ISO/IEC/JTC 1/SC 22/WG 23/N0727
19 July 2017

Overarching cross-language standard needs

These are recommended standards for the language developers' community, standards that if developed could be of use to all languages such as the standards ISO/IEC/IEC 60559 Floating-Point arithmetic, ISO/IEC 10967-1:1994, Part 1: Integer and floating point arithmetic, and ISO/IEC 10967-2:2001, Part 2: Elementary numerical functions:

1. Standardized terminology for type systems
   a. Standardize on a common, uniform terminology to describe type systems so that programmers experienced in other languages can reliably learn the type system of a language that is new to them.
   b. Standardize on a common, uniform terminology to describe generics/templates so that programmers experienced in one language can reliably learn and refer to the type system of another language that has the same concept, but with a different name.
2. Standardized calling
   a. Standardize provisions for inter-language calling.
   b. Standardize on where parameter checks are done; that is, the receiving program does the parameter checks, not the calling program. *(this is one I added)*
   (This needs wording in Part 1 to substantiate.)
   (Deal with compilation and static analysis that eliminate the need for runtime checks)
3. Standardized fault handling
   a. Standardize the terminology and means to perform fault handling.
   b. Standardize a set of mechanisms for detecting and treating error conditions so that all languages to the extent possible could use them. This does not mean that all languages should use the same mechanisms as there should be a variety, but each of the mechanisms should be standardized.
   c. *(Fault tolerance and failure strategies has moved from 6.37 to 7.??). In order to justify such a treatment, it may need resurrection as a very visible clause 7 issue.)*

Top 11 list of what a language should have or do. These were extracted from guidance to language designers from clause 6.X.6 in TR 24772-1. Wording has been adjusted to provide a more general context, where applicable, with the expectation that the more generalized wording will be inserted back into the TR.

1. *Floating point should adhere to a recognized standard definition*
    a. A language should adhere to ISO/IEC/IEC 60559 Floating-Point arithmetic.
    b. A language should adhere to ISO/IEC 10967-1:1994, Part 1: Integer and floating point arithmetic, and ISO/IEC 10967-2:2001, Part 2: Elementary numerical functions.
2. *Conversions should be type-safe*
    a. A language should not allow unchecked casts or should make them immediately recognizable as being unsafe.
    b. A language should provide mechanisms to prevent programming errors due to conversions.
3. *Bounds checking should be mandatory*
    a. A language should perform automatic bounds checking on accesses to array elements, unless the compiler or static analysis can statically determine that the check is unnecessary.
4. *Whole* array operations *should be provided*
    a. A language should provide whole array operations, such as full array assignment and safe copying of arrays that may obviate the need to access individual elements.
5. *Subprograms, and in particular libraries, should have contracts for callers*
    a. Provide language mechanisms to formally specify preconditions and postconditions.
    b. Language-defined libraries should provide the preconditions and postconditions for each call so that function arguments can be validated during compilation, execution or via other static analysis tools. (*change in TR 24772-1 clause 6.46.5 to reflect this more general statement)*
    c. A language should specify means to describe the signatures of subprograms.
6. *Overflow errors should be detected and handled*
    a. Language should provide facilities to specify either an error, a saturated value, or a modulo result when numeric overflow occurs. Ideally, the selection among these alternatives could be made by the programmer.
7. *Undefined/unspecified/implementation defined behaviour should be minimized*
    a. A language should provide a list of undefined, unspecified and implementation-defined behaviours.
    b. A language should minimize the amount of unspecified and undefined behaviours, and minimize the number of possible behaviours for any construct with unspecified behaviour.
8. *Use of deprecated features should be diagnosed*
    a. A language should provide language mechanisms that optionally disable deprecated language features, in particular where deprecation for security or safety reasons. *(this one could be dropped in place of a more worthy "top 10" recommendation)*

9. *Synchronization among parallel/concurrent constructs should be supported*
    a. A language should create primitives that let applications specify regions of sequential access to data using mechanisms such as protected regions, Hoare monitors, or synchronous message passing between threads.
10. *Termination of for loops should be easier to guarantee*
    a. A language should add an identifier type for loop control that cannot be modified by anything other than the loop control construct. *(Add the notion of 1-time evaluation of the bounds) (consider in main document also)*

Complete list of ISO/IEC 24772-1 Implications for Standardization (this is what is in sections 6.x.6):

