ISO/IEC JTC 1/SC 22/OWGV N0040

Editor's draft 1 of PDTR 24772, 1 October 2006

ISO/IEC JTC 1/SC 22 N 0000

Date: 2006-10-01

ISO/IEC PDTR 24772

ISO/IEC JTC 1/SC 22/OWG

Secretariat: ANSI

Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

Élément introductif — Élément principal — Partie n: Titre de la partie

Document type: International standard
Document subtype: if applicable
Document stage: (20) Preparation
Document language: E

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772 which is a Technical Report of type 3, was prepared by Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 22, Programming Languages.

Introduction

A paragraph.

The **introduction** is an optional preliminary element used, if required, to give specific information or commentary about the technical content of the document, and about the reasons prompting its preparation. It shall not contain requirements.

The introduction shall not be numbered unless there is a need to create numbered subdivisions. In this case, it shall be numbered 0, with subclauses being numbered 0.1, 0.2, etc. Any numbered figure, table, displayed formula or footnote shall be numbered normally beginning with 1.

Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

1 Scope

1.1 In Scope

1) Applicable to the computer programming languages covered in this document.
2) Applicable to software written, reviewed and maintained for any application.
3) Applicable in any context where assured behavior is required, e.g. security, safety, mission/business criticality etc.

1.2 Not In Scope

This technical report does not address software engineering and management issues such as how to design and implement programs, using configuration management, managerial processes etc.

The specification of the application is *not* within the scope.

1.3 Approach

The impact of the guidelines in this technical report are likely to be highly leveraged in that they are likely to affect many times more people than the number that worked on them. This leverage means that these guidelines have the potential to make large savings, for a small cost, or to generate large unnecessary costs, for little benefit. For these reasons this technical report has taken a cautious approach to creating guideline recommendations. New guideline recommendations can be added over time, as practical experience and experimental evidence is accumulated.

Some of the reasons why a guideline might generate unnecessary costs include:

1) Little hard information is available on which guideline recommendations might be cost effective
2) It is likely to be difficult to withdraw a guideline recommendation once it has been published
3) Premature creation of a guideline recommendation can result in:
   i. Unnecessary enforcement coast (i.e., if a given recommendation is later found to be not worthwhile).
   ii. Potentially unnecessary program development costs through having to specify and use alternative constructs during software development.
   iii. A reduction in developer confidence of the worth of these guidelines.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC TR 15942:2000, "Information technology - Programming languages - Guide for the use of the Ada programming language in high integrity systems"

Joint Fighter Air Vehicle: C++ Coding Standards for the System Development and Demonstration Program. Lockheed Martin Corporation. December 2005.

ISO/IEC 9899:1999, *Programming Languages – C*

ISO/IEC 15291:1999, Information technology - Programming languages - Ada Semantic Interface Specification (ASIS)

Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B). December 1992.

IEC 61508: Parts 1-7, Functional safety: safety-related systems. 1998. (Part 3 is concerned with software).

ISO/IEC 15408: 1999 Information technology. Security techniques. Evaluation criteria for IT security.

J Barnes. High Integrity Software - the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2004 (second edition)[1].

[1] The first edition should not be used or quoted in this work.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1 Vulnerability

4. Symbols (and abbreviated terms)

5 Vulnerability issues

Vulnerabilities might be targeted by external threats such as worms and viruses, or might be faults that can occur during the expected normal execution of the software.

The economic impact of a vulnerability will depend on the how it changes the behavior of a program and the real world events that are affected by that program. For instance, the impact of a variable that is not initialized can range from failure to a coffee machine to deliver hot water to people dying in an aircraft accident.

The following subsections cover some of the sources of vulnerabilities.

5.1 Human factors

Possible human factors include the following:

- Cognitive failure, external pressures on readers and writers results in them failing to invest the time and effort needed to fully comprehend the code,

- Knowledge failure:
    - people reading source code having incomplete and incorrect knowledge of the appropriate language semantics,
    - people reading source code having incomplete and incorrect knowledge of how it will be executed by a particular implementation,
    - people reading source code having incomplete and incorrect knowledge of the interaction between its various components,

[Note: At London meeting it was decided to add the cost of obtaining the necessary knowledge]

- Competence.

