cnes	**CNES** **STANDARDS REFERENCE** RNC

Reference: RNC-CNES-Q-HB-80-501
Version 4
17 September 2009

MANUAL

PRODUCT ASSURANCE

COMMON CODING RULES FOR PROGRAMMING LANGUAGES

APPROVAL of Standardisation Office	**BN no. 39 dated 25/02/08 – BN no. 44 dated 08/09/08** **BN no. 54 dated 16/09/09**

DOCUMENT ANALYSIS PAGE

TITLE: COMMON CODING RULES FOR PROGRAMMING LANGUAGES	
KEYWORDS: Common rule Generic Programming language	
EQUIVALENT STANDARD: None	
REMARKS: None	
ABSTRACT: This document sets out the common rules for using programming languages.	
DOCUMENT STATUS: This document is part of the collection of approved Manuals in the CNES Standards Reference. This document is affiliated to document RNC-ECSS-ST-Q-80 "Software Product Assurance".	
NUMBER OF PAGES: 83	**Language**: English (translated from the original French)
SOFTWARE PACKAGES USED / VERSION: Word 2007	
MANAGING DEPARTMENT: General Inspectorate and Quality Directorate (IGQ)	
AUTHOR(S): Jean-Charles DAMERY	**DATE: 17/09/09**

MODIFICATION REVISION SHEET

VERSION	DATE	PAGES MODIFIED	REMARKS
1	10/12/2006	Creation	Creation with the support of T. Leydier (Virtual Reality). See “FEB 48/2006” accepted in BN no. 22 dated 06/03/06. Document accepted in BN no. 34 dated 25/06/07 for document to be introduced in the RNC.
2	10/04/2008	Page 74 § 8.1	Following “FEB 77/2008” accepted in BN no. 39 dated 25/02/2008, correction of a minor error in the summary table of rules.
3	02/06/2008	All	Change of nomenclature following the ECSS benchmarking stage (former reference "RNC-CNES-Q-80-501").
4	17/09/2009		Following “FEB 91/2009” accepted in BN no. 54 dated 16/09/09 introducing the new manual “RNC-CNES-Q-HB-80-535” in the RNC, the tailoring tool in document “RNC-CNES-Q-HB-80-501” is updated.

TABLE OF CONTENTS

1. INTRODUCTION

The document "Common Coding Rules for Using Programming Languages" is affiliated with the document RNC-ECSS-Q-ST-80 "Software Product Assurance". It describes applicable rules for the programming languages used by CNES.

2. PURPOSE

The aim of this document is to establish common rules for programming languages. These rules have been established based on the “state of the art” and the “lessons learned” accumulated over the projects. This document is essential when using a programming language for a CNES project. It is supplemented by documents that are specific to each language.

3. SCOPE

This document applies to all CNES projects.

It must be adapted and/or completed by the Project Manager and/or Quality Engineer as regards code organisation, identifier nomenclature and any other specific rules according to established quality objectives that may be defined using the RD1 document.

This document is never used alone, but rather is used in conjunction with the language document; for example, for a JAVA project, common rules will be combined with the rules presented in the JAVA manual. This combination and rule selection will be performed at project outset using a tailoring tool.

The tailoring tool is an interface that allows the appropriate common rules and "language" to be selected for each project, according to the project's criteria (maintainability, criticality, test effort). It is included in this document; it may also be activated by clicking on the button below:

Lancer l'outil de tailorisation

Remark: The language manuals used by the tailoring tool must be in the current directory.

4. DOCUMENTS

4.1. REFERENCE DOCUMENTS

RD	Identification	Title
(RD1)	RNC-ECSS-Q-ST-80	Software Product Assurance

4.2. APPLICABLE DOCUMENTS

AD	Identification	Title
None		

5. TERMINOLOGY

5.1. GLOSSARY

Term	Definition
Library	A group of functions or procedures with a common theme.
Function	An operation that provides a result. A function does not generally modify the value of its parameters.
Module	A programming unit that groups together data and operations. Modules are generally associated with a code file.
Operation	A processing unit (a software procedure or function that performs processing).
Scope	The programming area in which data may be used: when the scope is local, data may be used locally, in a function, for example; when it is global, the data may be used anywhere in the code.
Procedure	An operation that does not produce a result, and which groups together instructions. Unlike functions, procedures may modify the value of their parameters.
Main program	An operation that starts program execution.
Sub-program	A function or sub-program.
Task	A control flow managed at operating system level. A synonym of a process.
Tailoring	A selection of rules applicable to the project, which may involve adaptation or the addition of new rules.
Thread	A control flow that is lighter than a task and managed at language level. By "light", we mean: - memory space is shared between the threads of a process. - changing thread context from one to another is faster than changing process context from one to another.

5.2. ABBREVIATIONS

5.2.1. Rule coding

Each rule is presented in a table containing 4 items of information:

<Identification>	<rule title>
<Tailorisation>	
<Type de Projet>	

This table features the following fields:

- Rule identification as a <sub-chapter>.<rule code>; the sub-chapter is standard and is coded using a standard mnemonic. The following coding is used for sub-chapters:
  Org: Code organisation
  Pre: Code presentation
  Id: Identifiers
  Data: Data
  Pro: Processing
  Err: Error management
  Dyn: Dynamic
  Int: Interfaces
  QA: Quality
  OR: Other rules
- Rule title: this is the rule label.
- Tailoring: this is a field for defining 4 quantitative tailoring parameters. These parameters are identified by a letter and are presented as follows:
  - M=<m>;R=<r>;P=<p>;V=<v>
  - With
    - m: a maintainability score from 0 to 3 (0 = the rule has little impact on maintainability, 3 = the rule significantly impacts maintainability)
    - r: a reliability score from 0 to 3 (0 = the rule has little impact on reliability, 3 = the rule significantly impacts reliability)
    - p: relative priority from 1 to 200; this information allows rules to be classified (1 = the most important rule, 200 = the least important rule)
    - v: verifiability score from 0 to 2 (2 = rule is easily verified (usually automatically using an analyzer), 1 = rule is not easily verified (rule compliance may generally be assessed by combining manual actions and analyzer results), 0 = rule can not be verified).

  These scores have been gathered from the lessons learned by the authors of this document. They may change in accordance with forthcoming lessons learned.
- Project type: this is another tailoring parameter; it defines the project category concerned by the rule. It must be selected from among the following values: On-board, Ground, Any.

The rule is completed by 3 mandatory paragraphs:

- a description,
- a justification,
- examples.

<u>Example of a rule:</u>

id.NomDonnee	The name of a datum must be a common name taken from everyday language; the plural form must be used if the datum is a set or group.
M=3;R=0;P=43;V=0	
Any	

Description

Not Applicable

Justification

Enhances readability.

Example

In C++

```
TheStarTracker
TheResults
TheDaysOfTheWeek
```

5.2.2. Other abbreviations or acronyms

Term	Definition
RNC	CNES Standard Reference

6. COMPLIANCE WITH RELEVANT SPECIFICATIONS AND STANDARDS

Not Applicable

7. RULES

7.1. CODE DESIGN / ORGANISATION

Org.DonneesOper	Data and operations must be grouped together in modules to form consistent packages, by using the available resources of the language.
M=3;R=0;P=41;V=0	
Any	

Description

This rule concerns all resources and all conceptual levels proposed by the language used, in regard to modularity.

Justification

Reinforces the priority and precedence of design activities over coding activities.
Ensures consistency between software design and code.

Example

In SHELL
This rule concerns scripts.
In ADA
This rule concerns units, packages and libraries.
In C++
This rule concerns classes, files and namespaces.
In JAVA
This rule concerns classes, files and packages.
In FORTRAN
An example of grouping data to manage a valve:

```
module VALVES
   ! ========= defining a valve =================
   integer, parameter :: StatusOpen = 1, &
                         StatusClosed  = 2, &
                         StatusTransitional = 3
   type VALVE
      integer :: ident
      integer :: status
      real(DOUBLE) :: flowrate
      integer :: upstream, downstream
   end type VALVE
   ! ========= global valve table, by ident ====
   integer, parameter :: MaxNoValves = 1000
   type(VALVE), dimension(MaxNoValves) :: VALVE_TABLES
   ! ========= valve file =====================
   character(LEN=*), parameter :: ValvesFile = 'valves.dat'
   integer, parameter :: ValvesChannel = 17
   ...
end module VALVES
```

Org.ModuleNom	A module name must convey the conceptual unit that the module represents
M=1;R=0;P=105;V=0	
Any	

Description

This rule concerns all types of conceivable modules, according to the language concerned. It also concerns associated files, their location, name and file extension. It is a logical consequence of the rule Org.DonneesOper .
The rule must be adapted to production environment constraints, such as: The file management system, the use of a code generator or compiler constraints.
Correspondence rules between "design units" and "support source file" must also be defined.

