ISO/IEC JTC 1/SC 22/OWGV N 0149
Comments on the "Draft of the Fortran Annex of the OWG-V TR".
Date: 17 September 2008
Original Filename: owvg-cmts.txt
Contributed by: Nick Maclaren (via email)

I read this and was not happy about some aspects, so I circulated it to the BSI Fortran mailing list to get more expert opinions. I am passing on comments written by Malcolm Cohen, the ISO/IEC 1539-1 Editor, which cover my concerns and more.

Comments on the "Draft of the Fortran Annex of the OWG-V TR".

> This Annex provides Fortran-specific advice
> for the items in clause 6, specifically
> the 6.x.5 Avoiding the vulnerability or mitigating its effects
> subclause. This annex does not repeat the other
> sections of each vulnerability.
> Therefore, readers will find OWG-V n0138 helpful
> when reading this document. This draft is written
> with the draft Fortran 2008 standard in mind,

At the time of writing, this is only a draft CD (the CD registration ballot is still underway). It is more appropriate to use actual ISO standards not internal Working Drafts that might replace the standard in a year or two.

> %%% 6.1 BRS
> Leveraging Experience
>

This kind of management-speak should be avoided.

> Avoid the use of decremental features.
>

The so-called Obsolescent features are completely standardised and in some cases have no problems re safety.

> Avoid the use of processor extensions, including processor-defined
> intrinsic procedures.
>

This (and a number of other points) would be better covered by a simple requirement (for the program) to conform to the ISO Fortran standard.

> %%% 6.2 BQF
> Unspecified Behavior
>
> The Fortran term is undefined behavior (see also 6.4).

This is not, in fact, a Fortran term in the current ISO standard nor in the referenced document.

> Review subclause 1.3 and Annex A for a list of undefined behavior.

This is unhelpful in view of the fact that this is not a Fortran term, and Annex A consists mostly a list of "Implementation-defined Behavior".

> %%% 6.3 EWF
> Undefined Behavior
>
> Use only those forms and relationships where the Fortran standard

> provides an interpretation.

This is entirely covered by the requirement to conform to the standard: a program without a standard-provided interpretation is not conforming, as per the first sentence of subclause 1.5 of IS 1539-1(2004).

> %%% 6.4 FAB
> Implementation-defined Behavior
>
> The Fortran term is processor-dependent behavior (see also 6.4).
> Use only forms and relationships where the result of the program
> execution does not depend upon the processor choice from among
> the possibilities defined by the processor dependency.

This is completely unusable. The result of nearly every mathematical intrinsic function is a "processor-dependent approximation", as for that matter is normal floating-point arithmetic.

> Review Annex A for a list of processor dependencies.
>
> %%% 6.5 MEM
> Deprecated Language Features
>
> See Annex B of f08.
> Review Annex A for a list of deleted or obsolescent features.
>
> %%% 6.6 BVQ
> Unspecified Functionality
>
> No specific Fortran recommendation.
>
> %%% 6.7 NMP
> Pre-processor Directives
>
> No Fortran preprocessor.
>
> %%% 6.8 NAI
> Choice of Clear Names
>
> Avoid names differing only in case.

This is either inappropriate, inapplicable, or misleading, since the Fortran standard specifies that names are not case-sensitive.

> Use unqualified private statements to limit export of names
> from modules.
> Use only clauses on use statements to limit import of names
> from modules.

These conditions do not seem to have any connection to the topic
  "Choice of Clear Names".

Furthermore, this is missing the obvious Fortran-specific exhortation, which is
  "Don't use names that match language keywords or intrinsic procedures.".

> %%% 6.9 AJN
> Choice of Filenames and other External Identifiers
>
> Take care with external identifiers as they are all processor-dependent.

Note that according to the previous rule in 6.4, this means that filenames and other external identifiers should not be used (further emphasising the excessively stringent nature of 6.4).