5.2 Predictable execution

It is intended that this technical report identify issues that will enable a greater level of predictability to be achieved for the same level of investment of time and money. The following are some of the mechanisms used to achieve this goal:

- reducing the amount of cognitive effort that needs to be invested by readers of the source code,

- reducing the amount of knowledge needed by readers of the source code,

- reducing the probability that incorrect developer knowledge will result in incorrect prediction of behavior,

- recommending against the use of constructs that are costly or impractical to check automatically using tools,

- recommending against the use of constructs that are costly or impractical to check during testing,

- suggesting annotations which provide information against which additional consistency checks can be made,

- creating a widely adopted set of guidelines make it economically worthwhile to use checking tools, which in turn reduce the cost of achieving a desired level of confidence in predicted program behavior.

Verifying that the predicted behavior of a program is as intended (i.e., that it meets its specification) is outside the scope of this technical report.

5.2.1 Language definition

Languages frequently support constructs whose behaviour is undefined, implementation defined, or unspecified. If the output from a program has a dependency on these constructs having a particular behaviour, then the people and tools that reader the code need to be aware of, and take account of, this particular behaviour. In some cases the undefined and unspecified behaviours are likely to change frequently and it can be costly and timing consuming to continually have to track these changes and the impact they have on overall program behaviour.

Language constructs that are undefined, implementation defined, or unspecified need to be documented and the cost effectiveness of recommending against their use carried out.

5.3 Portability

Portability can refer to people or to tools. The skills people learn on one platform are likely to be the ones they apply, at least initially, to a different platform. The behavior of source code can change when it is built using different language translators and libraries (generating code for the same/different processor or same/different operating system).

Restricting the use of language constructs to those whose behavior does not vary between different translators and libraries increases the likelihood that a programs behavior will not change across platforms and that different people will correctly predict this behavior.

[Note: London 2006, this section should be rewritten – no words supplied]

5.4 Vulnerabilities Issues List

The following list has been taken from ISO/IEC 15942:2000 document, with slight wording changes to broaden the scope from an Ada programming language only list.

5.4.1 Strong typing vs. weak typing

The choice of storage used to support an algorithm is a trade-off between the possible underlying representations possible on the machine, the efficiency of access associated with those underlying representations, and the language/compiler/tool support available to support the choices made. Most languages choose a trade-off which maps one of a few fixed-size representations for integer-based types, real numbers, characters, booleans and other types.

On the other hand, the algorithm required usually has well-known properties for range, boundedness, and precision.

All digital programming language systems make compromises which can result in vulnerabilities.

If the usual range of the algorithm fits within a chosen representation size but exceptional processing may exceed that size, there is a risk that exceeding the size may cause truncation of results (usually known as wrap-around), the generation of an exception, or unexpected change of representation to a larger size. For HI systems, it is usually undesirable to dynamically determine if such situations can occur, so static analysis and choice of representation are used to ensure that this does not occur[2].

If the usual range of the algorithm fits always within the chosen storage, there is still a risk that some results will exceed the algorithm bounds and cause chaotic behaviour. Therefore HI systems should be able to state and determine the relative bounds of types used in calculations and ensure that these bounds are not exceeded, except possibly during expression evaluation before a final result is determined. For languages with strong type-checking, good algorithm design can support static determination of most (if not all) calculations as long as the correctness of the inputs to those calculations can be guaranteed. For languages with weak type checking, auxiliary tools and additional annotations can be used for static analysis of the algorithms, and explicit runtime checks can be used to support the dynamic verification of the algorithms.

Usually the bounds and operations of one type have no relation to those of another type, unless they are combined controlled ways. Some characteristics are obvious, such as never performing boolean operations on integers or integer operations on booleans. Others are less obvious such as adding centimeters and inches. Language systems that support the separation of such concepts will not require additional tooling or annotations to show the correctness of the implementation of the algorithm; language systems without strong typing will require external tools and extended analysis to verify the correct usage of objects.