1. Language specifiers should standardize on a common, uniform terminology to describe their type systems so that programmers experienced in other languages can reliably learn the type system of a language that is new to them.
2. Provide a mechanism for selecting data types with sufficient capability for the problem at hand.
3. Provide a way for the computation to determine the limits of the data types actually selected.
4. Language implementers should consider providing compiler switches or other tools to provide the highest possible degree of checking for type errors.
5. For languages that are commonly used for bit manipulations, an *API* (Application Programming Interface) for bit manipulations that is independent of word size and machine instruction set should be defined and standardized.
6. Languages that do not already adhere to or only adhere to a subset of IEC 60559 [7] should consider adhering completely to the standard. Examples of standardization that should be considered: Languages should consider providing a means to generate diagnostics for code that attempts to test equality of two floating point values.
7. Languages should consider standardizing their data type to ISO/IEC 10967-1:1994 and ISO/IEC 10967-2:2001.
8. Languages that currently permit arithmetic and logical operations on enumeration types could provide a mechanism to ban such operations program-wide.
9. Languages that provide automatic defaults or that do not enforce static matching between enumerator definitions and initialization expressions could provide a mechanism to enforce such matching.
10. Languages should provide mechanisms to prevent programming errors due to conversions.
11. Languages should consider making all type-conversions explicit or at least generating warnings for implicit conversions where loss of data might occur.
12. Eliminating library calls that make assumptions about string termination characters. *(C Annex, SVP)*
13. Checking bounds when an array or string is accessed, see C Bounds Checking Library. *(C Annex, SVP)*
14. Specifying a string construct that does not need a string termination character. *(C Annex, SVP)*

15. Languages should provide safe copying of arrays as built-in operation.
16. Languages should consider only providing array copy routines in libraries that perform checks on the parameters to ensure that no buffer overrun can occur.
17. Languages should perform automatic bounds checking on accesses to array elements, unless the compiler can statically determine that the check is unnecessary. This capability may need to be optional for performance reasons. (Fix as in top 10, and in Part 1)
18. Languages that use pointer types should consider specifying a standardized feature for a pointer type that would enable array bounds checking. (*Remove in Part 1)*
19. Languages should consider providing compiler switches or other tools to check the size and bounds of arrays and their extents that are statically determinable.
20. Languages should consider providing whole array operations that may obviate the need to access individual elements.
21. Languages should consider the capability to generate exceptions or automatically extend the bounds of an array to accommodate accesses that might otherwise have been beyond the bounds.
22. Language-defined libraries should perform checks on the parameters to ensure that no buffer overrun can occur. *(make corresponding change in Part 1)*
23. Languages should consider providing full array assignment.
24. Languages should consider creating a mode that provides a runtime check of the validity of all accessed objects before the object is read, written or executed.
25. A language feature that would check a pointer value for `NULL` before performing an access should be considered.
    a. Implementations of the free function could tolerate multiple frees on the same reference/pointer or frees of memory that was never allocated.
    b. Language specifiers should design generics in such a way that any attempt to instantiate a generic with constructs that do not provide the required capabilities results in a compile-time error.
26. For properties that cannot be checked at compile time, language specifiers should provide an assertion mechanism for checking properties at run-time. It should be possible to inhibit assertion checking if efficiency is a concern.
    a. A storage allocation interface should be provided that will allow the called function to set the pointer used to NULL after the referenced storage is deallocated.
27. Language standards developers should consider providing facilities to specify either an error, a saturated value, or a modulo result when numeric overflow occurs. Ideally, the selection among these alternatives could be made by the programmer.
28. Languages should not provide logical shifting on arithmetic values or should consider flagging such usage for reviewers.
29. Languages that do not require declarations of names should consider providing an option that does impose that requirement.
30. Languages should consider providing optional warning messages for dead store.
31. Languages should consider requiring mandatory diagnostics for unused variables.