Justification

Enhances source readability.

Example

In ADA
A file name uses the name of the Ada compiler unit that it contains.
If it is a separate unit, the file name has the same prefix as the parent unit.
A file name that contains a package (resp. a body) specification has a suffix of _s (resp. _b) or has the extension .ads (resp. .adb).
In SHELL
Scripts have a descriptive name that will use the processing name plus the extension '.sh'.
In IDL
The suffix ".pro" must be used for IDL source files.
Define a suffix for batch files (for example, ".inc")

Org.Couplage	Linking between modules must be minimised: use links between modules must be uni-directional and be fewer than a limit set for the project.
M=3;R=1;P=32;V=1	
Any	

Description

Dependence between modules must be ordered and limited. Circular links are prohibited. The number of external variables (common to several compilation units) must be limited. References between modules performed using instructions such as "use" or "include" must be ordered and limited.

Justification

Significant linking complicates maintenance: changes made to a module may require changes to all dependent modules, and will, at best, require a regression search to be performed for these modules.

Example

In ADA
Context clauses (***with*** clause) in specifications for a package and/or its body define the entities needed by the specification and/or the package body. These clauses must not be used unless it is strictly necessary.

In C and C++

Global "include" should be avoided; only truly useful files should be included. A limit should be created for the number of files included and the level of inclusion.

In JAVA

"Generic" imports (that use *) should be avoided.

In FORTRAN, PVWAWE and IDL

The use of commons should be limited.

Org.Masquage	Data usage links should be avoided: read- and write-access operations should be used instead (information masking and data encapsulation principle), when this principle is not overly prejudicial for the language used.
M=2;R=1;P=44;V=1	
Any	

Description

The only data that can be directly accessed are constants.
In case a significant optimization of the execution time is needed, the rule may be waived: direct access to member data is quicker than a function, particularly when the language concerned does not support inline functions.

Justification

References to member data are uniform in the whole user code because functional notation must be used. The mode of access to member data may be controlled, thereby facilitating maintenance and updating: for example, all updates for a given piece of data may be traced via its write-access method.

Example

In C++

Member data will be declared "private" and access operations will be defined:

```
Changing implementation of a class which is transparent for users:
// File "Person.h"
class Person {
public: // Read access
     const Date& BirthDate();
     int age();
private:
     Date BirthDate_;
     int age_; // Data derived from BirthDate_
};
#include "Person.I"
// File "Person.I"
inline Person::BirthDate()  { return BirthDate_; }
inline Person::age()              { return age_; }
```

A second implementation is defined afterwards to minimise occupied memory space, even if this adversely impacts performance: the member data "*age_*" is deleted and age is calculated in the "*age*()" method using "*BirthDate_*".
This change of implementation is transparent for user classes because the interface for the Person class is unchanged.

In ADA

Package variables must be manipulated using only the primitives provided in the package specification. The variables themselves are declared in the package bodies and never in the specification.

In FORTRAN 90

Only the named constants will have PUBLIC visibility.

Org.Module	The code lay-out of each module must be standardised for the project.
M=1;R=0;P=104;V=1	
Any	

Description

Code lay-out concerns general aspects of the modules: compilation units and files, data declaration and the declaration of procedures, functions and other services.

Justification

Common code lay-out facilitates maintainability.

Example

In PVWAWE

A standard code lay-out for services and command files should be defined. For example, each service must contain:

- a header:
    - the name of the service,
    - the version,
    - the author,
    - the creation date,
    - a description,
    - a list of services used,
    - the call mode, as well as a description of parameters,
    - the COMMONs used,
    - a list of local variables,
    - the service's algorithm.
- the service body:
    - the inclusion of files,
    - the initialisation of return parameters,
    - the declaration (initialisation) of local variables,
    - a presence and validity test for optional variables,
    - processing,
    - error labels,
    - an end label.

In C++

It is recommended that public constructors and destructors be declared first.

A useful rule involves first establishing method categories and grouping the methods for a class interface according to these categories (constructor, destructor, access, status, etc.). Each category is introduced by a comment, which gives its name after the keyword *public* (which may be repeated more than once in C++). Method categories always appear in the same order in all classes.

Org.MultiLang	When more than one programming language are used for a project, correspondence rules must be defined for the elements exchanged between the languages.
M=1;R=1;P=58;V=0	
Any	

Description

The same identifiers should be used in each language, wherever possible. Case should also be respected (upper/lowercase). At the same time, when two neighbouring languages are mixed, thereby potentially creating confusion, mixing should be limited, and a rule should be defined to differentiate the two languages when they coexist.

Justification

This rule facilitates application maintenance and readability.

Example

In PVWAWE

Use the same variable names with the C/Fortran and WAVE programming languages.

In C and C++

C and C++ do not use the same mechanisms for passing parameters. If a C function is called in C++, its belonging to the C language must be highlighted in the identification of the function, and vice versa.

Org.Duplication	Code duplication must be avoided by intelligently using the techniques available at language level (passing parameters, using abstract operations, using metalanguages).
M=3;R=1;P=24;V=1	
Any	

Description

Each language proposes techniques for avoiding duplication: these techniques should be studied on a case-by-case basis, and the most appropriate technique selected. It is up to the programmer to choose the technique, but this is often a high-level decision that may be traced back to the design stage. In addition, this choice must account for the fact that excessive abstraction may adversely impact program maintainability. Consequently, in certain cases, parameterisation is preferable to generic programming.

Justification

Duplication must be avoided as it generates extra costs and a high risk for inconsistency in maintenance. The various techniques proposed by the languages are not equivalent: choosing an inappropriate technique may produce code that is not very readable or efficient.

Example

In C, C++ and ADA:

Some "short" functions may generate more instructions in passing the parameters, calling the function, returning, deleting parameters, than for the functionality itself. The *inline* instruction suggests to the compiler that the function's call code should be replaced by function code expansion. It may also be useful to use this mechanism for a larger function called only one time.

In C++ and JAVA:

Function model and polymorphism are two concurrent techniques for generic programming. One or the other should be selected after examining the advantages and disadvantages offered on a case-by-case basis.

In C++:

Example of factorial calculation using a function model:

```
// recursion is used to iterate
template<int n>
inline int FACT () { return n * FACT<n-1>()  ;}

// specialisation is used to stop recursion
inline int FACT<0> () { return 1 ;}
```

Org.Principal	The main program must be limited to the highest-level control flow: creating tasks, initialisation, sequencing. It must not contain processing algorithms or calculations.
M=0;R=1;P=94;V=0	
Any	

Description

The main program must be short. It must summarise the processing process. It handles activation of general initialisation processing, of one or more processes required to attain a set target, and manages errors returned by called sub-programs.

Justification

Program understanding is facilitated if the main program contains only the software control flow.

Example

In FORTRAN

```
PROGRAM DEMO
     ........    declaration of variables
CALL INIT1
IF  (condition) THEN
     CALL PROC1
     CALL CONT1
     ..............
ELSE
     CALL PROC2
ENDIF
     ...........
END
```

Org.MatérielIndep	Codes that have dependencies with hardware or operating system must be kept separate from the rest of the software code.
M=2;R=0;P=84;V=0	
Any	

Description

Dissociate as much as possible the hardware interface and operating system from the software being developed. This rule must be applied, even if a homogeneous module need to be divided in order to extract the non-portable functionalities.

Justification

Enhances portability.

Example

Not applicable.

7.2. CODE LAY-OUT

Pr.Indentation	Code must be indented. A convention for representing control structures must be defined and respected.
M=2;R=0;P=74;V=2	
Any	

Description

The code created must use uniform indentation throughout the entire project. The recommended value for indentation is 3 characters. The value used for indentation may be conditioned by the code editing, presentation and printing tool used for the project. A convention for control structures lay-out must also be defined.

Justification

Indentation enhances readability and improves code comprehension.

Example

In IDL:

Control structures are explicitly written in IDL:
Example of presentation of WHILE

```
WHILE (index GT 3) DO BEGIN
     index = index + 1
     PRINT, “INDEX = “, index
ENDWHILE
```

Pr.Aeration	The text in a program must be well-spaced. Operators and operands must be separated by spaces.
M=2;R=0;P=75;V=2	
Any	

Description

Unary operators must be followed or preceded by their operand without spaces. Binary operators must feature spaces on either side.

Justification

Ensures uniform program presentation and allows unary operators to be distinguished from other operators.
Enhances readability.