When static checks are insufficient and runtime checking is required, weakly typed languages or strongly typed languages with runtime checking disabled will require visible checks of legal values to ensure correct operation of the algorithm.

For many algorithms, the range used by the representation chosen does not use the complete storage of the memory used. The excess memory is never used by the algorithm, and could be available to deliberate or accidental use to carry information. There is not much risk in ranged types since such information would affect range tests, but is possible for simple non-mathematical types or for composite types. This risk is non-existent for strongly typed languages since the unused portion is not addressable from within the algorithm and conversion between this type and types which could access these portions is illegal. For weakly typed languages, additional tooling or explicit checks that unused portions are always a known value (say null) would be required to prevent such a vulnerability.

5.4.2 Unbounded types

All objects are bounded. Simple objects such as integer types have word size or multi word sizes and rules about conversions between.

Facet: Static Analysis

5.4.3 Runtime support for typing

When support for the typing mechanism requires runtime artefacts, requires additional processing and reduced efficiency, makes static analysis less predictable.

[2] Note that such overflows could also occur during expression evaluation on a partial result even if the final result can be shown to be within bounds.

5.4.4 Arrays

Arrays consist of a set of storage for replicated data together with possibly a set of bounds for each dimension.

The major issues for language systems for arrays are as follows:

Static or dynamic bounds

In strongly typed systems, static bounds and invisibility of the explicit storage make arrays secure.

For strongly typed systems with dynamic bounds, the bounds are not directly accessible but attempts to exceed the bounds will result in exceptional processing.

In weakly typed systems, arrays which should be statically bounded can often be cast to other forms of access, and access outside the bounds is also possible. Tooling or explicit runtime checks are required to ensure that this does not occur.

In weakly typed systems, arrays which can be dynamically bounded will require explicit bounds to be maintained. These bounds can be changed by the application, resulting in inappropriate access to memory.

For some language systems, the access to the storage region containing the object can be manipulated in ways other than access through the base object and an index. For High Integrity systems, tools and static analysis is required to show that this does not happen.

5.4.5 Objects with variant structure

Most programming languages have ways of permitting a contiguous set of storage locations to be viewed in different ways at different times within the application. The most common application-visible way to accomplish this is the union (C/C++) or variant record (Ada, Pascal).

In weakly typed systems, or in unconstrained objects in strongly typed systems, the view of the object can be arbitrarily changed by the application, which may permit values in one view to be viewed or changed in a different view, and there may be gaps or portions of the object in one view which are not overwritten by writes to a different view.

Also, the size of such an object in one view may differ from other views, permitting possible hiding of data in an otherwise legal application.

In High Integrity systems, it is recommended that multiple views of the same object be forbidden.

5.4.6 Name overloading, operator overloading, overriding

Overloaded names helps preserve human cognitive space, if all items with the same name perform the same basic algorithm. Statically determinable overloaded names can be successfully evaluated by tools, but humans trying to evaluate calls to such overloaded subprograms (especially operators) may experience difficulty determining the correct call from all calls possible. Similar issues exist in languages that have a single name space but case sensitive names, as two names with the same spelling but different casing could be mistaken by humans.

In High Integrity systems, it is preferable to give unique names to entities or to use tools and likely annotations to show statically that the entities have the same behaviour.

5.4.7 Unbounded objects

Some languages can produce objects that have sizes which are non-static or even unbounded. This discussion does not include objects which are bounded but the language does not check bounds on every access.

Unbounded objects include objects with no embedded bounding mechanism and those with embedded bounding mechanisms. In either case, dynamic memory techniques are required to allocate the object and deallocation after a copy of an object may leave a valid reference to deallocated space.

