ER1)


normalise(Mant/B, E1/E2, Ra, Ma, ER1/ER2, Rx/Ry)
        Mant >= B*Ma,
        Mant =< B*Ma*Ra,
        ER1 is E1, ER2 is E2, Rx is Mant, Ry is B.

normalise(Mant/B, E1/E2, Ra, Ma, ER1/ER2, Rx/Ry) :-
        Mant < B*Ma ->
           normalise((Ra*Mant)/B, E1/(E2*Ra), Ra, Ma, ER1/ER2, Rx/Ry)

normalise(Mant/B, E1/E2, Ra, Ma, ER1/ER2, Rx/Ry) :-
        Mant > B*Ma*Ra ->
           normalise(Mant/(Ra*B), (E1*Ra)/E2, Ra, Ma, ER1/ER2, Rx/Ry)


/* Test cases follow, with xxx being of the 18 classes, from round to mulxxx.

/* Select types using compute_xxx, then use satisfy_xxx to check legal case, then
   produce test output (if it succeeds). */

/* round *

compute_round1 :- /* add 'Small/4 */
        pick_fx(NX, S/B, L/B, U/B, _, X/B),
        satisfy_round(L/B, U/B, (4*X+S)/(4*B) ),
        write(NX), write('_1('),
        write(NX), write('('), write_x((4*X+S)/(4*B)), write('))'),
        write('='), write(NX), write('_2('), write_x(X/B), write(')' , nl.

compute_round2 :- /* subtract 'Small/4 */
        pick_fx(NX, S/B, L/B, U/B, _, X/B),
        satisfy_round(L/B, U/B, (4*X-S)/(4*B) ),
        write(NX), write('('), write(NX), write('_1('),
        write_x((4*X-S)/(4*B)), write('))'),
        write('='), write(NX), write('_2('), write_x(X/B), write(')'), nl.

satisfy_round(L/B, U/B, X/B1) :-
        B1 =:= 4*B,
        4*L =< X,
        4*U >= X.

test_round :-
        write('-- Start (v0.2) round'), nl,
        fail.
```

```
test_round :-
        compute_round1,
        fail.

test_round :-
        compute_round2,
        fail.

test_round :-
        write('-- Finished round' , nl

 * addx */

compute_addx :- /* sum two values */
        pick_fx(NX, S, Lu, Up, MR, VX1/B),
        pick_fx(NX, S, Lu, Up, MR, VX2/B),
        satisfy_addx(Lu, Up, VX1/B, VX2/B),
        write(NX), write('('), write_x(VX1/B), write(')+'),
        write(NX), write('_1('), write_x(VX2/B), write(')'),
        write('='), write(NX), write('_2('), write_x((VX1+VX2)/B), write(')' , nl

satisfy_addx(L/B, U/B, X1/B, X2/B)
        L =< X1+X2,
        U >= X1+X2.

test_addx :-
        write('-- Start (v0.2) addx  , nl,
        fail.

test_addx :-
        compute_addx,
        fail.

test_addx :-
        write('-- Finished addx' , nl

/* subx *

compute_subx :- /* sum two values */
        pick_fx(NX, S, Lu, Up, MR, VX1/B),
        pick_fx(NX, S, Lu, Up, MR, VX2/B),
        satisfy_subx(Lu, Up, VX1/B, VX2/B),
        write(NX), write('('), write_x(VX1/B), write(')-'),
        write(NX), write('_1('), write_x(VX2/B), write(')'),
        write('='), write(NX), write('_2('), write_x((VX1-VX2)/B), write(')'), nl

satisfy_subx(L/B, U/B, X1/B, X2/B)
        L =< X1-X2,
        U >= X1-X2.

test_subx :-
        write('-- Start (v0.2) subx'), nl,
        fail.

test_subx :-
        compute_subx,
        fail.
```

```
test_subx :-
        write('-- Finished subx' , nl.

/* negx */

compute_negx :-
        pick_fx(NX, _, L/B, U/B, _, X/B),
        satisfy_negx(L/B, U/B, (-X)/B ),
        write('-'), write(NX), write('_1('), write(NX), write('('),
        write_x(X/B), write('))'),
        write('='), write(NX), write('_2('), write_x((-X)/B),
        write(')'), nl.

satisfy_negx(L/B, U/B, X/B)
        L =< X,
        U >= X.

test_negx :-
        write('-- Start (v0.2) negx' , nl,
        fail.

test_negx :-
        compute_negx,
        fail.

test_negx :-
        write('-- Finished negx'), nl.