> Note that the <scalar-default-char-literal> on an include line
> may not represent a file name, or the file name might be qualified
> by a processor-dependent search path.
>
> %%% 6.9+ YZS
> Unused Subprogram Argument
>
> Use compiler options to detect unused dummy arguments.
> Use explicit interfaces to ensure correct references.
> Make unused arguments optional and test for presence to trap
> references with the unused argument.

This does not make sense. Unused arguments are either there for a reason, in which case they must stay, or extraneous in which case they can be deleted. In neither case should they be "made optional". Indeed, changing them to be optional would be very likely to invalidate previously-correct code.

> Use argument intents to prevent inadvertent stores
> to the caller's storage.
>
> %%% 6.10 XYR
> Unused Variable
>
> Use implicit none to require explicit declaration of all variable names.
> Use translator options to generate warning when unused variables
> are encountered.

Inconsistent use of terminology "compiler" vs "translator" vs "processor"; one term should be chosen and used consistently throughout.

The Fortran standard uses the term "processor".

> Use a compiler option to warn of undeclared variables if use
> of implicit none is impractical.

This does not make much sense, since if "use of implicit none is impractical", then probably either one cannot change the source code (rendering the whole exercise pointless) or there is ubiquitous use of implicit typing (thus producing many "false positives" if the compiler option is used).

> Use unqualified private statements to limit export of names
> from modules.
> Use only clauses on use statements to limit import of names
> from modules.

These last two don't seem to have any relationship to "Unused Variable".

> %%% 6.11 YOW
> Identifier Name Reuse
>
> Do not use blocks to override names from other nested scopes.
> When using a block to declare a new name,
> specifically ensure that the name is unique across all
> enclosing scopes and enclosed scopes.
> Note that Fortran doesn't have a scope resolution operator.
> Use consistent names within groups of related entities.
> Ensure that all names are 63 characters or fewer.

The last requirement is an explicit requirement in the Fortran standard, and thus entirely covered by requiring conformance to the standard.

> %%% 6.11+ JIO

> Type Choice
>
> Use explicit kind type parameters to select the correct kind.
> Use the selected_?_kind() intrinsics to select the kind type values.
> Centralize an application's kind selections in a single module.
>
> %%% 6.12 IHN
> Type System
>
> Always use IMPLICIT NONE.
> Avoid implicit type conversions and mixed-mode arithmetic.
> Use compiler options to enable warnings where appropriate.
>
> %%% 6.13 STR
> Bit Representations
>
> Always use the Fortran bit model ordering.

That is in fact the only bit model available, so is trivially satisfied by all standard-conforming Fortran programs.

> Do not manipulate bits of negative numbers
> without documenting the treatment of negative numbers
> on the target platform.

There is no problem manipulating bits of "negative numbers" as long as you are not doing arithmetic with them as well - the results are completely standardised in terms of the bit model. What this ought to be prohibiting is indeed doing bit manipulation of the "sign bit" as well as doing arithmetic on the variable.

> %%% 6.14 PLF
> Floating-point Arithmetic
>
> Do not test for real equality.

Numeric analysis is a deep topic, and superficial rules like this not exclude correct and reliable programs, but fail to exclude most incorrect ones. Sometimes testing for floating-point equality is a mistake, but not always.

> Do not use comparisons
> where the difference between > and >= or < and <= are crucial.
> Compare real quantities within a tolerance
> rather than for strict equality.

Sometimes right, sometimes wrong.

> Do not use real loop counters.

There is no such thing as a "real loop counter". If the reference is to DO loops with floating-point index variables, this feature was deleted from Fortran more than a decade ago; thus it is covered by requiring conformance to the standard.

> Use the inquiry intrinsics to determine the characteristics
> of the real model rather than computing them.
> Use the intrinsics to get or set the fields of real data.

"fields of real data" does not seem to be a well-defined term.

> Avoid the use of bit constants as floating point constants.