Facet: Dynamic storage techniques

5.4.8 Constants

5.4.8.1 Ada Constants

Constants take the following forms in Ada:

- Any object declared a constant in the declarative part of a package or subprogram.
- Any "in" parameter of a subprogram (either explicitly declared `in` in a procedure or all parameters of a subprogram.
- Any `in` formal parameter of a generic.
- Any loop iteration variable.

Constants are initialized at the point of declaration.

Language rules prohibit the explicit assignment to constants, except as part of the constant declaration/creation process.

5.4.9 Uninitialized variables

The declaration and initialization of a variable can either occur in a single place or as two distinct steps. Issues for the initialization of objects:

- An object with an unknown value before its first use in an expression represents a serious vulnerability, with possible unbounded behaviour resulting from access to such objects before initialization.
- Initial values of variables should never be left to chance. Many systems `zero` global memory as the program is beginning, but applications must not rely on this since `zero` may not be a legal value and since any environmental change could result in non-zero values for variables, and objects declared on a stack or in other non-global areas are unlikely to be initialized.
- Where the object can be initialized as part of a declaration, this should be done. In languages such as Ada, there is a phase before subprogram execution commences (such as in elaboration phase or package body execution) where this elaboration can be done. In languages without this intermediate place, applications must determine where the first access in the complete program will occur and ensure that initialization occurs prior to that event (this may be a challenging computation).
- Some systems prefer initial illegal values be declared to support testing, but careful thought should be given to this approach as leaving this approach in operational systems could cause unplanned exceptional behaviour, or cause a substantial change between tested code and operational code.

5.4.10 Aliasing

Aliasing of a variable (access via multiple paths) makes it difficult to verify that the variable is being updated or accessed correctly. Aliasing can result from access to objects through access types (pointers), having local (via a parameter) and global view of an object, and making the same object an actual parameter for multiple parameters in a single call. Ada has copy-in/copy-out semantics for subprogram and entry parameters eliminates some problems associated with order of access, and the ability to construct and use compound objects as such parameters eliminates many access types in Ada. Applications must still show, however, that aliasing does not occur, or that it is correctly identified and handled if it does occur.

5.4.11 Nested subprograms

Some languages permit subprograms to be textually nested inside other subprograms. Such nesting makes test coverage almost impossible except in simple cases. Nested subprograms also have the property that local variables of one subprogram are visible from nested subprograms and may be accessed directly instead of being parameterized.

5.4.12 Expressions on objects of composite types

Some languages permit operations on objects which cause significant non-visible code to create, copy, compare. This could cause problems in timing analysis or in object code analysis.

On the other hand, operations on composites where the language does not support whole-object operations mean that each component of the object must be explicitly created, meaning that static analysis must be performed to show full coverage. This presents special challenges during maintenance when new components can be added.

5.4.13 Expressions on multiple conditions.

Potential problems with order of evaluation, unintended casting, short-circuit forms.

5.4.14 Object slices

A slice of an object is a part of it. When the target and the result of an operation target parts of the same object and those parts overlap, competing access to the same location may create errors. Such access will likely be problematic for static analysis tools.

Where slices are defined in a language, dynamic bounds to slices are problematic for static analysis tools.

5.4.15 `goto` Statement

Static analysis of code almost always assumes standard control constructs. Use of `goto` when using these tools causes code to be intractable for these kinds of analysis.

The usual place that `goto` is used in some languages is to escape from deeply nested control structures where an alternative construct is absent.

Languages with a good alternative construct there should be no need for use of the `goto` statement.

5.4.16 Loop statements

Loop statements include the loop controls mechanism and the loop start and end mechanisms. Simple loops with static control mechanisms and well-defined start and end mechanisms have almost no issues with any analysis mechanisms or cognitive issues.

For loops with static bounds, and where analysis can show that no modification of the loop control variable is possible are similarly analysable and safe. For a language such as Ada, language rules guarantee most loop properties, except that dynamic ranges for the loop control variable could make timing more difficult.

For languages where the control variable step function may be an arbitrary expression, static analysis of the loop control expression may be intractable.