32. Languages should require mandatory diagnostics for variables with the same name in nested scopes.
33. Languages should require mandatory diagnostics for variable names that exceed the length that the implementation considers unique.
34. Languages should consider requiring mandatory diagnostics for overloading or overriding of keywords or standard library function identifiers.
35. Languages should not have preference rules among mutable namespaces. Ambiguities should be invalid and avoidable by the user, for example, by using names qualified by their originating namespace.
36. Some languages have ways to determine if modules and regions are elaborated and initialized and to raise exceptions if this does not occur. Languages that do not, could consider adding such capabilities.
37. Languages could consider setting aside fields in all objects to identify if initialization has occurred, especially for security and safety domains.
38. Languages that do not support whole-object initialization, could consider adding this capability.
39. Language definitions should avoid providing precedence or a particular associativity for operators that are not typically ordered with respect to one another in arithmetic, and instead require full parenthesization to avoid misinterpretation.
    a. In developing new or revised languages, give consideration to language features that will eliminate or mitigate this vulnerability, such as pure functions.
40. Languages should consider providing warnings for statements that are unlikely to be right such as statements without side effects. A null (no-op) statement may need to be added to the language for those rare instances where an intentional null statement is needed. Having a null statement as part of the language will reduce confusion as to why a statement with no side effects is present in the code.
41. Languages should consider not allowing assignments used as function parameters.
42. Languages should consider not allowing assignments within a Boolean expression.
43. Language definitions should avoid situations where easily confused symbols (such as `=` and `==`, or `;` and `:`, or `!=` and `/=`) are valid in the same context. For example, `=` is not generally valid in an `if` statement in Java because it does not normally return a Boolean value.
    a. Language specifications could require compilers to ensure that a complete set of alternatives is provided in cases where the value set of the switch variable can be statically determined.
44. Adding a mode that strictly enforces compound conditional and looping constructs with explicit termination, such as "`end if`" or a closing bracket.
45. Syntax for explicit termination of loops and conditional statements.
46. Features to terminate named loops and conditionals and determine if the structure as named matches the structure as inferred.
47. Language designers should consider the addition of an identifier type for loop control that cannot be modified by anything other than the loop control construct.
48. Languages should provide encapsulations for arrays that:
    a. Prevent the need for the developer to be concerned with explicit bounds values.

    b. Provide the developer with symbolic access to the array start, end and iterators.
49. Languages should support and favor structured programming through their constructs to the extent possible.
50. Programming language specifications could provide labels—such as `in`, `out`, and `inout`—that control the subprogram's access to its formal parameters, and enforce the access.
51. Do not provide means to obtain the address of a locally declared entity as a storable value; or
52. Define implicit checks to implement the assurance of enclosed lifetime expressed in sub-clause 5 of this vulnerability. Note that, in many cases, the check is statically decidable, for example, when the address of a local entity is taken as part of a return statement or expression.
53. Language specifiers could ensure that the signatures of subprograms match within a single compilation unit and could provide features for asserting and checking the match with externally compiled subprograms.
54. A standardized set of mechanisms for detecting and treating error conditions should be developed so that all languages to the extent possible could use them. This does not mean that all languages should use the same mechanisms as there should be a variety, but each of the mechanisms should be standardized.
55. Languages should consider providing a means to perform fault handling. Terminology and the means should be coordinated with other languages.
56. Because the ability to perform reinterpretation is sometimes necessary, but the need for it is rare, programming language designers might consider putting caution labels on operations that permit reinterpretation. For example, the operation in Ada that permits unconstrained reinterpretation is called `Unchecked_Conversion`.
57. Because of the difficulties with undiscriminated unions, programming language designers might consider offering union types that include distinct discriminants with appropriate enforcement of access to objects.
58. Provide means to create abstractions that guarantee deep copying where needed.
59. Languages can provide syntax and semantics to guarantee program-wide that dynamic memory is not used (such as the configuration `pragmas` feature offered by some programming languages).
60. Languages can document or specify that implementations must document choices for dynamic memory management algorithms, to hope designers decide on appropriate usage patterns and recovery techniques as necessary
61. Language specifiers should standardize on a common, uniform terminology to describe generics/templates so that programmers experienced in one language can reliably learn and refer to the type system of another language that has the same concept, but with a different name.
62. Language specifiers should design generics in such a way that any attempt to instantiate a generic with constructs that do not provide the required capabilities results in a compile-time error.
63. Language specifiers should provide an assertion mechanism for checking properties at run-time, for those properties that cannot be checked at compile time. It should be possible to inhibit assertion checking if efficiency is a concern.