Should be "Don't use BOZ literal constants as the argument to the intrinsic

functions CMPLX, DBLE or REAL.": these are the only contexts where this is
allowed by the standard.

> Do not assume knowledge of the rounding mode of the translator.
> Explicitly control the rounding mode of input/output transfers.
> Do not assume that register precision equals memory precision.

There is no concept of "register" in the Fortran standard.

> %%% 6.15 CCB
> Enumerator Issues
>
> Use enumerators only when interfacing to C programs.

This unnecessary restriction does not improve portability or reliability.

> Do not assume variables of enumeration type are the same
> as any kind of integer.

Perhaps this is just bad wording, but they are required to be the same as some
kind of integer! Again, this is relatively easy to use portably and reliably.

> %%% 6.16 FLC
> Numeric Conversion Errors
>
> Check the value of input data.
> Check values before critical operations.
> Set an error to occur when appropriate.
> Use explicit conversion intrinsics to indicate intention.
> Always use an explicit kind as part of a real or complex constant.
> Use a named constant with explicit kind for real or complex constants.

The last two rules are contradictory.

Using a named constant "zero" is not automatically an improvement over using
a literal constant such as "0._wp"; in particular using the named constant
means you have to go and look at its definition to see whether it is right.

> Be aware of the unexpected properties when using list directed input,
> or namelist input.
>
> %%% 6.17 CJM
> String Termination

Fortran strings are not terminated, they are counted.

> Use character operations rather than explicit loops.

This is very unclear: if you are operating on characters, all the operations
are character operations whether they are within loops or not.

> When explicit loops are necessary, use inquiry intrinsics
> to determine the loop count.
>
> %%% 6.18 XYX
> Boundary Beginning Violation
>
> Check values used as array indices.
> Use whole array operations when possible to avoid array index calculations.

Array operations do not need to be "whole" array operations to do that.

Furthermore, there are reliable coding methods which avoid boundary "violation"

when array index calculations are being done.

> Adjust extent lower bounds as indicated by the problem.

This appears to be slightly garbled; in any case, it is either unclear or incorrect.

> Use bounds checking to find out-of-bounds conditions.
>
> %%% 6.19 XYZ
> Unchecked Array Indexing
>
> Check values used as array indices.
> Use whole-array operations when possible to avoid array index calculations.

As mentioned above, this is unnecessary (these can be avoided in many ways).

Furthermore, avoiding array index calculations does not remove the possibility of bounds violations, so this requirement is not only unnecessary but also insufficient.

> Use inquiry intrinsics to determine array bounds.
>
> %%% 6.20 XYW
> Buffer Overflow in Stack
>
> Check values used as array indices.
> Use whole-array operations when possible to avoid array index calculations.
> Use inquiry intrinsics to determine array bounds.
> Use explicit ALLOCATE statements to place large object on the heap.

This last rule has nothing to do with "Buffer Overflow in Stack", but with "Stack Overflow", a horse of an entirely different colour.

> Indicate in code where implicit allocate/deallocate is expected.

Has even less to do with "Buffer Overflow" than anything else.

> Do not use superfluous "(:)" to indicate whole array operations.

That appears a personal preference of the author that does not belong in any International Standard or TR. Furthermore, it is contrary to established good practice in certain application fields. Finally, it has nothing whatsoever to do with "Buffer Overflow".

> %%% 6.21 XZB
> Buffer Overflow in Heap
>
> Indicate in code where implicit allocate or deallocate is expected.

This has nothing to do with Buffer Overflow.

> Check values used as array indices.
> Use whole-array operations when possible to avoid array index calculations.

As previously mentioned, array operations do not need to be whole array operations to avoid index calculations, and using whole array operations does not eliminate the possibility of buffer overflow.

> Use inquiry intrinsics to determine array bounds.
> Use explicit ALLOCATE statements to place large object on the heap.

This has nothing to do with Buffer Overflow.

(It is also ungrammatical.)