Example

In C:

```
Result = x + y ;
```

Pr.Instruction	There should be no more than one instruction per line.
M=2;R=0;P=76;V=2	
Any	

Description

Long instructions may extend over several lines; they must therefore be cut:

- before: reserved words, operators, assignment symbols, opening parentheses
- after: a comma, semi-colon

Justification

Ensures uniform program presentation and allows unary operators to be distinguished from other operators.
Enhances readability.

Example

In FORTRAN 77
The character "&" is used to indicate following line (in column 6).
In ADA

```
THE_ACCUMULATION_OF_TWO_LONG_IDENTIFIERS
     := THE_VALUE_OF THE_FIRST_IDENTIFIER
            + THE_VALUE_OF THE_SECOND_IDENTIFIER;
```

Pr.LongLine	The maximum number of characters in a line of source code is less than a limit defined for the project.
M=2;R=0;P=77;V=2	
Any	

Description

The limit must be established. Firstly, it must account for potential compiler limits. Secondly, it must ensure that the project entry, display, analysis and printing resources all allow the code to be readily handled and consulted.
A high limit will be set to allow the programmer to readily enter code, insomuch as the rules proposed here create long lines: descriptive name, prefix, naming by association, parameter alignment, indentation, etc.
This also applies to comments.

Justification

Certain compilers ignore the characters that exceed a given line length. After a certain length, long lines are not easily displayed and printing is truncated. Setting a maximum code line length in the project at a high value, but one that is below set limits, facilitates compilation, and source handling and consultation.

Example

Not Applicable

Pr.CartStd	A standard comment box defined for the project must be used to comment on the header of each module and the definition of an operation.
M=2;R=1;P=50;V=1	
Any	

Description

This header presents the essential logic behind the module or the operation, as well as critical programming aspects (for example: pre-conditions for calls, processing exceptions, possible side effects, portability constraints, task synchronisation conditions, etc.). A header may be addressed to the person using or maintaining the module.
The exact contents of headers should be set out in the initial conventions of each project.

Justification

This rule results in a more uniform, readable and maintainable code.
It guarantees the existence of at least one header per file.

Example

In C

File header comment (c or h):

```
/////////////////////////////////////////////////////////////////////////
// PROJECT: <>
// APPLICATION: <>
// AUTHORS: <>
// CREATION DATE: <>
// DESCRIPTION: <>
//
/////////////////////////////////////////////////////////////////////////
```

Function header comment:

```
/////////////////////////////////////////////////////////////////////////
// FUNCTION NAME: <>
// ROLE:
// INPUT PARAMETERS:
// UPDATE PARAMETERS:
// RETURN CODE: <>
/////////////////////////////////////////////////////////////////////////
```

Pr.CartDonnée	Each data declaration must be commented.
M=2;R=0;P=78;V=1	
Any	

Description

Variables must be presented and commented, particularly those with critical functional importance.

Justification

Maintenance is significantly facilitated.

Example

Not Applicable

Pr.CommFonc	Comments must be functional and not duplicate the code.
M=3;R=0;P=42;V=0	
Any	

Description

Comments must serve exclusively to provide additional information to the reader; they must supplement the information that the reader finds in the code itself, such as the names of types, variables, formal parameters, loops, blocks and exits, in the introduction of temporary variables or sub-types, and in using qualification or renaming. The additional information provided by the comment must be significant: a specific feature of the variable, the purpose of the block, the originality of the algorithm, etc.
Comments must not be used to paraphrase or to make up for inexpressive names of identifiers, parameters or functional blocks. The goal is therefore not to attain a certain percentage of comments, but rather to have only **useful** comments.
Comments provide the answers to "why" while the code indicates "how".
Comments may also be non-existent if the code is expressive enough on its own.

Justification

Limits double maintenance and code/comment discrepancies.

Example

Not Applicable

Pr.CommIdent	Comments must be located in the same area as the relevant code, and indented at the same level as this code.
M=2;R=0;P=79;V=2	
Any	

Description

For short assignment instructions, comments should be placed at the end of the line.
In languages such as C, C++ or JAVA, the series of closing brackets does not indicate to which opening bracket it corresponds. Incorrectly positioned closing brackets are a frequent cause of errors. Comments allow ambiguity to be avoided.

Justification

Enhances visibility.

Example

In C:

In C or C++, each closing bracket can have comments.

```
while (Condition)
{
     Processing_1;
     if (Condition_2)
     {
           Processing_2;
     } // end of case 2
} // end of processing loop body
```

7.3. IDENTIFIERS

Id.IdentSignif	Identifiers must be descriptive.
M=3;R=1;P=25;V=0	
Any	

Description

Identifiers should be chosen for their explicitness. Abbreviations are prohibited unless found in the project glossary or an integral part of the project culture.

Justification

Attempt to have source language that is as close to natural language as possible, which may be readily understood and which is unambiguous.

Example

In FORTRAN 77

It is often difficult to apply this rule in FORTRAN 77 (symbolic names limited to 6 characters). If portability and/or security constraints allow, it is recommended that the features of the "extended" FORTRAN 77 standard be used, in order to code names using 31 characters.

Id.IdentRegle	Identifiers must be simple or created by concatenating several terms; the same concatenation, use of determinants and upper and lowercase letters must be common to all identifiers used in the project.
M=2;R=1;P=51;V=0	
Any	

Description

Rules for naming identifiers are defined at the beginning of the project. They are customised for the project and concern all activities. Identifiers must be differentiated from other words in the language (in particular, reserved words).
In strict FORTRAN 77 (symbolic names limited to 6 characters), rules may be defined as explicitly as possible while being compact.

Justification

Enhances readability.

Example

In ADA

1. The different words that make up an identifier are separated by an underscore
`A_TELEMETRY_BLOCK, THE TELECOMMAND_BATTERY, etc.`
2. Global variables are in uppercase letters, local variables are in lowercase and their names represent what they identify.

Id.NomDonnee	The name of a datum must be a common name taken from everyday language; the plural form must be used if the datum is a set or group.
M=3;R=0;P=43;V=0	
Any	

Description

Not Applicable

Justification

Enhances readability.

Example

In C++

```
TheStarTracker
TheResults
TheDaysOfTheWeek
```

Id.VarSignif	The name of a variable must convey its meaning.
M=3;R=1;P=26;V=0	
Any	

Description

The name of a variable must fully identify said variable. It must both express what the variable is and identify the variable unambiguously. A naming rule should be adopted for variable identifiers and the specifiers used. For example: an article or possessive adjective for a variable, a verb phrase to express a true or false status for a Boolean. In addition, each variable name should have at least 3 characters, except loop indexes.

Justification

Enhances source readability and the distinction between variable identifiers.

Example

In ADA:

```
THE_TM_STATUS: A_CORRECTIVE_CODE;
```

Id.VarType	The name of a variable may also convey its type, nature or scope.
M=2;R=1;P=41;V=1	
Any	

Description

This rule essentially concerns weak typing languages or those for which static control is not strict.

Justification

For these languages, this rule improves code quality.

Example

In PVWAWE
- prefix local COMMONs to a module using **CL_**
- prefix COMMONs shared with other modules using **CG_**
- prefix the name of constants using **CST_**
- prefix structure types using **TS_**

In IDL:

Variable are named according to the rule: *Scope_Type_Desc*.

```
"Scope" represents the scope of the variable:
Global variable: use "g_"
Local variable: use "l_"
Variable belonging to a global common block: use "CG_"
Variable belonging to a local common block: use "CL_"
Member data of an object: use "m_"
"Type" represents the type of variable:
BYTE type: use "b"
INTEGER type: use "n"
Unsigned LONG type: use "ul"
Signed LONG type: use "l"
FLOATING type: use "f"
DOUBLE type: use "d"
COMPLEX type: use "c"
STRING type: use "s"
OBJECT type: use "o"
POINTER type: use "p"
STRUCTURE type: use "st"
"Desc" is the description of the variable.
```

Id.ConstSignif	The name of a constant must convey its meaning and not its value.
M=3;R=1;P=27;V=0	
Any	

Description

The name of each constant must follow the naming rules defined for the project, except when being reused. In particular, these rules must allow the user to readily distinguish between constants and variables. The names of constants should be written in UPPERCASE letters, and each name should represent what it is identifying. This rule also applies to constants defined in enumerated types and in macros in the C and C++ languages.

Justification

Enhances readability.

Example

In ADA:

```
SIZE_OF_BUFFER: constant := 100;    --  rather than HUNDRED;
```

Id.ClasseType	Though not dictated by the language, the name of a type or class must be a general term that identifies a group or category of data.
M=3;R=0;P=40;V=0	
Any	

Description

Not Applicable

Justification

Enhances source readability and the distinction between type identifiers.