/* absx */

compute_absx :-
        pick_fx(NX, _, L/B, U/B, _, X/B),
        (X < 0 -> AX is -X; AX is X ),
        satisfy_absx(L/B, U/B, AX/B ),
        write('abs '), write(NX), write('_1('), write(NX), write('('
        write_x(X/B), write('))'),
        write('='), write(NX), write('_2('), write_x(AX/B),
        write(')'), nl.

satisfy_absx(L/B, U/B, X/B) :-
        L =< X,
        U >= X.

test_absx :-
        write('-- Start (v0.2) absx'), nl,
        fail.

test_absx :-
        compute_absx
        fail.

test_absx :-
        write('-- Finished absx' , nl

/* mulxi and mulix */

compute_mulxi :-
        pick_fx(NX, _, Lu, Up, _, VX/B),
```

```
        pick_i('Integer', _, _, VI),
        satisfy_mulxi(Lu, Up, VX/B, VI),
        write(NX), write('_1('), write(NX), write('('),
        write_x(VX/B), write('))*'),
        write_i(VI), write('='), write(NX), write('_2('), write_x((VI*VX)/B),
        write(')'), nl,
        /* mulix */
        write_i(VI), write('*'), write(NX), write('('),
        write(NX), write('_1('), write_x(VX/B),
        write('))'), write('='), write(NX), write('_2('), write_x((VI*VX)/B),
        write(')'), nl.


satisfy_mulxi(L/B, U/B, VX/B, VI) :-
        L =< VX*VI,
        U >= VX*VI.

test_mulxi :-
        write('-- Start (v0.2) mulxi' , nl,
        fail.

test_mulxi :-
        compute_mulxi,
        fail.

test_mulxi :-
        write('-- Finished mulxi' , nl.

 * convix *

compute_convix :-
        pick_fx(NX, _, _, _, _, A/B),
        i_type(NI, Fi, La, _),
        satisfy_convix(Fi, La, A/B),
        write(NI), write('('), write(NX), write('_1('), write(NX), write('(
        write_x(A/B), write(')))'),
        round_int(A/B, 0, X .. Y),
        ( X =:= Y -> (write('='), write(X) );
                     (write(' in '), write(X), write('      , write(Y) )
        ),
        nl.

satisfy_convix(Fi, La, A/B)
        Fi*B =< A,
        La*B >= A.

test_convix :-
        write('-- Start (v0.2) convix' , nl,
        fail.

test_convix :-
        compute_convix,
        fail.

test_convix :-
        write('-- Finished convix' , nl.

/* convxi */
```

```
compute_convxi :-
        pick_i(NI, _, _, VI),
        fx_type(NX, A/B, Lu, Up, MR, _),
        satisfy_convxi(Lu, Up, VI),
        write(NX), write('('), write(NI), write('_1('), write(NI), write('('
        write(VI), write(')))'),
        round_fx(VI/1, A/B, MR, X/D .. Y/D),
        ( X =:= Y -> (write('='), write_x(X/D) );
                     (write(' in '), write_x(X/D), write(' .. '), write_x(Y/D) )
        ),
        nl.

satisfy_convxi(Lu/B, Up/B, VI) :-
        Lu =< VI*B,
        Up >= VI*B.

test_convxi :-
        write('-- Start (v0.2) convxi'), nl
        fail.

test_convxi :-
        compute_convxi,
        fail.

test_convxi :-
        write('-- Finished convxi' , nl

/* convfx */

compute_convfx :-
        pick_fx(NX, _, _, _, _, A/B),
        fp_type(NP, Ra, Ma, MR, _),
        write(NP), write('('), write(NX), write('_1('), write(NX), write('(')
        write_x(A/B), write(')))'),
        round_fp(A/B, Ra, Ma, MR, X/D .. Y/D),
        ( X =:= Y -> (write('='), write_x(X/D) );
                     (write(' in '), write_x(X/D), write(' .. '), write_x(Y/D) )

        nl

test_convfx :-
        write('-- Start (v0.2) convfx'), nl
        fail.

test_convfx :-
        compute_convfx,
        fail.

test_convfx :-
        write('-- Finished convfx'), nl.