64. Language specification should include the definition of a common versioning method.
65. Compilers should provide an option to report the class in which a resolved method resides.
66. Runtime environments should provide a trace of all runtime method resolutions.
67. Provide language mechanisms to formally specify preconditions and postconditions.
68. Find a solution to the problem. *(remove from Part 1)*
69. Do not allow unchecked casts.
70. Clearly state whether translators can extend the set of intrinsic procedures or not.
71. Clearly state what the precedence is for resolving collisions.
72. Clearly provide ways to mark a procedure signature as being the intrinsic or an application provided procedure.
73. Require that a diagnostic is issued when an application procedure matches the signature of an intrinsic procedure.
74. Ensure that all library functions defined operate as intended over the specified range of input values and react in a defined manner to values that are outside the specified range.
75. Languages should define libraries that provide the capability to validate parameters during compilation, during execution or by static analysis.
76. Develop standard provisions for inter-language calling with languages most often used with their programming language.
77. Provide a means so that a program can either automatically or manually check that the digital signature of a library matches the one in the compile/test environment.
78. Provide correct linkage even in the absence of correctly specified procedure signatures. (Note that this may be very difficult where the original source code is unavailable.) *(Look at Part 1 and rework)*
79. Provide specified means to describe the signatures of subprograms.
80. For languages that provide exceptions, provide a mechanism for catching all possible exceptions (for example, a 'catch-all' handler). The behaviour of the program when encountering an unhandled exception should be fully defined.
81. Provide a mechanism to determine which exceptions might be thrown by a called library routine.
82. Reduce or eliminate dependence on lexical-level pre-processors for essential functionality (such as conditional compilation).
83. Provide capabilities to inline functions and procedure calls, to reduce the need for pre-processor macros.
84. Language designers should consider removing or deprecating obscure, difficult to understand, or difficult to use features.
85. Language designers should provide language directives that optionally disable obscure language features.
86. Languages should minimize the amount of unspecified behaviours, minimize the number of possible behaviours for any given "unspecified" choice, and document what might be the difference in external effect associated with different choices.
87. Language designers should minimize the amount of undefined behaviour to the extent possible and practical.

88. Language designers should enumerate all the cases of undefined behaviour.
89. Language designers should provide mechanisms that permit the disabling or diagnosing of constructs that may produce undefined behaviour.
    a. Portability guidelines for a specific language should provide a list of common implementation-defined behaviours.
    b. Language specifiers should enumerate all the cases of implementation-defined behaviour.
90. Language designers should provide language directives that optionally disable obscure language features.
91. Obscure language features for which there are commonly used alternatives should be considered for removal from the language standard.
92. Obscure language features that have routinely been found to be the root cause of safety or security vulnerabilities, or that are routinely disallowed in software guidance documents should be considered for removal from the language standard.
93. Language designers should provide language mechanisms that optionally disable deprecated language features.
94. Consider including automatic synchronization of thread initiation as part of the concurrency model.
95. Provide a mechanism permitting query of activation success.
96. Provide a mechanism (either a language mechanism or a service call) to signal either another thread or an entity that can be queried by other threads when a thread terminates.
97. Languages that do not presently consider concurrency should consider creating primitives that let applications specify regions of sequential access to data. Mechanisms such as protected regions, Hoare monitors or synchronous message passing between threads result in significantly fewer resource access mistakes in a program.
98. Provide the possibility of selecting alternative concurrency models that support static analysis, such as one of the models that are known to have safe properties. For examples, see [9], [10], and [17].
99. Provide a mechanism to preclude the abort of a thread from another thread during critical pieces of code. Some languages (for example, Ada or Real-Time Java) provide a notion of an abort-deferred region.
100. Provide a mechanism to signal another thread (or an entity that can be queried by other threads) when a thread terminates.
101. Provide a mechanism that, within critical pieces of code, defers the delivery of asynchronous exceptions or asynchronous transfers of control.
102. Raise the level of abstraction for concurrency services.
103. Provide services or mechanisms to detect and recover from protocol lock failures.
104. Design concurrency services that help to avoid typical failures such as deadlock.
105. Ensure all format strings are verified to be correct in regard to the associated argument or parameter.

Complete list of ISO/IEC 24772-1 Implications for Standardization contents (these entries are identical to the ones in the previous section, and it also contains the complete text of 6.x.6 so that one can see where each entry originates):

6.2.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specifiers should standardize on a common, uniform terminology to describe their type systems so that programmers experienced in other languages can reliably learn the type system of a language that is new to them.
- Provide a mechanism for selecting data types with sufficient capability for the problem at hand.
- Provide a way for the computation to determine the limits of the data types actually selected.
- Language implementers should consider providing compiler switches or other tools to provide the highest possible degree of checking for type errors.