For languages where the control variable termination function may be an arbitrary function or may be dynamic, static analysis of the loop control expression may be intractable, and combined with d), arbitrary loop increments and arbitrary termination expressions may cause non-terminating loops.

For languages where assignment to the control variable is permitted, static analysis of the loop control expressions may be intractable.

Recommend that HI systems only permit static expressions for loop start, increment and termination

Loops with embedded exit conditions usually protect the exit with some kind of conditional test. The placement of such an exit (including the `goto` statement) and the nature of the test may make timing analysis difficult.

5.4.17 Function side-effects

Functions which have only `in` variables and which update only local variables are side-effect free, safe, and amendable to static analysis. Functions with parameters that are access types or explicit `var` parameters[3] provide a vehicle for the program to update aliased objects through those parameters, and updates to non-local objects destroy the side-effect-free aspect of functions.

HI Applications should always document all input and output to all subprograms. For those subprograms where the access or update is through access parameters or through non-local objects, this must be documented through comments or non-programming mechanisms.

5.4.18 Order of Evaluation

A predictable order of evaluation is fundamental for showing correct behaviour of high integrity systems. We identify the following order of evaluation classifications and their issues.

[Note: Should we define these "in", "out" and "var" parameters in a more general way?]

5.4.18.1 Expression order of evaluation

Where the language specifies evaluation order in all cases, the application can depend upon that order; for those languages or situations where the order is not specified, applications must be written such that order of evaluation does not matter. In fact, it is recommended that expressions always be written such that the order of evaluation of expressions does not affect the correctness of the algorithm.

Explicit use of brackets to control evaluation order for complex expressions should be considered carefully. Too many levels of brackets cause as much confusion for the human reader as do too may expression terms. Reducing expression complexity by dividing them it multiple statements is often superior to heavy use of brackets.

5.4.18.2 Parameter order of evaluation

Where actual parameters of a subprogram contain expressions, if subprograms can have side effects, or for possibly aliased components, the order of evaluation of those parameters can affect the correctness of the execution of the subprogram. For languages with copy-in/copy-out semantics and specify parameter order for

[3] This is equivalent to a variable that is passed by reference in C

subprograms, avoiding access types (pointers), access parameters, and actual parameters which name the same object effectively eliminates evaluation order issues. For languages with pointer semantics for `out` parameters as well as cases where the actual parameter is an access type, applications must be written such that order of evaluation upon subprogram call or return is irrelevant to the correct operation of the subprogram.

5.4.18.3 Subprogram parameters – Aliasing

Some use local-copy/aliased actual-model, some use local-copy/ copy-in-copy-out/aliased-actual model. Use of aliased actual means that update of actual occurs immediately when the parameter is updated, and may leave actuals of subprogram inconsistent if exception or context switch occurs.

[Note: does "actuals" need defining?]

5.4.18.4 Subprogram parameter matching

Ada's subprogram parameters are intimate with the strong typing of the language: each call to a subprogram statically matches the type of each parameter with the specification of the subprogram, and the implementation must also statically match. In addition, Ada's named parameter calling convention helps eliminate mistakes when similar or overlapping types may be used in the same call, or when the order or number of parameters in a subprogram may change during maintenance.

For languages which are less permissive, tools must be used to guarantee that every subprogram call statically matches the specification of the subprogram, and that the implementation of the subprogram matches the specification (this includes verification of the type of each parameter, possibly the range of each parameter and the number of parameters). Where positional notation is the only way of creating a subprogram call and the types of the actuals of the call overlap, additional annotations may be useful to help static checking tools verify that the code matches the intent.

[Note: Same an above on "actuals", way to Ada oriented, needs to be more general.]

5.4.18.5 Aliasing of subprogram parameters

Special case of above issue, but aliasing of some object by 2 or more parameters is problematic.