/* convxf

compute_convxf :-
        pick_fp(NP, _, _, _, A/B),
        fx_type(NX, S, Lu, Up, MR, _),
        satisfy_convxf(Lu, Up, A/B),
```

```
        write(NX), write('('), write(NP), write('_1('), write(NP), write('('
        write_x(A/B), write(')))'),
        round_fx(A/B, S, MR, X/D .. Y/D),
        ( X =:= Y -> (write('='), write_x(X/D) );
                     (write(' in '), write_x(X/D), write('     '), write_x(Y/D) )
        ),
        nl


satisfy_convxf(Lu/B, Up/B, C/D)
        Lu*D =< C*B,
        Up*D >= C*B.

test_convxf :-
        write('-- Start (v0.2) convxf' , nl,
        fail.

test_convxf :-
        compute_convxf,
        fail.

test_convxf :-
        write('-- Finished convxf' , nl

/* convxx */

compute_convxx :-
        pick_fx(NX2, _, _, _, _, A/B),
        fx_type(NX1, S, Lu, Up, MR, _),
        NX1 \== NX2,
        satisfy_convxx(Lu, Up, A/B),
        write(NX1), write('('), write(NX2), write('_1('), write(NX2), write('('
        write_x(A/B), write(')))'),
        round_fx(A/B, S, MR, X/D .. Y/D),
        ( X =:= Y -> (write('='), write_x(X/D) );
                     (write(' in '), write_x(X/D), write('       , write_x(Y/D)

        nl

satisfy_convxx(Lu/B, Up/B, C/D)
        Lu*D =< C*B,
        Up*D >= C*B.

test_convxx :-
        write('-- Start (v0.2) convxx' , nl,
        fail.

test_convxx :-
        compute_convxx,
        fail.

test_convxx :-
        write('-- Finished convxx'), nl.

/* mulpxx */

compute_mulpxx :-
        pick_fx(NX1,            VX1/B1),
```

```
        pick_fx(NX2, _, _, _, _, VX2/B2),
        fp_type(NP, Ra, Ma, MR, _),
        write(NP), write('('),
        write(NX1), write('_1('), write(NX1), write('('
              write_x(VX1/B1), write('))*'),
        write(NX2), write('_1('), write(NX2), write('('
              write_x(VX2/B2), write(')))'),
        round_fp((VX1*VX2)/(B1*B2), Ra, Ma, MR, X/D .. Y/D)
        ( X =:= Y -> (write('='), write_x(X/D) );
                     (write(' in '), write_x(X/D), write('       , write_x(Y/D) )
        ),
        nl.


test_mulpxx :-
        write('-- Start (v0.2) mulpxx' , nl,
        fail.

test_mulpxx :-
        compute_mulpxx,
        fail.

test_mulpxx :-
        write('-- Finished mulpxx'), nl

/* mulixx */

compute_mulixx :-
        pick_fx(NX1, _, _, _, _, VX1/B1),
        pick_fx(NX2, _, _, _, _, VX2/B2),
        i_type(NI, Fi, La, _),
        satisfy_mulixx(Fi, La, (VX1*VX2)/(B1*B2)),
        write(NI), write('('),
        write(NX1), write('_1('), write(NX1), write('('), write_x(VX1/B1), write('))*'),
        write(NX2), write('_1('), write(NX2), write('('), write_x(VX2/B2),
        write(')))'),
        round_int((VX1*VX2)/(B1*B2), 0, X .. Y),
        ( X =:= Y -> (write('='), write(X) );
                     (write(' in '), write(X), write(' .. '), write(Y) )
        ),
        nl.

satisfy_mulixx(Fi, La, C/D)
        Fi*D =< C,
        La*D >= C.

test_mulixx :-
        write('-- Start (v0.2) mulixx' , nl,
        fail.

test_mulixx :-
        compute_mulixx,
        fail.

test_mulixx :-
        write('-- Finished mulixx' , nl

 * divpxx */
```

```
compute_divpxx :-
        pick_fx(NX1, _, _, _, _, VX1/B1),
        pick_fx(NX2, _, _, _, _, VX2/B2),
        fp_type(NP, Ra, Ma, MR, _),
        satisfy_divpxx( (VX1*B2)/(VX2*B1) ),
        write(NP), write('('),
        write(NX1), write('_1('), write(NX1), write('('),
              write_x(VX1/B1), write('))/'),
        write(NX2), write('_1('), write(NX2), write('('),
              write_x(VX2/B2), write(')))'),
        round_fp((VX1*B2)/(VX2*B1), Ra, Ma, MR, X/D .. Y/D),
        ( X =:= Y -> (write('='), write_x(X/D) );
                     (write(' in '), write_x(X/D), write(' ..   , write_x(Y/D) )
        ),
        nl.

satisfy_divpxx(_/D)
        D =\= 0.

test_divpxx :-
        write('-- Start (v0.2) divpxx'), nl,
        fail.

test_divpxx :-
        compute_divpxx,
        fail.

test_divpxx :-
        write('-- Finished divpxx'), nl.