Example

In ADA

```
type A_CORRECTIVE_CODE is array (1 .. NB_OF_BITS) of BOOLEAN;
```

In C

```
typedef struct
{
     int positionX;
     int positionY;
} tPosition;
```

Id.Pointeur	If the language supports pointer or reference concepts, the name of a pointer or reference must convey the semantics of the object it identifies (pointed or referenced object).
M=3;R=3;P=4;V=0	
Any	

Description

Not Applicable

Justification

Enhances readability and the distinction between pointer identifiers.

Example

In ADA:

```
type A_LINK_PTR is access A_LINK ;
PTR_CURRENT : A_LINK_PTR ;
```

Id.Procedure	Procedure names must be infinitive verbs or verb groups that indicate the action to be completed.
M=3;R=1;P=80;V=0	
Any	

Description

The verbs must be active verbs. This rule also concerns macros in the C and C++ languages. When data is masked, the write-access methods will have a standard prefix.

Justification

Enhances readability.

Example

In C++:

Definition of a Complex type and the access methods to the real and imaginary parts of the complex number:

```
class Complex {
      …
      {}
public: // Access
      int virtual obtainRealPart(void)
            // Real part of complex number
            {
                return realPart_;
            }

      int virtual obtainImaginaryPart(void)
            // Imaginary part of complex number
            {
                return imaginaryPart_;
            }
…
};
```

Id.Tache	Task names must be composed using procedures and events associated with and used to trigger or sequence the task.
M=3;R=2;P=19;V=0	
Any	

Description

Associated procedures are operations called by the task: when the task is very functionally consistent, only one operation is called by this task.

Justification

Enhances readability.

Example

In C:

`void GetStarAngle ()` is the procedure called cyclically each second to acquire the angle with a given star; the associated task will be called:

```
GetStarAngle_1s
```

In ADA:

```
task THE_BUFFER is
      entry TAKE (THE_ELEMENT: out AN_ELEMENT);
end THE_BUFFER;
```

Id.Fonction	Functions must be named using a noun that represents the value supplied by this function. For a function that returns a Boolean value, a verb phrase should be used to express a true or false status.
M=3;R=1;P=29;V=0	
Any	

Description

This rule also concerns macros in the C and C++ languages. When data is masked, the read-access methods will have a standard prefix.

Justification

Enhances readability.

Example

In ADA:

```
function SQUARE_ROOT (OF: in A_REAL) return A_REAL;
function ALREADY_EXISTS  (THE_PATH: in A_PATH) return BOOLEAN;
```

Id.NomParFormel	The name of a formal parameter must convey the relationship between the parameter and the operation concerned.
M=3;R=1;P=80;V=0	
Any	

Description

Not applicable

Justification

Enhances readability. Facilitated reading must take precedent over facilitated writing. A more explicit semantic form is obtained as a result.

Example

In ADA

```
procedure CREATE  (    WITH_THE_STRING   : in  STRING;
                       THE_WORD          : out A_WORD);
```

7.4 DATA

Don.Declaration	All data used must be explicitly declared.
M=3;R=3;P=1;V=2	
Any	

Description

This rules concerns permissive languages that allow declarations to be omitted.
Declaration instructions will be specified (public, private, static, etc.).
In addition, all data declared must be used.

Justification

Improves maintainability and reliability.

Example

In FORTRAN 90
The instruction IMPLICIT NONE is mandatory.
In C++
Declaration instructions will not be used by default.

Don.Separee	Each piece of data must have a separate declaration.
M=2;R=1;P=45;V=2	
Any	

Description

One line will be used for each declaration.

Justification

Each declaration will be able to be commented as a result.

Example

In C or C++

Incorrect

```
float sBeg sEnd, sAverage;   // Speed calculation variables.
```

Correct

```
float sBeg;         // Speed at the beginning of acceleration.
float sEnd;         // Speed at end of acceleration.
float sAverage;     // Average speed.
```

Don.Typage	Data must be systematically and explicitly typed.
M=3;R=3;P=2 ;V=2	
Any	

Description

Types must correspond to the data variation domains. The most "limited" definition domains will be used, in accordance with data semantics.
All allocation directives must be explicitly specified.

Justification

An absence of explicit typing may indicate a programming anomaly.
Assigning a type by default may create errors and portability issues.

Example

In ADA

```
integer NBNODE:= 10
type(NODE), dimension(:), allocatable :: TABNODE
integer, dimension(:,:), allocatable, target :: CONNECTIVITY
```

In FORTRAN 90

Use the allocated declaration form.

Don.TypeAnonyme	Anonymous types must not be used.
M=3;R=2;P=32;V=1	
Any	

Description

An anonymous type is a type that is implicitly declared through data declaration, but that is not declared as such as a type.
Data declarations made using semantically-equivalent anonymous types are not allowed.
In C, compound literals, whose scope in a function is limited to the enclosed instruction block, and tags should be avoided.

Justification

Eliminates type incompatibility problems.
Enhances scalability and type reuse.

Example

In ADA

Replace:

```
THE_CHESSBOARD: array (1 .. 8, 1 .. 8) of A_SQUARE;
THE_OTHER_CHESSBOARD: array (1 .. 8, 1 .. 8) of A_SQUARE;
```

by:

```
AN_CHESSBOARD is array (1 .. 8, 1 .. 8) of A_SQUARE;
THE_CHESSBOARD: AN_CHESSBOARD;
THE_OTHER_CHESSBOARD: AN_CHESSBOARD;
```

Don.Localite M=3;R=1;P=30;V=1 Any	Local data declarations are preferred over more global declarations: data that is local to a module are preferable to global data, formal parameters are preferable to global data, local data for an operation are preferred over module-level data, and local data for an instruction block are preferable to local data for an operation.

Description

This rule is very general and must be applied according to context and language.

Justification

Readability is enhanced if variables are limited in scope (said variables are not relevant outside of their scope).
The use of more global variables is always more costly in terms of memory use and access time.
The use of more global variables renders the code less generic and more difficult to maintain or reuse.
Use of more global variables makes the code less reliable.
Where appropriate, the compiler may avoid useless allocations or code: local data that is not assigned is not allocated; local data that is not reused is not calculated (the code that assigns it is not generated). This is particularly effective when conditional compilation is used.
This is to limit the scope of the variables as much as possible.

Example

In FORTRAN or IDL

The COMMON mechanism should be avoided in order to use parameters

In SHELL or PERL

Environment variables should be avoided

In SHELL

Local variables for a function should be defined using a typeset or a local attribute

In C++ or JAVA

Static data should be avoided

In C++

An example of an itemised declaration:

```
void f4 (int &x, int &y, int z  []) {

    // PreProcessing
    f (x);
    g (y);

    // Development of local 1
    int local1 = x+y ;

}
```

Don.Invariant	Constants must be defined for entities whose value is invariant.
M=2;R=1;P=52;V=1	
Any	

Description

If an invariant is used only once (for a given semantic), the definition of a constant may be debated.

Justification

This rule allows invariants to be guaranteed, thereby enhancing reliability. In addition, code is not impacted if the value of the constant is modified (location of the modification and uniqueness). This facilitates adaptability and scalability.

Example

In ADA

```
package PACKAGE_EXAMPLE is
     MAX_LINE_LENGTH: constant := 255 ;
     type A_LINE_LENGTH is range 0.. MAX_LINE_LENGTH;
     MY_CARD_LENGTH: constant A_LINE_LENGTH:= 80 ;
     ...
```

Don.Enumeration	The use of constants or symbols must be preferred (enumerative, if the language allows) over the use of whole numerical data. The use of whole numerical data must be essentially limited to simple calculation or counting.
M=2;R=2;P=36;V=1	
Any	

Description

All constants (including table dimensions) must be named using symbols. Literal constants are prohibited, except in special cases such as increments of 1 and -1. Constants must be typed, if the language allows. If the language proposes several mechanisms for implementing constants, the mechanism that is best adapted to the context should be used.

Justification

This technique ensures code consistency, scalability and reusability.

Example

In ADA

Replace:

```
type AN_INSTRUMENT is range 1.. 4;
            -- 1 corresponds to CAMERA
            -- 2 corresponds to ALTIMETER
            -- 3 corresponds to INTERFEROMETER
            -- 4 corresponds to LASER
```

by:

```
type AN_INSTRUMENT is (CAMERA, ALTIMETER, INTERFEROMETER, LASER);
```

In C and C++

The *#define* instruction does not exit in the C language, but is rather a command for its pre-processor *cpp*: *#define* allows a literal constant to be replaced by its value in the source code. The compiler works on a post-processing version of the source and therefore does not know the original literal constant. Declaring enumerations increases the possibilities for control.