> Indicate in code where implicit allocate/deallocate is expected.

This has nothing to do with Buffer Overflow.

> Do not use superfluous "(:)" to indicate whole array operations.

This is bad advice that does not belong in an International Standard.

> %%% 6.22 HFC
> Pointer Casting and Pointer Type Changes
>
> Does not happen in Fortran.

This claim is incorrect. If the user does this he violates a requirement of the standard, but it is a runtime requirement that cannot always be detected at compile time.

In particular, using TYPE(C_PTR) and the procedure C_F_POINTER allows unsafe pointer conversion.

Furthermore, CLASS(*) allows unsafe pointer conversions for SEQUENCE types.

Which brings up the question: where is the recommendation not to use SEQUENCE types? (These are more error-prone than extensible types.)

> %%% 6.23 RVG
> Pointer Arithmetic
>
> Does not happen in Fortran.
>
> %%% 6.24 XYH
> Null Pointer Dereference
>
> Verify that pointers and allocatables are defined

That is totally impossible except by proving theorems about your program; this is not substantially easier in Fortran than in C.

> and are not null
> by use of the associated intrinsic or the allocated intrinsic.

In fact, undefined pointers are not permitted to be arguments to the ASSOCIATED intrinsic (this can only be used on defined pointers).

Furthermore, the ALLOCATED intrinsic tells about whether an allocatable is ALLOCATED, nothing about whether it is DEFINED.

> %%% 6.25 XYK
> Dangling Reference to Heap
>
> Use finalizers to clear pointers and allocatables
> in derived types.

This advice is nearly always counterproductive for reliable programming.

It is incorrect in the case of allocatable components, for these are already required by the standard to be automatically deallocated by the compiler.

The use of finalisers in Fortran, as in other languages, is frequently difficult to analyse, and they should only be used as a case of last resort when no other system suffices.

> Use automatic allocation/deallocation.

Good advice, but the earlier advice that requires indication of where it
is occurring will make a program that makes extensive use of this feature
hard to read.

> Put allocate and deallocate in the same module
> to make analysis tractable.

Partly contradicts "use automatic allocation/deallocation".

> %%% 6.26 SYM
> Templates and Generics
>
> Not applicable to Fortran.
>
> %%% 6.27 LAV
> Initialization of Variables
>
> Ensure all variables are initialized before use.
> Provide a default initialization for derived type variables.

This is usually unnecessary. Furthermore, there is nothing special about
derived types (vs intrinsic types) that means they absolutely require
initialisation whereas intrinsic types don't.

In fact, this makes use of derived types more computationally expensive,
especially when allocating large arrays (do you really want to sit there while
the processor churns through memory initialising your 8GB array?).

Thus, this is just bad advice. For specific cases it can be useful, but in
general it is unwarranted.

> Keep variable usage confined to one module
> to make analysis tractable.

This rule is suitable only for toy programs. In any case, general programming
advice like this is not Fortran-specific.

> Ensure that defined assignment assigns to all components.
> Use constructors rather than component-wise assignments.

This is only useful if you want to set all the values at once, otherwise
component-wise assignment is obviously better.

> Avoid use of COMMON.
> Do not rely upon variables being initialized to zero.
>
> %%% 6.28 XYY
> Wrap-around Error
>
> Check values for proper range between bit operations
> and arithmetic operations.
> Never use bit operations where an intrinsic procedure
> performs the intended calculation (for example,
> using and-with-sign-bit in place of sign()).

This would already be covered by the "don't mix bit manipulation and arithmetic
on negative numbers" rule (and is certainly already covered by the overly
strict "don't do bit manipulation on negative numbers" rule).

Furthermore, it appears to be attempting to prohibit a poor coding practice

that largely if not entirely died out decades ago.

> %%% 6.29 XZI
> Sign Extension Error
>

Fortran doesn't have unsigned integers, so surely this cannot arise.