/* divixx */

compute_divixx :-
        pick_fx(NX1, _, _, _, _, VX1/B1),
        pick_fx(NX2, _, _, _, _, VX2/B2),
        i_type(NI, Fi, La, _),
        satisfy_divixx(Fi, La, (VX1*B2)/(VX2*B1) ),
        write(NI), write('('),
        write(NX1), write('_1('), write(NX1), write('('), write_x(VX1/B1), write('  /'
        write(NX2), write('_1('), write(NX2), write('('), write_x(VX2/B2),
        write(')))'),
        round_int((VX1*B2)/(VX2*B1), 0, X .. Y),
        ( X =:= Y -> (write('='), write(X) );
                     (write(' in '), write(X), write(' .. '), write(Y) )
        ),
        nl.


satisfy_divixx(Fi, La, C/D)
        D =\= 0,
        Fi*D =< C,
        La*D >= C.

test_divixx :-
        write('-- Start (v0.2) divixx  , nl,
        fail.
```

```
test_divixx :-
        compute_divixx,
        fail.

test_divixx :-
        write('-- Finished divixx' , nl

 * divxxx */

compute_divxxx :-
        pick_fx(NX2, _, _, _, _, VX2/B2),
        pick_fx(NX3, _, _, _, _, VX3/B3),
        fx_type(NX1, S, Lu, Up, MR, _),
        satisfy_divxxx(Lu, Up, (VX2*B3)/(VX3*B2) ),
        write(NX1), write('('),
        write(NX2), write('_1('), write(NX2), write('('),
              write_x(VX2/B2), write('))/'),
        write(NX3), write('_1('), write(NX3), write('('),
              write_x(VX3/B3), write(')))'),
        round_fx((VX2*B3)/(VX3*B2), S, MR, X/D .. Y/D),
        ( X =:= Y -> (write('='), write_x(X/D) );
                     (write(' in '), write_x(X/D), write('     '), write_x(Y/D)
        ),
        nl


satisfy_divxxx(Lu/B, Up/B, C/D)
        D =\= 0,
        Lu*D =< C*B,
        Up*D >= C*B.

test_divxxx :-
        write('-- Start (v0.2) divxxx'), nl,
        fail.

test_divxxx :-
        compute_divxxx,
        fail.

test_divxxx :-
        write('-- Finished divxxx' , nl.

/* mulxxx */

compute_mulxxx :-
        pick_fx(NX2, _, _, _, _, VX2/B2),
        pick_fx(NX3, _, _, _, _, VX3/B3),
        fx_type(NX1, S, Lu, Up, MR, _),
        satisfy_mulxxx(Lu, Up, (VX2*VX3)/(B2*B3) ),
        write(NX1), write('('),
        write(NX2), write('_1('), write(NX2), write('('),
              write_x(VX2/B2), write('))*'),
        write(NX3), write('_1('), write(NX3), write('('),
              write_x(VX3/B3), write(')))'),
        round_fx((VX2*VX3)/(B2*B3), S, MR, X/D .. Y/D),
        ( X =:= Y -> (write('='), write_x(X/D) );
                     (write(' in '), write_x(X/D), write('       , write_x(Y/D) )
```

```
        nl


satisfy_mulxxx(Lu/B, Up/B, C/D)
        Lu*D =< C*B,
        Up*D >= C*B.

test_mulxxx :-
        write('-- Start (v0.2) mulxxx')  nl
        fail.

test_mulxxx :-
        compute_mulxxx,
        fail.

test_mulxxx :-
        write('-- Finished mulxxx'), nl.