Incorrect example:

```
#define WHITE 0
#define BLACK 1
#define RED 2
#define GREEN 3
#define BLUE 4
int aColour;
aColour = RED; // correct for compilation
```

Correct example:

```
typedef enum { white, black } tColour1 ;
typedef enum { white, black, red, green, blue } tColour2 ;
tColour1 aColour = red; // justified refusal for compilation
```

Don.Structure	When a conceptual object must be implemented as several data, this data must be grouped in a structuring entity (class, structure, record, type) according to the possibilities provided by the language.
M=3;R=2;P=18;V=0	
Any	

Description

Not Applicable

Justification

This ensures enhanced consistency between units of code.

Example

Not Applicable

Don.Homonymie	The use of homonyms must be avoided except in cases of overload or explicit redefinition.
M=2;R=1;P=46;V=1	
Any	

Description

A variable that is local to a sub-program must not have the same name as a compilation unit global variable or an external variable.

Justification

Enhances readability.
Avoids visibility conflicts involving rules that may be complex.

Example

In C

Using naming rules makes it possible to distinguish between local variables and static variables, thereby avoiding this type of error, which is often difficult to detect.

Don.Initialisation	Variables must be initialised before being used for the first time.
M=2;R=3;P=11;V=2	
Any	

Description

All variables must be initialised, either when declared, or before being used for the first time. If possible, variables should be initialised when declared: this concerns, in particular, all simple variables (integer, float, char, etc.), pointers and references, local variables and environment variables used by the program and the scripts.
Initialisation must be performed at declaration, if the variable can be initialised with a significant value.
Note that some languages may impose or verify variable initialisation, specifically for local variables.

Justification

Avoids side effects and potential portability problems. If the variable is not initialised it amounts to using the memory initialisation performed by the operating system, which may be different from one computer to another.

Example

In FORTRAN

When COMMON is used to pass variables from one service to another, it must always be initialised by the caller.

In C

```
const int MAX_STRING = 80;
int Number_aircraft = 0;
char Firstname[MAX_STRING]="";
const int SIZE = 10;
int Tab[SIZE]={1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10};
```

Data.PointeurNonAff	If the language supports the pointer concept, when a pointer is not associated with a specific object at declaration, a comment must specify the object that will be associated with it and, if the language allows, initialise it to null.
M=2;R=3;P=13;V=1	
Any	

Description

The purpose of this rule is to document the use of pointers and references with a complicated dynamic.

Justification

One of the most frequent causes for error when using pointers or references is the use of a null reference.

Example

In JAVA

```
Point P1 ;  // First end of segment
            // Will be assigned as soon as the segment is created
Point P2 ;  // Second end of segment
            // Will be assigned as soon as the segment is created
```

```
Segment S = new Segment () ;
…
P1=S.First () ;
P2=S.Last () ;
```

Don.LocalUnique	Each local datum must have a unique use.
M=2;R=0;P=80;V=0	
Any	

Description

The definition of general data reused at various points in the code should be avoided.

Justification

Code is more consistent.
This reduces the risk of side effects due to previous initialisation of the variable.
Increasing local data does not adversely impact performance: recent compilers know how to effectively manage associated resources.

Example

Not Applicable

Don.Utilisee	All data that is defined must be used; a datum that is no longer used must be deleted.
M=2;R=0;P=81;V=2	
Any	

Description

Local data created for a specific need should be deleted when this need ceases to exist. This facilitated by respecting rule Don.TypeAnonyme .

Justification

A variable that has been declared but not used corresponds to useless code that adversely impacts readability and pollutes the program

Example

Not Applicable

Don.TablePrincipe	The processing principal (line x column or column x line) for double entry tables must be defined.
M=2;R=2;P=37;V=0	
Any	

Description

The principles for using double entry tables must be defined; how they should be declared, and which indexes correspond to lines and columns.

Justification

Without a specific rule, confusion may arise between two developers: one sees the double entry table as it is and the other sees it as being transposed.
In most languages, the addressing mode for table elements skews performance according to whether the table is browsed line-by-line or column-by-column

Example

In FORTRAN

Process tables by column rather than by line
Use (processing a column in the innermost loop):

```
DO   J = 1,N
 DO   I = 1,N
  A(I,J) = B(I,J) * 5.0
 END DO
END DO
```

rather than (processing a line in the innermost loop):

```
DO  I = 1,N
 DO  J = 1,N
  A(I,J) = B(I,J) * 5.0
 END DO
END DO
```

In PVWAWE

Loop indexes in a table beginning with columns and then lines.

In IDL

For multi-dimensional tables, loop indexes by browsing through the first indexes in the innermost loops

Don.TableOper	Global operations for tables (initialisation, copy, duplication, comparison) must be performed using standard primitives provided by the language, when they exist.
M=2;R=2;P=38;V=1	
Any	

Description

Not Applicable

Justification

Code is more readable.
Code is more efficient.

Example

In C and C++

The functions memset, memcpy etc. should be used

In IDL:

The ARRAY_EQUAL function allows for quick comparison of the contents of 2 tables, without having to use FOR loops or WHERE instructions.

Don.ChaineOper	Global operations for character strings (initialisation, copy, duplication, comparison, search, modification) must be performed using standard primitives provided by the language, when they exist.
M=2;R=2;P=39;V=1	
Any	

Description

Not Applicable

Justification

Code is more readable.
Code is more efficient.

Example

In FORTRAN 77:
LEN and INDEX functions are used.
In C:
<string.h> interface functions (strcpy, strcmp, strcat, etc.) are used.
In C++:
The String type from STL will be used.
In PERL:
Comparing character strings requires the use of dedicated alphabetical operators (eq, lt, gt, le, ge) rather than standard numerical operators (==, <, >, <=, >=). Using numerical comparison operators on character strings does not cause a syntax error (only a warning), but will not return a correct value.

Don.AllocDynbord	Dynamic memory allocation is prohibited.
M=0;R=3;P=30;V=1	
On-board	

Description

All instructions that lead to dynamic memory allocation or deallocation are prohibited.

Justification

Allocation and subsequent deallocation may lead to significant memory fragmentation. To avoid CPU load problems, it is clear that bringing a process on-board to continuously defragment the memory is not acceptable.

Example:

In C:
Use of dynamic memory allocation mechanisms that use malloc/free (standard library) is prohibited in on-board real-time applications.

Don.AllocDynSol M=0;R=2;P=61;V=0 Ground	If the language supports the concept, dynamic memory allocation must be used sparingly, and with caution.

Description

The project may choose to prohibit dynamic memory allocation, or to limit it to certain compilation units in order to manage memory usage.

Justification

Dynamic allocation requires an analysis of the application's dynamic, and may lead to memory fragmentation problems that may adversely affect performance

Example

Not Applicable

Don.AllocEchec M=0;R=2;P=62;V=1 Ground	If the language supports the concept of dynamic allocation, the potential failure of a memory allocation request must be systematically provided for.

Description

A dynamic memory allocation request may fail as a result of insufficient available memory.
In all cases, a process must exist in the event of failure.

Justification

A memory allocation error is a serious error.
It is generally very difficult to trace the cause (the failure of the allocation request) from one of the effects.

Example

In C++

The following possibilities exist to prevent the risk of allocation failure:
Define a global error processing function, that is positioned as a function called implicitly when *new* fails, due to the error management primitive "*set_new_handler*".
Redefine the *new* operator for a given class: this technique is more complicated, but creates processing adapted for each class.
Use the exception of the standard library "bad_alloc ".
Test the return value of a call to new and provide for an ad hoc process if a null value is returned, which corresponds to an allocation error. In this case, the *new* operator should be used with the (*nothrow*) instruction to avoid throwing an exception.

Don.AllocLiberation	All allocated memory must be freed at the same conceptual level.
M=1;R=1;P=53;V=0	
Ground	

Description

All memory area allocation involves explicit deallocation as soon as possible and at the same conceptual level: operation, service, module, class. It should be noted that this rule is not applicable for languages such as JAVA, which automatically free memory.

Justification

Systematic deallocation saves memory resources.
It is easiest to free memory at the conceptual level at which this memory was allocated.

Example

In C

If a module offers a memory allocation function, it should also offer a function to free memory.

In C++

If constructors allocate memory, a destructor frees memory.

Donc.AllocErreur	An error that occurs during processing must not cause memory to not be freed.
M=0;R=2;P=63;V=0	
Ground	

Description

A code sequence that leads to an exception risks skipping the code that frees resources.

Justification

Allocated resources must be freed, regardless of code sequence.