6.3.6 Implications for standardization

In future standardization activities, the following items should be considered:

- For languages that are commonly used for bit manipulations, an API (Application Programming Interface) for bit manipulations that is independent of word size and machine instruction set should be defined and standardized.

6.4.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages that do not already adhere to or only adhere to a subset of IEC 60559 [7] should consider adhering completely to the standard. Examples of standardization that should be considered:

- Languages should consider providing a means to generate diagnostics for code that attempts to test equality of two floating point values.
- Languages should consider standardizing their data type to ISO/IEC 10967-1:1994 and ISO/IEC 10967-2:2001.

6.5.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages that currently permit arithmetic and logical operations on enumeration types could provide a mechanism to ban such operations program-wide.

Languages that provide automatic defaults or that do not enforce static matching between enumerator definitions and initialization expressions could provide a mechanism to enforce such matching.

6.6.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should provide mechanisms to prevent programming errors due to conversions.
- Languages should consider making all type-conversions explicit or at least generating warnings for implicit conversions where loss of data might occur.

6.7.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Eliminating library calls that make assumptions about string termination characters.
- Checking bounds when an array or string is accessed, see C Bounds Checking Library[13].

Specifying a string construct that does not need a string termination character.

6.8.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should provide safe copying of arrays as built-in operation.
- Languages should consider only providing array copy routines in libraries that perform checks on the parameters to ensure that no buffer overrun can occur.
- Languages should perform automatic bounds checking on accesses to array elements, unless the compiler can statically determine that the check is unnecessary. This capability may need to be optional for performance reasons.
- Languages that use pointer types should consider specifying a standardized feature for a pointer type that would enable array bounds checking.

6.9.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing compiler switches or other tools to check the size and bounds of arrays and their extents that are statically determinable.
- Languages should consider providing whole array operations that may obviate the need to access individual elements.
- Languages should consider the capability to generate exceptions or automatically extend the bounds of an array to accommodate accesses that might otherwise have been beyond the bounds.

6.10.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider only providing libraries that perform checks on the parameters to ensure that no buffer overrun can occur.
- Languages should consider providing full array assignment.

6.11.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider creating a mode that provides a runtime check of the validity of all accessed objects before the object is read, written or executed.

6.12.6 Implications for standardization

- [None]

6.13.6 Implications for standardization

In future standardization activities, the following items should be considered:

- A language feature that would check a pointer value for NULL before performing an access should be considered.

6.14.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Implementations of the free function could tolerate multiple frees on the same reference/pointer or frees of memory that was never allocated.
- Language specifiers should design generics in such a way that any attempt to instantiate a generic with constructs that do not provide the required capabilities results in a compile-time error.
- For properties that cannot be checked at compile time, language specifiers should provide an assertion mechanism for checking properties at run-time. It should be possible to inhibit assertion checking if efficiency is a concern.
- A storage allocation interface should be provided that will allow the called function to set the pointer used to NULL after the referenced storage is deallocated.

6.15.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language standards developers should consider providing facilities to specify either an error, a saturated value, or a modulo result when numeric overflow occurs. Ideally, the selection among these alternatives could be made by the programmer.

6.16.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Not providing logical shifting on arithmetic values or flagging it for reviewers.

6.17.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages that do not require declarations of names should consider providing an option that does impose that requirement.

6.18.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing optional warning messages for dead store.

6.19.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider requiring mandatory diagnostics for unused variables.

6.20.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should require mandatory diagnostics for variables with the same name in nested scopes.

- Languages should require mandatory diagnostics for variable names that exceed the length that the implementation considers unique.

- Languages should consider requiring mandatory diagnostics for overloading or overriding of keywords or standard library function identifiers.

6.21.6 Implications for Standardization

In future standardization activities, the following items should be considered:

- Languages should not have preference rules among mutable namespaces. Ambiguities should be invalid and avoidable by the user, for example, by using names qualified by their originating namespace.

6.22.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Some languages have ways to determine if modules and regions are elaborated and initialized and to raise exceptions if this does not occur. Languages that do not, could consider adding such capabilities.

- Languages could consider setting aside fields in all objects to identify if initialization has occurred, especially for security and safety domains.

- Languages that do not support whole-object initialization, could consider adding this capability.