test :
        test_round,
        test_addx,
        test_subx,
        test_negx,
        test_absx,
        test_mulxi,
        test_convix,
        test_convxi,
        test_convfx,
        test_convxf,
        test_convxx,
        test_mulpxx,
        test_mulixx,
        test_divpxx,
        test_divixx,
        test_divxxx,
        test_mulxxx.
```

D.1 Known defects

The program has a number of known defects as follows:

1. Literals are always output in decimal notation. This implies that the existing system cannot be used to test a fixed point type having a value of `'Small` which has a recurring decimal value.

   This defect would be comparatively straightforward to remove by replacing literals by an expression. However, this would then imply that the compiler is not being checked on the handling of literals, which seems unsatisfactory.

   The based number notation could be used to handle a larger class of literals, including those that arise from minutes/hours, or minutes/degrees.

   Neither of these 'enhancements' have been made, since it appears that critical systems do not use non-binary values for `'Small`.

2. The logic used in the generator is to check for overflow before applying the rounding. In those cases in which `'Machine_Rounds` is true, or rounding to an integer, the complete argument range is not tested. For instance, suppose a fixed point type `FX` has as its upper limit the value 127.0 and `FX'Machine_Rounds` is true. Then `FX(Y)` is 127.0, when `Y` is a floating point value 127.375.

This enhancement has not been made since it would make the Prolog program significantly more complex. Also, if a compiler had a bug in this case alone, it would seem unreasonable to imply the compiler was unsuitable for critical applications.

3. For a floating point result, the possibility of exponent overflow or underflow is ignored. This situation could only arise if a fixed point type definition has a range near the underflow or overflow limits of a floating point type. This seems an absurd practice. If the situation did arise with overflow, the resulting test should fail to compile. With underflow, the resulting test would test for equality with zero using a literal which would underflow, which would be correct, if surprising.

   Hence it is concluded that this defect need not be removed.

4. Too many trivial test cases generated. Many tests are multiplication by zero or one, or of a nature that would make it very surprising if an error were detected. No attempt has been made to reject such tests, since the execution process is automatic, there seems little point in removing them.

5. The generator does not check the not equals operator, or the relational operators.

   The objective of the generator was to check the numeric operations, and the use of equality was necessary for this. The use of `in` was the convenient means of checking those numeric operations which do not have an exact answer.

   Hence the other operators were not within the specification of the generator, but could be included by a minor enhancement.

6. The Prolog program is not portable. It is thought that the program will function correctly on any implementation that supports unbounded integers in which integer division truncates to zero. (The program does not work correctly with SB Prolog Version 3.0, due to errors in the arithmetic relation operators, apart from not having unbounded integers.)

E The Ada test program

The resulting Ada test program is very straightforward. The program to include the generated Ada is listed below.

As the Prolog generator merely produces Boolean expressions which should be True, the Ada test program can encorporate these expressions in many ways. The example program here is the simplest that could be used, but has proved adequate.

The Boolean expressions output by the generator are converted into procedure calls by means of a small Perl script. The hand edits (to avoid the overflow of the largest negative value of some types) should also be handled by the Perl script.

Since problems were encountered by the generator producing very long lines, the Perl script splits each test into two lines. The additional comment gives the number of the test for diagnostic purposes.

```
   Single Ada program to execute generated tests

with Text_IO;
use Text_IO;
procedure Fixed_Tests is
 package My_Int is new Integer_IO(Integer);
 use My_Int;

 Case_Number   Integer := 0;

 -- insert types here
 -- All types must use just model numbers

 type Second2 is delta 0.01 range -327.68 .. 327.67
 for Second2'Small use 0.01;

Byte_Con : constant := 8;
Word_Con : constant := Byte_Con * 2;
D_Word_Con : constant := Byte_Con * 4;

type F16_3_T is delta 2.0**(-3) range -2.0**12    2.0**12 - 2.0**(-3);
  for F16_3_T'SIZE use Word_Con;

type F16_28_T is delta 2.0**(-28) range -2.0** -13) .  2.0**(-13) - 2.0**(-28);
  for F16_28_T'SIZE use Word_Con;

type F32_8_T is delta 2.0**(-8) range -2.0**23    2.0**23 - 2.0**(-8);
  for F32_8_T'SIZE use D_Word_Con;

type F32_33_T is delta 2.0**(-33) range -2.0**(-2)  . 2.0**(-2) - 2.0**(-33);
  for F32_33_T'SIZE use D_Word_Con;

    end inserted types

 - Now insert number declarations for 'First values that overflow

 T16 :  constant := -32768;
 T32 :  constant := -2147483648;
 MSec:  constant := -327.68;
 M163:  constant := -4096.0;
 M1628: constant := -0.0001220703125;
 M328:  constant := -8388608.0;
 M3233: constant := -0.25;