Example

In C++

An example of a function interrupted by an exception that leads to resources not being freed

```
void Exception1 (void) {
      try {
            tA * pA ;

            // Local allocation
            pA = new tA (0);

            // Processing interrupted by an exception
            // ...

            // Resource freed
            delete pA ;
          }
      catch (MyException e) {
            // ... Processing exception
      }
```

```
}
```

An example of a function throwing an exception without freeing resources

```
void Exception2 (void) throw (MyException) {
     tA * pA ;
     // Local allocation
     pA = new tA (0);
     // Processing throwing an exception
     if (true) throw MyException (1);
     // Resource freed
     delete pA ;
}
```

7.5. PROCESSING

Tr.TestEgalite	Use of the equality or difference test must be replaced by inequality where possible.
M=0;R=3;P=45;V=1	
Any	

Description

Equality or difference tests are difficult to manage when browsing intervals.

Justification

Enhances robustness.

Example

In C:

Replace:

```
for (int i=0 ; i != MAX ; i++)
```

by:

```
for (int i=0 ; i < MAX ; i++)
```

Tr.ComparaisonStrict	Strict comparison (equality, difference) between floating numbers (real, complex) must be replaced by inequality.
M=0;R=3;P=46;V=1	
Any	

Description

Equality between reals will never be tested using an equality operator, but rather by framing their difference.

Justification

The strict equality of two real-type operands does not make sense.

Example

In ADA:

Replace:

```
if MY_REAL = YOUR_REAL then
```

by:

```
if (abs(MY_REAL - YOUR_REAL) < EPSILON) then
```

where EPSILON represents machine accuracy.

Tr.ModifConst	The value of a constant must not be modified.
M=3;R=3;P=3;V=2	
Any	

Description

This rule concerns languages for which the concept of "constant" is not defined.
In C and C++, casting mechanisms that might modify the value of a constant will be avoided.

Justification

Constants represent invariants that must be respected.

Example

In C or C++

Avoid the following code:

```
const double pi=3.1415926 ;
const double * ptr1 = & pi ;
double * ptr2 = (double *) (ptr1) ;
*ptr2 = 3 ;
```

Tr.ControleRacc	If the language supports the concept, shortcut forms of control must be used whenever appropriate.
M=2;R=2;P=35;V=1	
Any	

Description

Shortcut control forms are specific to the languages and correspond to common specific cases of control forms: iterative forms, decisional forms etc.

Justification

Enhances readability.
Speeds code execution.

Example

In C:

Use the instruction `for` rather than `while`, when possible.
Use a switch rather than a series of `if` – `else if` if the conditions concern the enumeration of the values in a whole expression.

In ADA:

Prefer:

```
      if Y /= 0 and then (X / Y) = 10 then .    -- OK
```

over:

```
      if Y /= 0 and (X / Y) = 10 then ...       -- CONSTRAINT_ERROR
 POSSIBLE
```

Tr.Choix	A choice instruction must be used rather than a simple conditional instruction when there is more than one alternative.
M=2;R=0;P=70;V=1	
Any	

Description

Not Applicable

Justification

For a multiple choice, the compiler builds a table of branching addresses for each case (which is possible if the values to be tested are numerically consecutive), so that the access time to each case does not vary. Enhances code readability and self-description.

Example

In ADA:

```
type A_RESPONSE is (YES, NO, MAYBE) ;
THE_RESPONSE_OF_THE_OPERATOR := OPERATOR_RESPONSE ( OF_THE_OPERATOR =>
ACTIVE_OPERATOR) ;
case THE_RESPONSE_OF_THE_OPERATOR is
     when YES          => PROCESS;
     when NO           => DO_NOT_PROCESS;
     when MAYBE  => DECIDE;
end case;
```

In FORTRAN 77:

Avoid using calculated goto, which is less readable, and use nested if elseif

In C and C++:

When there is more than one alternative possible, use a **switch/case** instruction rather than an **if/else if/else** instruction

Tr.OrdreChoix	When using a choice instruction, all possible cases must be provided, preferably explicitly and in the "logical" order of the cases.
M=3;R=2;P=17;V=1	
Any	

Description

This means, among other things, that processing by default cannot be used.

Justification

Improves software maintainability and reliability.

Example

In ADA:

Prefer:

```
type A_RESPONSE is (YES, NO, MAYBE);
THE_RESPONSE_OF_THE_OPERATOR := OPERATOR_RESPONSE ( OF_THE_OPERATOR =>
ACTIVE_OPERATOR) ;
case THE_RESPONSE_OF_THE_OPERATOR is
     when MAYBE  => DECIDE;
     when YES          => PROCESS;
     when NO           => DO_NOT_PROCESS;
end case;
```

over:

```
type A_RESPONSE is (YES, NO, MAYBE);
THE_RESPONSE_OF_THE_OPERATOR := OPERATOR_RESPONSE ( OF_THE_OPERATOR =>
ACTIVE_OPERATOR) ;
case THE_RESPONSE_OF_THE_OPERATOR is
     when YES          => PROCESS;
     when OTHERS       => DO_NOT_PROCESS;
end case;
```

Tr.Goto	The unconditional branching instruction (goto) must only be used in very limited and specific cases.
M=3;R=3;P=6;V=2	
Any	

Description

Goto must be used only for error processing. If the language supports exception processing, as does ADA, JAVA or C++, the use of goto is prohibited.
It is prohibited to perform backward branching, or in a structured instruction such as a loop.

Justification

The instruction **goto** often leads to a destructured program, which increases complexity and the risk for errors.

Example

In C

Use of goto is tolerated for error processing:

```
while(Condition_1)
{
     Processing_1;
     if (Condition_2)
     {
           goto Error
     }
     Processing_2;
     if (Condition_3)
     {
           goto Error
     }
     ...
}
goto End;

Error: Processing_Error;

End: ...
```

However, the long jump (set jump, long jump) is prohibited.

Tr.BoucleSortie	A loop must feature a unique nominal exit.
M=3;R=2;P=16;V=2	
Any	

Description

A well-structured loop algorithm must not require several possible exits. It is the condition that must potentially test the different possibilities for interrupting the loop.
Use of the unconditional exit instruction can be tolerated if respecting the rule leads to far more complex loop programming.

Justification

A large number of loop exits destructures the program and adversely impacts comprehension.
The unconditional exit instruction in a loop destructures the program and increases its complexity.

Example

In C

```
// incorrect
 Index = 0;
 while (Index < MAX)
 {
      if (Letter[Index] == KEY)
      {
            break;
      }
      Index ++;
      Processing;
 }
// correct
 Index = 0;
 while (Index < MAX) && (Letter[index] != KEY))
 {
      Index ++;
      // processing
 } // end of loop for variable
```

Tr.ModifCondSortie	The loop exit condition must not be modified in loop processing.
M=3;R=3;P=8;V=1	
Any	

Description

The loop exit test must compare the loop parameter value with a value known at loop entry and independent of loop body processing.

Justification

Enhances readability.

Example

In C

```
// incorrect
 for (I = 0; I == Max ; I++)
```

```
{
      ...
      Max = Func_1();
      ...
}
```

Tr.ModifCompteur	The loop counter must not be modified in loop processing.
M=3;R=3;P=7;V=2	
Any	

Description

The loop parameter value must not be modified by loop body processing, unless to provide iterative instructions that do not implicitly modify the loop counter. In the latter case, the loop counter will only be modified once, at the end of the loop body.

Justification

Enhances readability.

Example

In C

```
// Incorrect
for (I = 0; I <= max; I++)
{
      ...
      I = Func_1();
      ...
}
// Correct
while (I <= max)
{
      ...
      I++ ;
}
```

Tr.RecursifSol	Recursive operations must not be used unless they are conceptually simpler than an equivalent iterative operation.
M=0;R=2;P=64;V=1	
Ground	

Description

All recursive problems have iterative solutions. However, certain types of data are particularly well-suited to recursive algorithms. In this case, this type of solution should be preferred. However, before applying a recursive solution, the mechanisms for exiting recursivity should be defined from the design phase. For example, the maximum call depth attained during execution can be assessed to determine whether it is permissible. If this is the case, this maximum value can be used as a type limit for a depth control parameter in order to correctly process the exception raised by a test in the event this value is exceeded.

Justification

The use of recursivity is more difficult to test and to understand: its use must be limited for this reason.

Example

Not Applicable

Tr.RecursifBord	Recursivity is prohibited.
M=0;R=3;P=64;V=2	
On-board	

Description

This rule concerns direct recursivity, as well as indirect or cross recursivity.

Justification

Recursivity may cause non-deterministic behaviour that may be dangerous when the depth (i.e. the number of successive calls) is not known from the outset. It is therefore difficult to assess the size of the execution stack required to execute a recursive algorithm.