[Note: Needs work]

5.4.19 Arithmetic Types

5.4.19.1 Integer Types

There are a number of issues for integer types. The only issues arising from Ada's Integer types occur in evaluating expressions that can result in the expected range being exceeded. In other languages, other issues must be addressed, such as silently exceeding the safe range of an object (usually tied to a word size) causing wraparound or an exception, silent promotion of an expression to an object of a different type,

For languages with weak type checking or in situations where it is necessary to statically determine if expression results and all partial expression results will be within the range of the target type or within the range of the base type of any partials

5.4.19.2 Silent type conversions

As a strongly-typed language, Ada does not permit silent conversion between any types except subtypes derived from the same base type. This typing effectively forbids the uncontrolled use or inappropriate pairing of types and operations that do not match the type. The exception for Ada is Modular Types which permit bit-wise Boolean operations on objects of these types.

More weakly typed languages can permit an object to be silently accessed as an object of a different type (e.g. performing Boolean operations on integers or characters). This lack of separation makes the static analysis of applications quite difficult.

5.4.19.3 Modular Types

Modular types have the traditional integer operations of integers, but have wrap-around semantics and permit bit-wise operations. Using any these operations prevents reasoning about order or the range of any object of these types. HI programs that use these operations in expressions with objects of these types must resort to dynamic checks of the final result for correct ranges when booleans are used and must dynamically verify that all input objects are within the correct ranges to prevent potential overflow before the expression is executed.

Languages with wraparound semantics on integer types and permit boolean or bitwise Boolean operations on integers have the same issues as Ada's modular types and must take the same precautions listed above for all integer operations. It is advisable that boolean operations on integer types be severely constrained to modules with appropriate analysis or banned completely.

5.4.19.4. Fixed Point Types

Fixed point types in Ada represent a way to perform integer-based arithmetic on real numbers. The default representation of such numbers is to use the closest binary representation of the smallest number representable. For example

```
   type One_Seventh is delta 1/7 range -100.0..100.0;
```

will represent 1/7 as 1(binary), 2/7 as 10(binary), 3/7 as 11(binary), 4/7 as 101(binary), and 1.0 as 1000(binary).

Another representation is available in which 1.0 would be represented as 111(binary). This representation provides exact arithmetic but care must be taken in conversion between numbers with different representations.

The use of such numbers lets programs perform real number calculations as scaled integers while hiding the explicit scaling and eliminates problems in floating point numbers for some types of calculations.

Other languages that do not provide such a type can create scaled integers and hide the details inside appropriate modules. If scaled integers are used, strategies to handle the issues raised above, as well as separating objects of this type from other integers will be required.

5.4.10.5 Floating Point Types

5.4.20 Low Level

5.4.20.1 Explicit Control of Low Level Mechanisms

Low Level routines are those designed to explicitly control aspects of the execution environment that support the running program, such as object size and layout, bit patterns associated with data, volatility or sharing of objects.

Strongly typed languages hide such details from the program and force explicit syntax to perform such access. For these languages, checking that such techniques are not used is almost trivial.

Weakly typed languages also have explicit mechanisms, but these mechanisms are almost regarded as part of the normal environment (for example pointer arithmetic or bitwise boolean operators). Such mechanisms effectively prevent static analysis of the program from being done, make any kind of reasoning analysis very difficult, and make the program non-portable.

While many HI programs have a few places where such low level mechanisms are required, it is fundamental that these places be restricted and bounded to those places where it is mandatory and banned from elsewhere. External tools will be required to ensure that rules are enforced, and places where they are used excluded from program static analysis.

5.4.20.2 Interfacing

[Note: Needs words]

5.4.21 Memory

5.4.21.1 Dynamic Memory

Dynamic memory is memory which is not assigned to any variable before the start of the main program, but which becomes assigned to an object at some point after, and possibly is disconnected from that variable at some later point and possibly connected to another variable later. There are two basic kinds of dynamic memory, stack and heap.

5.4.21.2 Stack Memory

Since stack memory is used to support the dynamic call chain and allocation of local storage, the major issue for HI programs is that one can statically show that stack usage is bounded and that the upper bound is less than the space allocated for the program stack. In a strongly typed language where allocated space depends upon static properties of the program, there exist static (though possibly computationally hard) algorithms to evaluate the stack requirements. In other cases, additional help such as formal annotations are probably required for this verification.