   Now insert Identity functions for types being tested
```

```
-- The functions are named <typename>_1 and <typename>_2.
-- A generic instantiation could be used here, but this was not
-- done since some Ada subsets exclude generics.

function F16_3_T_1(X  F16_3_T) return F16_3_T is
   begin
   return X;
   end;

function F16_3_T_2(X: F16_3_T) return F16_3_T is
   begin
   return X;
   end;

function F16_28_T_1(X: F16_28_T) return F16_28_T is
   begin
   return X;
   end;

function F16_28_T_2(X  F16_28_T) return F16_28_T is
   begin
   return X;
   end;

function Integer_1(X: Integer) return Integer is
   begin
   return X;
   end;

function Integer_2(X: Integer) return Integer is
   begin
   return X;
   end;

function Long_Integer_1(X  Long_Integer) return Long_Integer is
   begin
   return X;
   end;

function Long_Integer_2(X: Long_Integer) return Long_Integer is
   begin
   return X;
   end;

function Float_1(X: Float) return Float is
   begin
   return X;
   end;

function Float_2(X  Float) return Float is
   begin
   return X;
   end;

function Long_Float_1(X: Long_Float) return Long_Float is
   begin
   return X;
   end;
```

```
 function Long_Float_2(X: Long_Float) return Long_Float is
    begin
    return X;
    end;

 procedure Check(B: Boolean) is
    begin
    Case_Number := Case_Number + 1;
    if not B then
       Put("Test case failure: number ");
       Put(Case_Number);
       New_Line;
    end if;
    end Check;

begin
 -- tests go in here, a sample reproduced here
-- Start (v0.2) round
Check(Second2_1(Second2((-327.6775)))
     = Second2_2((MSec))); -- 1
Check(Second2_1(Second2((-0.0075)))
     = Second2_2((-0.01))); -- 2
Check(Second2_1(Second2(0.0025))
     = Second2_2(0.0)); -- 3

Check(F32_33_T(F32_33_T_1(0.00000000090221874415874481201171875))
     = F32_33_T_2(0.000000000931322574615478515625)); -- 49
Check(F32_33_T(F32_33_T_1(0.249999999854480847716331481933593 75))
     = F32_33_T_2(0.24999999988358467817306518554687 5)); -- 50
-- Finished round
-- Start (v0.2) addx
Check(Second2((MSec))+Second2_1(0.0)
     = Second2_2((MSec))); -- 51
Check(Second2((MSec))+Second2_1(0.01)
     = Second2_2((-327.67))); -- 52

-- Start (v0.2) mulpxx
Check(Float(Second2_1(Second2((MSec)))*Second2_1(Second2((MSec))))
     = 107374.18359375); -- 875
Check(Long_Float(Second2_1(Second2((MSec)))*Second2_1(Second2((MSec))))
     = 107374.182399999999786959961056709289550781 25); -- 876

-- Start (v0.2) divxxx
Check(F32_8_T(Second2_1(Second2((MSec)))/Second2_1(Second2(0.01)
     = (-32768.0)); -- 6651
Check(Second2(Second2_1(Second2((MSec)))/Second2_1(Second2(1.0))
     = (MSec)); -- 6652
Check(F16_3_T(Second2_1(Second2((MSec)))/Second2_1(Second2(1.0))
     in (-327.75) .. (-327.625)); -- 6653
Check(F32_8_T(Second2_1(Second2((MSec)))/Second2_1(Second2(1.0))
     in (-327.68359375) .. (-327.6796875)); -- 6654

 Put("All tests executed: "
 Put(Case_Number);
 Put(" tests");
 New_Line;
```

```
exception
 when Numeric_Error =>
       Put("Numeric_Error raised");
       New_Line;
       Put("Last case executed: number ");
       Put(Case_Number);
       New_Line;
 when Constraint_Error =>
       Put("Constraint_Error raised");
       New_Line;
       Put("Last case executed: number ");
       Put(Case_Number);
       New_Line;
 when others =>
       Put("Unexpected exception raised");
       New_Line;
       Put("Last case executed: number ");
       Put(Case_Number);
       New_Line;
end Fixed_Tests;
```