Example

Not Applicable

Tr.FonctionSortie	A function must only contain one exit instruction.
M=3;R=2;P=15;V=2	
Any	

Description

Functions are exited using a return instruction that must be accompanied by a significant nominal value. Multiple exit points are tolerated for error processing (return instruction associated with the return of an error code value).

Justification

This rule improves the maintainability of the sub-program in the event that processing must be added before the exit instruction.
One single nominal exit and one error exit reduce the complexity of the sub-program and the associated test effort required.

Example

In C:

```
// function returning an integer
int Function_1 (void)
{
     int Res ;
     if feof (F_Desc)
     {
            Res = 0;
     }
     else
     {
```

```
            Res = 1;
        }
        return (Res);
    }
```

Tr.ProgDefensive	Defensive programming, which involves the use of pre-conditions and post-conditions, should be preferred.
M=1;R=3;P=20;V=1	
Any	

Description

Defensive programming involves adding assertions in the code in order to verify invariants: as inputs, these are pre-conditions; as outputs, these are post-conditions. If an assertion is not verified, an error is reported or an exception is flagged.
To avoid adversely impacting performance, assertions may be made "optional" using conditional compilation techniques.

Justification

Function use constraints are formally specified (pre-condition). Post-conditions provide the user with guarantees regarding processing performed.
Application fine tuning is facilitated.

Example

In SHELL

Programs must verify the validity of all of their arguments before beginning processing and check all user entries (with the keyboard)

In C++

```
class TableUnsigned{
     // Table of positive integers.
public: // Constructor
     Table(unsigned min, unsigned max);
           // Creation of a table of min and max. limits
public: // Access
     int& operator [ ](int index)
           // Read-write access
     {
           precondition( index > min() && index < max() );
                 // Pre-condition
           int& return = accessReadWrite(index);
                 // Call from delegation function.
           post-condition( return >= 0 );
                 // Post-condition
           return return;
     }
private: // operator delegation function [].
     int& accessReadWrite(int index);
}
```

Tr.Residus	No programming residue must exist as comments in the code: an instruction that is no longer used must be deleted.
M=2;R=0;P=71;V=2	
Any	

Description

Residues are often portions of dead code that appear after the code has been modified. However, unattainable code may exist due to robustness issues: this code must be commented.

Justification

Dead code weighs down code and negatively impacts readability.
Dead code may cause useless test efforts to be performed.

Example

In FORTRAN

All labels must be used. Labels that are no longer used must be deleted.

Tr.Parenthèses	Expressions must be systematically enclosed in parentheses.
M=1;R=2;P=42;V=2	
Any	

Description

Syntactically redundant parentheses are added to enhance readability.

Justification

Enhances code readability and facilitates portability.

Example

In C

Replace:

```
totalPressure = forceA / SurfaceA + forceB / SurfaceB ;
```

With:

```
totalPressure = (forceA / SurfaceA) + (forceB / SurfaceB) ;
```

Tr.CalculStatique	In compiled languages, it is better to perform calculations on static expressions at compilation, with maximum accuracy, rather than dynamically calculated expressions.
M=0;R=1;P=100;V=1	
Any	

Description

This point is even more important when the target machine is less efficient than the machine used for development.

Justification

Portability, performance

Example

In ADA:

In this first case, all data are constants: the values may therefore be calculated once and for all, at compilation, and with the level of accuracy offered by the development machine.

```
PI : constant := 3.1415926536;
PI_OVER_2 : constant := PI/2.0 ;
OF_DEGREE_TO_RADIAN: constant := PI_OVER_2 / 90.0 ;
OF_RADIAN_TO_DEGREE: constant := 1.0 / OF_DEGREE_TO_RADIAN;
```

In the second case, all data are variable: consequently, the compiler generates an initialisation code, which will be executed on the machine with the accuracy of the latter.

```
PI : real := 3.1415926536;
PI_OVER_2 : real := PI/2.0 ;
OF_DEGREE_TO_RADIAN: real := PI_OVER_2 / 90.0 ;
OF_RADIAN_TO_DEGREE: real := 1.0 / OF_DEGREE_TO_RADIAN;
```

Tr.Booleen	A complex conditional expression must be replaced by a unique Boolean that expresses a state.
M=2;R=0;P=72;V=1	
Any	

Description

Not applicable.

Justification

Enhances code comprehension and readability.

Example

In C

Replace:

```
if (forceA >= Limit1 &&  abs(forceB) < Limit2) …
```

With:

```
bool constraintA = forceA >= Limit1 ;
bool constraintB = abs(forceB) < Limit2 ;
bool conditionAB = constraintA && constraintB ;
if (conditionAB) …
```

Tr.DoubleNeg	Double negatives must be avoided in Boolean expressions.
M=2;R=1;P=47;V=2	
Any	

Description

Not applicable.

Justification

Double negatives make code difficult to understand.

Example

In ADA

Prefer:
```
 if  EXISTS then ....          -- COMPREHENSIBLE
```
over:
```
 if not DOES_NOT_EXIST then ....    -- HEAVY
```

Tr.MelangeType	Different types of data should not be mixed in the same expression.
M=3;R=3;P=9;V=2	
Any	

Description

The type of an arithmetic expression is generally determined by the compiler according to the type of operands and rules, which may sometimes elude the developer.
The following are exceptions to this rule:

- exponentiation by an integer (which is not an exception, strictly speaking, because coercion rules do not require conversion in this case),
- multiplication by a literal scalar integer of small value.

Justification

Enhances readability.
Allows expression assessment to be managed

Example

In FORTRAN

This example shows how mixing different types of data can produce assessment that is less accurate than initially desired
```
      REAL OPER1,OPER2
      DOUBLE ACCURACY RESUL,OPER3
            .............
      RESUL = OPER1 + OPER2 + OPER3
```
In FORTRAN 77, the previous instruction is equivalent to the following sequence:
```
            REAL TIME
            TIME = OPER1 + OPER2
            RESUL = DBLE(TIME) + OPER3
```

For maximum accuracy, the following should have been used:
```
            RESUL = DBLE(OPER1) + DBLE(OPER2) + OPER3
```

Tr.ComparConst	In a comparison with a constant, the variable must always be to the left of the comparison operator.
M=1;R=1;P=59;V=1	
Any	

Description

The expression comparing a variable to a constant may be written in two different ways, depending on whether the variable is compared to the constant, or vice versa. The code should always be written such as to compare the variable to the constant.

Justification

This type of comparison improves program readability.

Example

In C or C++

```
// incorrect
#define MAX_PARAM
if (MAX_PARAM >= Nb_Param)
{
     // nominal processing
}
// correct
#define MAX_PARAM
if (Nb_Param <= MAX_PARAM)
{
     // nominal processing
}
```

Tr.OrdreParFormel	The declaration order for formal parameters must be standardised.
M=1;R=0;P=116;V=1	
Any	

Description

The order will be defined for the project. Passing modes will not be used by default (for example, "in" in ADA).

Justification

Readability is improved by clarifying semantics.

Example

In ADA
Parameters are cited according to the order (***in*** then ***in out***, followed by ***out***).
In FORTRAN
The parameter list must use the following order:
- name of sub-program,
- input,
- input/output,
- outputs,
- return code

Tr.ParamOptionnel	Optional parameters must not be used when defining an operation.
M=1;R=0;P=115;V=2	
Any	

Description

Optional parameters will not be used when they are possible; some evolved languages, such as JAVA, have already abandoned the use of this mechanism, which is considered to be too risky.

Justification

The use of optional parameters masks real interfaces for operations and may cause them to exhibit surprising behaviour.

Example

In C++

Replace:
```
void Calculation (double x, double epsilon=0.00001) { …
```
With:
```
void Calculation (double x) { Calcul (x, 0.00001) ; }
void Calculation (double x, double epsilon) { …
```

Tr.ModifParSortie	An operation must not modify input parameters.
M=2;R=1;P=48;V=2	
Any	

Description

This is particularly true for non-scalar input elements (such as tables, structures and instances) that are passed by address or reference.

Justification

Declaring an input parameter as constant contributes to application reliability because this constant is verified by the compiler. This also serves as a formal comment for function clients, who are ensured that objects passed to parameters will not be modified.

Example

In C or C++

An input argument passed by address will be obligatorily protected by the qualifier `const`
```
int Seek_Ind (const int * Tab, int Dim, int Val)
{
    …
}
```

Tr.ModifVarGlobal	A function must not modify the value of a global variable or involve output parameters.
M=2;R=3;P=12;V=2	
Any	

Description

A sub-program with only one output parameter must be a function unless this parameter is simply a processing sub-product (in which case, a procedure will be used).