5.4.21.3 Heap Memory and Access Types (pointers)

Heap memory is problematic for HI programs. The first issue is that all such memory is accessed through pointers, and there is substantial risk that memory used in this way will be accessed by multiple objects (aliased). It is even possible that such memory will be returned by one pointer but still referenced by another.

5.4.21.4 Dynamic Memory Allocation

Memory that can be explicitly allocated and deallocated may be reallocated with some other base type, and if not completely initialized could be used to carry information covertly between program parts. It can also result in dangling access from uncleared pointers which now point to illegal objects.

5.4.21.5 Space Reclamation

Often the recovery of space does not match program unit termination, and it is hard to show that allocated memory is ever released. This can result in memory leaks and possibly exhaustion of memory.

5.4.21.6 Heap fragmentation.

Repeated allocation and deallocation of disparate types or memory amounts can lead to fragmented memory, resulting in failed allocations, even when there is enough total space, because insufficient contiguous space exists.

6 Guideline Selection Process

It is possible to claim that any language construct can be misunderstood by a developer and lead to a failure to predict program behavior. A cost/benefit analysis of each proposed guideline is the solution adopted by this technical report.

The selection process has been based on evidence that the use of a language construct leads to unintended behavior (i.e., a cost) and that the proposed guideline increases the likelihood that the behavior is as intended (i.e., a benefit). The following is a list of the major source of evidence on the use of a language construct and the faults resulting from that use:

- a list of language constructs having undefined, implementation defined, or unspecified behaviours,
- measurements of existing source code. This usage information has included the number of occurrences of uses of the construct and the contexts in which it occurs,
- measurement of faults experienced in existing code,
- measurements of developer knowledge and performance behaviour.

The following are some of the issues that were considered when framing guidelines:

- An attempt was made to be generic to particular kinds of language constructs (i.e., language independent), rather than being language specific.
- Preference was given to wording that is capable of being checked by automated tools.
- Known algorithms for performing various kinds of source code analysis and the properties of those algorithms (i.e., their complexity and running time).

6.1 Cost/Benefit Analysis

The fact that a coding construct is known to be a source of failure to predict correct behavior is not in itself a reason to recommend against its use. Unless the desired algorithmic functionality can be implemented using an alternative construct whose use has more predictable behavior, then there is no benefit in recommending against the use of the original construct.

While the cost/benefit of some guidelines may always come down in favor of them being adhered to (e.g., don't access a variable before it is given a value), the situation may be less clear cut for other guidelines. Providing a summary of the background analysis for each guideline will enable development groups.

Annex A provides a template for the information that should be supplied with each guideline.

It is unlikely that all of the guidelines given in this technical report will be applicable to all application domains.

6.2 Documenting of the selection process

The intended purpose of this documentation is to enable third parties to evaluate:

- the effectiveness of the process that created each guideline,
- the applicability of individual guidelines to a particular project.

7 Language Definition Issues

7.1 Execution Order

If the execution order is not defined, then a combinatorial problem can arise in attempting to predict the execution characteristics of a program.

7.2 Side-effects in functions

This could be regarded as a special case of the execution order problem, but from the point of view of program analysis, banning side-effects is best.

7.3 Permitted Optimizations

The C language introduces sequence points for this purpose, but causes some difficulties in establishing predictable execution.

7.4 Parameter Passing

Fortran introduced special wording, which very few people understood to allow some flexibility in this area.

Ada does something similar which can cause problems unless aliasing can be avoided. (In some situations, Ada structures can be passed by copy or reference.)

7.5 Aliasing

If an item of storage is accessible in more than one way, then the compiled code may depend upon how two different accesses are handled. Program proof has similar problems. Particularly troublesome with pointers.

7.6 Storage Control

This is handled automatically with Java (but then gives problems with timing). Ada has an unsafe feature for reclaiming storage and hence does not require garbage collection.