6.23.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language definitions should avoid providing precedence or a particular associativity for operators that are not typically ordered with respect to one another in arithmetic, and instead require full parenthesization to avoid misinterpretation.

6.24.6 Implications for standardization

In future standardization activities, the following items should be considered:

- In developing new or revised languages, give consideration to language features that will eliminate or mitigate this vulnerability, such as pure functions.

6.25.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing warnings for statements that are unlikely to be right such as statements without side effects. A null (no-op) statement may need to be added to the language for those rare instances where an intentional null statement is needed. Having a null statement as part of the language will reduce confusion as to why a statement with no side effects is present in the code.

- Languages should consider not allowing assignments used as function parameters.

- Languages should consider not allowing assignments within a Boolean expression.

- Language definitions should avoid situations where easily confused symbols (such as = and ==, or ; and :, or != and /=) are valid in the same context. For example, = is not generally valid in an if statement in Java because it does not normally return a Boolean value.

6.26.6 Implications for standardization

- [None]

6.27.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specifications could require compilers to ensure that a complete set of alternatives is provided in cases where the value set of the switch variable can be statically determined.

6.28.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Adding a mode that strictly enforces compound conditional and looping constructs with explicit termination, such as “end if” or a closing bracket.

- Syntax for explicit termination of loops and conditional statements.

- Features to terminate named loops and conditionals and determine if the structure as named matches the structure as inferred.

6.29.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language designers should consider the addition of an identifier type for loop control that cannot be modified by anything other than the loop control construct.

6.30.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should provide encapsulations for arrays that:

o Prevent the need for the developer to be concerned with explicit bounds values.

o Provide the developer with symbolic access to the array start, end and iterators.

6.31.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should support and favor structured programming through their constructs to the extent possible.

6.32.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Programming language specifications could provide labels—such as in, out, and inout—that control the subprogram's access to its formal parameters, and enforce the access.

6.33.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Do not provide means to obtain the address of a locally declared entity as a storable value; or

- Define implicit checks to implement the assurance of enclosed lifetime expressed in sub-clause 5 of this vulnerability. Note that, in many cases, the check is statically decidable, for example, when the address of a local entity is taken as part of a return statement or expression.

6.34.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specifiers could ensure that the signatures of subprograms match within a single compilation unit and could provide features for asserting and checking the match with externally compiled subprograms.

6.35.6 Implications for standardization

- [None]

6.36.6 Implications for standardization

In future standardization activities, the following items should be considered:

- A standardized set of mechanisms for detecting and treating error conditions should be developed so that all languages to the extent possible could use them. This does not mean that all languages should use the same mechanisms as there should be a variety, but each of the mechanisms should be standardized.

6.37.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing a means to perform fault handling. Terminology and the means should be coordinated with other languages.

6.38.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Because the ability to perform reinterpretation is sometimes necessary, but the need for it is rare, programming language designers might consider putting caution labels on operations that permit reinterpretation. For example, the operation in Ada that permits unconstrained reinterpretation is called Unchecked_Conversion.

- Because of the difficulties with undiscriminated unions, programming language designers might consider offering union types that include distinct discriminants with appropriate enforcement of access to objects.

6.39.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Provide means to create abstractions that guarantee deep copying where needed.

6.40.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages can provide syntax and semantics to guarantee program-wide that dynamic memory is not used (such as the configuration pragmas feature offered by some programming languages).

- Languages can document or specify that implementations must document choices for dynamic memory management algorithms, to hope designers decide on appropriate usage patterns and recovery techniques as necessary

6.41.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specifiers should standardize on a common, uniform terminology to describe generics/templates so that programmers experienced in one language can reliably learn and refer to the type system of another language that has the same concept, but with a different name.

- Language specifiers should design generics in such a way that any attempt to instantiate a generic with constructs that do not provide the required capabilities results in a compile-time error.

- Language specifiers should provide an assertion mechanism for checking properties at run-time, for those properties that cannot be checked at compile time. It should be possible to inhibit assertion checking if efficiency is a concern.

6.42.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specification should include the definition of a common versioning method.
- Compilers should provide an option to report the class in which a resolved method resides.
- Runtime environments should provide a trace of all runtime method resolutions.

6.43.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Provide language mechanisms to formally specify preconditions and postconditions.

6.44.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Find a solution to the problem.

6.45.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Do not allow unchecked casts.