> Use explicit conversion between differently sized integers.

Overly broad - there are very many cases where this would just degrade the readability (and thus maintainability) of the program.

> Do not use transfer().

Some uses of TRANSFER are perfectly safe. In any case, when TRANSFER is what you need to do, TRANSFER is what you should be using and not some other circumlocution or loophole.

> Do not use equivalence.
>
> %%% 6.30 JCW
> Operator Precedence/Order of Evaluation
>
> Use parenthesis where order of evaluation is critical.

Parentheses affect associativity (this is frequently not the same as the order of evaluation, though perhaps the main text makes this clearer).
Also, "parenthesis" should be plural "parentheses".

> Use auxiliary variables to order operations.
> Beware of high levels of optimization and isolate critical operations
> in separately-compiled procedures.
>
> %%% 6.31 SAM
> Side-effects and Order of Evaluation
>
> Use pure functions whenever possible.
> Use subroutines when side effects are needed.
> Declare argument intents to enable checks of argument usage.
>
> %%% 6.32 KOA
> Likely Incorrect Expression
>
> Simplify expressions.
> Use variables to hold partial results of long expressions.
> Vertically align similar terms in long expressions
> to ease visual checking.
>
> %%% 6.33 XYQ
> Dead and Deactivated Code
>
> Dead code or unreachable code should be removed.

Overly strict. It is fairly common (and good) practice to make use of dead code for portability (code that cannot be reached on one system might be reachable on another, possibly future, system).

> If available, use compiler options to warn of unreachable code.
>
> %%% 6.34 CLL
> Switch Statement and Static Analysis
>

> Ensure that select case constructs cover all cases.

Overly strict. The semantics of SELECT CASE are perfectly well defined when cases are deliberately omitted.

> Use a case default clause as needed, perhaps to provide notice
> of unexpected values.

Excessively application-specific.

> %%% 6.35 EOJ
> Demarcation of Flow Control
>
> Always use block forms of do-loops.
> Always use end-<block> statements.

This is required by the language syntax itself.

> Use named constructs for all but possibly the smallest constructs.
> Indent code.
> Use a syntax-highlighting editor.

These last two are not Fortran-specific.

> %%% 6.36 TEX
> Loop Control Variables
>
> Use the loop induction variable inside a loop

Use of terminology obscures the meaning here.

> (as opposed to an auxiliary variable),

And here.

> especially as an array index.
> Check that procedure references within a loop
> do not modify the loop variable.
> Ensure that procedures within a loop have argument intents
> for all arguments.
>
> %%% 6.37 XZH
> Off-by-one Error
>
> Use inquiry intrinsics to set loop limits.
> Use counted do-loops, do concurrent or forall for array computations.

These constructs do nothing to avoid off-by-one error. Array operations and masked array assignment do some to avoid it.

Furthermore, FORALL is widely recognised as the world's slowest "High Performance" feature, and its use should not be encouraged.

> Ensure that loop-local indexes have the same kind
> as variables of the same name outside the loop.

Inconsistent: 6.11 already forbids even having such variables.

> Use whole array operations if possible.

Inconsistent: we were just told to use DO and FORALL for array computations.

> Never use sentinel values for loop control,

Overly strict: there are well-recognised and safe procedures for using
sentinels when they are an appropriate solution to the problem.

> explicitly test for not associated when processing data structures
> involving pointers.

Overly prescriptive: circular data structures don't have null pointers.

Furthermore, either the algorithm already requires such testing, or the program
is in error if an (unexpected) null pointer is found. In the former case no
rule is needed, in the latter case there is frequently little different between
"STOP - IMPOSSIBLE NULL POINTER FOUND" and "Segmentation fault".

> %%% 6.38 EWD
> Structured Programming
>
> Use block structures in preference to go to statements.
> Name all structures where a cycle or exit statement is used and
> name the target structure of cycle and exit statements.

Unnecessarily prescriptive with no benefit.