7.7 Exceptions

The method which makes predictable execution easier to verify is to require that predefined exceptions are not raised. Many situations in C which result in unpredictable execution would raise an exception in Ada. In consequence an Ada coding with no exceptions being raised can be very similar to the C coding with no unpredictable execution.

7.8 Tasking

This is a very difficult area and is not considers currently in this document.

8 Vulnerability Description

8.1 Vulnerability Description Outline

8.1.1 Generic description of the vulnerability

[Note: Depending on the overall organization of the document, this might occur at a level higher than the individual vulnerability description.]

8.1.2 Categorization

8.1.3 Language

[Note: This section will explain to which languages this description is applicable. Implementation dependency would also be discussed here.]

8.1.4 Cross-references to enumerations

The vulnerability should be cross-referenced with other enumerations and taxonomies whenever possible.

8.1.5 Specific description of vulnerability

Details to the generic description that is dependent upon the programming language is question.

8.1.6 Coding mechanisms for avoidance

Coding examples, including examples that have the vulnerability and examples that avoid the vulnerability should be included whenever possible. The description would consider the effectiveness of the various code work-arounds that are documented.

8.1.7 Coding mechanisms for avoidance

[Note: This section would provide coding examples, including examples that have the vulnerability and examples that avoid the vulnerability. The description would consider the effectiveness of the various code work-arounds that are offered.]

8.1.8 Analysis mechanisms for avoidance

[Note: For vulnerabilities that cannot be avoided by coding, and for those situations where a code-based solution is undesirable, this section discusses analysis techniques for avoiding the vulnerability. It would consider different types of analysis (perhaps drawing on the categories in TR 15942) and their effectiveness in finding and avoiding the vulnerability.]

8.1.9 Other mechanisms for mitigation

[Note: For vulnerabilities that cannot be avoided--by either coding or analysis--this section discusses other prospects for locating and mitigating the vulnerability. The text

might recommend specific review techniques or dynamic techniques (such as testing and simulation) to search for and mitigate vulnerabilities.]

8.1.10 Nature of risk in not treating

[Note: This section would describe the nature of the risk that must be accepted and the nature of the threats and/or hazards against which the software cannot be defended. The relationship to application security techniques might be discussed here.]

8.2 Writing Profiles

[Note: Advice for writing profiles was discussed in London 2006, no words]

Annex A
(informative)

Guideline Recommendation Factors

A.1 Factors that need to be covered in a proposed guideline recommendation

These are needed because circumstances might change, for instance:

- Changes to language definition.
- Changes to translator behavior.
- Developer training.
- More effective recommendation discovered.

A.1.1 Expected cost of following a guideline

How to evaluate likely costs.

A.1.2 Expected benefit from following a guideline

How to evaluate likely benefits.

A.2 Language definition

Which language definition to use. For instance, an ISO/IEC Standard, Industry standard, a particular implementation.

Position on use of extensions.

A.3 Measurements of language usage

Occurrences of applicable language constructs in software written for the target market.

How often do the constructs addressed by each guideline recommendation occur.

A.4 Level of expertise.

How much expertise, and in what areas, are the people using the language assumed to have?

Is use of the alternative constructs less likely to result in faults?

A.5 Intended purpose of guidelines

For instance: How the listed guidelines cover the requirements specified in a safety related standard.

A.6 Constructs whose behaviour can very

The different ways in which language definitions specify behaviour that is allowed to vary between implementations and how to go about documenting these cases.

A.7 Example guideline proposal template

A.7.1 Coding Guideline

Anticipated benefit of adhering to guideline

- Cost of moving to a new translator reduced.
- Probability of a fault introduced when new version of translator used reduced.
- Probability of developer making a mistake is reduced.
- Developer mistakes more likely to be detected during development.
- Reduction of future maintenance costs.

Annex B
(informative)

Bibliography

[1] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2001

[2] ISO/IEC TR 10000-1, *Information technology — Framework and taxonomy of International Standardized Profiles — Part 1: General principles and documentation framework*

[3] ISO 10241, *International terminology standards — Preparation and layout*