6.46.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Clearly state whether translators can extend the set of intrinsic procedures or not.
- Clearly state what the precedence is for resolving collisions.
- Clearly provide ways to mark a procedure signature as being the intrinsic or an application provided procedure.
- Require that a diagnostic is issued when an application procedure matches the signature of an intrinsic procedure.

6.47.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Ensure that all library functions defined operate as intended over the specified range of input values and react in a defined manner to values that are outside the specified range.

- Languages should define libraries that provide the capability to validate parameters during compilation, during execution or by static analysis.

6.48.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Develop standard provisions for inter-language calling with languages most often used with their programming language.

6.49.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Provide a means so that a program can either automatically or manually check that the digital signature of a library matches the one in the compile/test environment

6.50.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Provide correct linkage even in the absence of correctly specified procedure signatures. (Note that this may be very difficult where the original source code is unavailable.)

- Provide specified means to describe the signatures of subprograms.

6.51.6 Implications for standardization

In future standardization activities, the following items should be considered:

- For languages that provide exceptions, provide a mechanism for catching all possible exceptions (for example, a 'catch-all' handler). The behaviour of the program when encountering an unhandled exception should be fully defined.

- Provide a mechanism to determine which exceptions might be thrown by a called library routine.

6.52.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Reduce or eliminate dependence on lexical-level pre-processors for essential functionality (such as conditional compilation).

- Provide capabilities to inline functions and procedure calls, to reduce the need for pre-processor macros.

6.53.6 Implications for standardization

[None]

6.54.6 Implications for standardization

[None]

6.55.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language designers should consider removing or deprecating obscure, difficult to understand, or difficult to use features.
- Language designers should provide language directives that optionally disable obscure language features.

6.56.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should minimize the amount of unspecified behaviours, minimize the number of possible behaviours for any given "unspecified" choice, and document what might be the difference in external effect associated with different choices.

6.57.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language designers should minimize the amount of undefined behaviour to the extent possible and practical.
- Language designers should enumerate all the cases of undefined behaviour.
- Language designers should provide mechanisms that permit the disabling or diagnosing of constructs that may produce undefined behaviour.

6.58.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Portability guidelines for a specific language should provide a list of common implementation-defined behaviours.
- Language specifiers should enumerate all the cases of implementation-defined behaviour.
- Language designers should provide language directives that optionally disable obscure language features.

6.59.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Obscure language features for which there are commonly used alternatives should be considered for removal from the language standard.
- Obscure language features that have routinely been found to be the root cause of safety or security vulnerabilities, or that are routinely disallowed in software guidance documents should be considered for removal from the language standard.

• Language designers should provide language mechanisms that optionally disable deprecated language features.

6.60.6 Implications for standardization

In future standardization activities, the following items should be considered:

• Consider including automatic synchronization of thread initiation as part of the concurrency model.

• Provide a mechanism permitting query of activation success.

6.61.6 Implications for standardization

In future standardization activities, the following items should be considered:

• Provide a mechanism (either a language mechanism or a service call) to signal either another thread or an entity that can be queried by other threads when a thread terminates.

6.62.6 Implications for standardization

In future standardisation activities, the following items should be considered:

• Languages that do not presently consider concurrency should consider creating primitives that let applications specify regions of sequential access to data. Mechanisms such as protected regions, Hoare monitors or synchronous message passing between threads result in significantly fewer resource access mistakes in a program.

Provide the possibility of selecting alternative concurrency models that support static analysis, such as one of the models that are known to have safe properties. For examples, see [9], [10], and [17].

6.63.6 Implications for standardization

In future standardization activities, the following items should be considered:

• Provide a mechanism to preclude the abort of a thread from another thread during critical pieces of code. Some languages (for example, Ada or Real-Time Java) provide a notion of an abort-deferred region.

• Provide a mechanism to signal another thread (or an entity that can be queried by other threads) when a thread terminates.

• Provide a mechanism that, within critical pieces of code, defers the delivery of asynchronous exceptions or asynchronous transfers of control.

6.64.6 Implications for standardization

In future standardization activities, the following items should be considered:

• Raise the level of abstraction for concurrency services.

• Provide services or mechanisms to detect and recover from protocol lock failures.

• Design concurrency services that help to avoid typical failures such as deadlock.

6.65.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Ensure all format strings are verified to be correct in regard to the associated argument or parameter.