> Do not use alternate returns,
> return a status variable and test its value instead.
> Do not use branch specifiers for input/output statements;
> use the status variable and test its value instead.

This is bad advice. Catching all input/output errors with ERR= in one place is
frequently far superior to the loss of readability that would result from
complicating a processing loop with incessant test-and-jump.

> %%% 6.39 CSJ
> Passing Parameters and Return Values
>
> Use argument intents.
> Minimize side effects in subroutines.

This does not make sense: the whole purpose of a SUBROUTINE is to cause a side
effect.

> Use pure or elemental functions wherever possible.
>
> %%% 6.39+ DTK
> Sentinel Values
>
> Do not rely upon sentinel values, which may be especially prevalent
> with procedure arguments and return values.
> Sentinel values spread between different modules may be harder
> to locate and remove.
> Use specific predicate procedures instead.

Overly prescriptive: a "specific predicate procedure" is frequently slower
than, and more opaque than, using a sentinel value.

> %%% 6.40 DCM
> Dangling References to Stack Frames
>
> Never associate a pointer having a greater longevity
> with a target having a shorter longevity.
> Use compiler options where available to warn of possible
> situations where the the pointer outlives the target.
>

> %%% 6.41 OTR
> Subprogram Signature Mismatch
>
> Always use explicit interfaces.

Insufficient. Explicit interfaces that are separately encoded still have the possibility of mismatch between the routine and the interface. The advice should be "not to use external procedures".

> Use optional arguments when needed.

Who could argue with this, or indeed with "use ANYTHING when needed".

However, it appears to have little to do with "signature mismatch".

> Use optional arguments when eliminating unused arguments from references.

This advice is useless and/or counterproductive in most cases.

> Use argument intents.
>
> %%% 6.41+ NSQ
> Library Signature
>
> Use compiler options or tools to generate explicit interfaces
> for libraries lacking them.
> Use a more recent edition of a library lacking explicit interfaces.
>
> %%% 6.41+ PUS
> Extra Intrinsics
>
> Give all application procedures an explicit interface,
> or the external attribute, to prevent confusion with intrinsics.
> Use the intrinsic attribute to specify intrinsic procedures.
>
> %%% 6.42 GDL
> Recursion
>
> Minimize the use of recursion.
> Convert recursive calculations to iterative calculations.
> Guarantee a recursive calculation has a maximum depth.

These rules are frequently inappropriate. They had good validity in the 1950s, but most computers have advanced since then.

> %%% 6.43 NZN
> Returning Error Status
>
> Treat errors locally if possible.
> Return a status variable, and always check its value.
>
> %%
% 6.44 RUE
> Termination Strategy
>
> Each application should have an error handling strategy.
> When an application is parallel, whether to kill the task,
> or halt the task but preserve its resources,
> or kill the application, must be decided and used consistently.
>
> %%% 6.45 AMV
> Type-breaking Reinterpretation of Data
>

> Do not use equivalence to reinterpret bit patterns.
> Do not use transfer.
> Do not use storage sequences unless necessary.
> When necessary, ensure consistent definition of common blocks.

No, the rule should be "Do not use common blocks.".

> %%% 6.46 XYL
> Memory Leak
>
> Ensure allocate statements match deallocate statements.
> Ensure that each deallocate statement is reached
> via all possible paths from the corresponding allocate statements.
> Use automatic deallocation where possible.
> Use allocatable variables rather than pointers.
> Put allocate and deallocate in the same module
> to make analysis tractable.
>
> %%% 6.47 TRJ
> Use of Libraries
>
> Use libraries with explicit interfaces (modules).
> If necessary, use compiler options to check interfaces
> where explicit interfaces are not present.
>
> %%% 6.48 NYY
> Dynamically-linked Code and Self-modifying Code
>
> Consider the use of static linking when useful.

This is outwith the scope of the Fortran standard and is not, in any case,
specific to Fortran.