Justification

Enhances readability by better highlighting the object of the triggered action.
Eliminates side effects.
Improves reliability and portability.

Example

In ADA

Replace:

```
EXTRAPOLATE       (THE_ORBIT  => THE_CURRENT_ORBIT,
                   ON_THE_DATE => THE_CURRENT_DATE) ;
```

With:

```
THE_CURRENT_ORBIT := EXTRAPOLATION  (OF => THE_CURRENT_ORBIT,
                              ON_THE_DATE => THE_CURRENT_DATE);
```

Tr.ParSortie	All output parameters for a procedure must have received a value before the first processing condition, by initialisation by default, if necessary. The same is true for variables used to return the value of a function.
M=;R=;P=25;V=1	
Any	

Description

However, it is more important for the sub-program to accomplish what it must accomplish (or raise an exception) than to give values.
This rule does not apply if one of the parameters is a return code; certain other ***out*** parameters may not be initialised if they are not significant (this style of programming must generally be avoided, but this is not always possible, especially when interfacing with other languages).

Justification

Avoids random results.

Example

In C:

Correct example:

```
void setAlpha (int * alpha) {
      *alpha=0 ; // value by default
      if (…) ///
}
```

Incorrect example:

```
void setAlpha (int * alpha) {
      if (…) ///
}
```

7.6. ERROR MANAGEMENT

Err.Mecanisme	Error management must be performed using resources implemented in the language (exceptions or other). If the language offers several possible mechanisms, the mechanism that best respects the other error management rules should be selected. If the language does not offer specific error-management mechanisms, a dedicated error management module must be created.
M=3;R=3;P=10;V=0	
Any	

Description

This mechanism will be used for languages that support the exception mechanism. For the others, the return code for functions or services will be used, and rules concerning this return value will be defined and the use of these return values will be required for callers.

Justification

When the exception mechanism is available, it clearly and effectively processes errors that occur during execution. Caller and callee roles will be well defined.

Example

In ADA

The failure management policy uses the ADA language's exception mechanism. The return code technique associated with sub-programs, that which associates a validity marker to variables, and that which centralises errors are all prohibited.

In IDL

The CATCH mechanism should be used rather than the ONERROR mechanism

In Java and C++

Functional return must not be used for error management. The exception mechanism should be used.

In SHELL

A program that ends correctly must always explicitly return the value 0. A program must always explicitly return an error code in the event of an incident. The project may define error families and associated termination codes because the shell program may fail for various different reasons

Err.TraitementDiff	Error processing must be differentiated according to fault.
M=0;R=2;P=65;V=0	
Any	

Description

Failure processing is differentiated according to the type of failure. This corresponds to the project's application logic and meets robustness targets. A distinction will be made between planned and unplanned failures and cases in which a solution for resolving the failure is known and those in which it is not. Failures are distinguished either by creating "families" of exceptions, or by coding error numbers.

Justification

Enhances reliability

Example

In C++ and JAVA

Exception levels will be defined using the heritage mechanism and by filtering errors using well-organised catch blocks.

In C

An integer should be used to code errors with the following rules:

```
errno < 1024 => system error
1024 <= errno < 2048 => input output error application level
Etc.
```

Err.Impression	The possibility of reporting an error message must be studied.
M=0;R=1;P=99;V=1	
Any	

Description

An error message may be reported using a trace, print, creation of an error file, use of an error window or console, or the use of a dedicated error peripheral

Justification

Facilitates fine tuning.

Example

In SHELL and PERL
Errors are recorded in a log file.

Err.Nom	An error process or an exception must have a name that expresses the reason for which the service requested may not be provided.
M=0;R=1;P=95;V=0	
Any	

Description

This rule mainly concerns exception languages. For other languages, meaningful symbols may be defined for each "error code".

Justification

Enhances readability.

Example

In ADA:

```
package THE_LISTS is
type A_LIST is limited private ;
    LIST_SATURATED: exception;
    -- Lifted when the list is saturated at creation or insertion.
end THE_LISTS;
```

In SHELL

Scripts will use the following abnormal end codes (the numerical value is indicated between parentheses):

```
BAD_ARGS (1)     Error in the number of arguments for a function
NO_FILE (2)      File access error
UNKNOWN (3)      All other errors.
```

Err.FinOperation	Error processing must be localised at the end of the operation.
M=1;R=1;P=60;V=1	
Any	

Description

In most cases, when an exception is flagged, the operation is stopped. Processing of all exceptions that may be flagged must be performed at the end of the operation, rather than by nested blocks, each of which manages its own exceptions.

Justification

This rule enhances code readability. The nominal algorithm is not polluted by error processing, and processing that is common to several exceptions may be factorised at the end of the operation.

Example

Not Applicable

Err.Operation	Error processing must be performed at the level of the operation that may process this error.
M=1;R=3;P=21;V=0	
Any	

Description

In exception languages, an exception must not be recovered by a function that does not have the resources to process it.

Justification

Processing an error too early weighs down the code unnecessarily.
If processed too early, the programmer risks forgetting to propagate the error one level up, where it may be processed.

Example

In C++ or JAVA

Incorrect example:

```
// method1 can send the exception MyException.
void method1() throws MyException {
    if (...) {
        throw new MyException();
    }
}
// method2 calls method1 but does not know how to process
// MyException.
void method2() {
    try {
        method1();
    } catch (MyException e) {
        // Simple trace, no processing of the error.
    }
}
```

Correct example:

```
void method1() throws MyException {
    if (...) {
```

```
            throw new MyException();
        }
    }

    // Declare that method2 can return the exception.
    void method2() throws MyException {
        // If method1 tags the exception MyException
        // it is propagated to caller of method2.
        method1();
    }
```

Err.IntegriteDonnee	Error triggering must not modify data integrity.
M=1;R=3;P=22;V=0	
Any	

Description

With OO languages, for objects with non-trivial construction, an exception thrown during modification of an object can lead to non-integrity.

Justification

Lack of data integrity causes serious problems or unpredictable operation.

Example

In C++:

An example in which a failed allocation during an assignment operation destroys data integrity

```
class  String {
private:
     char * string ;
public:

     String (const char *s) ;
     String (String & s) ;
     ~String () ;

     // Assignment operator
     String & operator= (const String & s) ;
};

String::String (const char * s) {
     int size = strlen (s) + 1 ;
     string = new char [size] ;
     strcpy (string,s);
}

String & String::operator= (const String & s) {
     if (this!=&s) {
           delete [] string ;
           int size = strlen (s.string) + 1 ;
           string = new char [size] ;
           // if the allocation fails,
           // string points to an area that has just been
           // freed (and may therefore be reused later)
           strcpy (string,s.string);
```

```
        }
        return *this ;
    }
```

To avoid this problem, each call to new must be placed in a try catch block; in the event of an error, string must be assigned 0; the string will then be verified to be different from 0 before each legitimate operation, including destruction.

Err.ToutesTraitées	All errors must be processed. No errors must be masked or ignored: error triggering must never abruptly interrupt the program.
M=0;R=3;P=42;V=0	
Any	

Description

To enhance application robustness, the application must receive all signals (or interruptions) and define a case-by-case processing strategy according to the need and nature of the signal.

Justification

An unprocessed exception leads to the program being abruptly stopped, which is never desirable.

Example

In general: the case of numeric exceptions

In the specific case of the arithmetic processor, all numeric exceptions must be examined. Certain exceptions may be masked (generally rounding and underflow exceptions) with a justification. Unmasked exceptions must be associated with dedicated software processing.

In C++

A non-processed exception is returned to the highest level, and causes untreated exception handlers to be triggered. Two handlers exist: "terminate", which mandatorily terminates the ongoing execution, and "unexpected" which may allow the current exception (not processed) to be rerouted to a processed exception. These handlers may be redefined using customised functions: this may be useful in attaining ultimate robustness or to process very general exceptions at a high level. It must not replace local processing of exceptions. The precise functioning and the connections between the two handlers "terminate" and "unexpected" generally depend on the application context: their respective behaviour should therefore be carefully analysed when being used.

```
Example of redefining "terminate"
void MyEnd () {
     cerr << "No processed exception\n" ;
     exit (-1) ;
}

void End () throw (char *) {
     // Modification of termination handler
     terminate_handler previousTerminate = set_terminate (MyEnd) ;

     // Code
     if (…) throw "Exception" ;

     // Recovering standard handler
     set_terminate (previousTerminate);
}
```

```
Example of redefining "unexpected"
void MyEnd2 () throw (int) {
     cerr << "Unexpected exception\n" ;
     throw (1);
}

void End2C () throw (char *) {
     // Modification of termination handler
     unexpected_handler previousUnexpected = set_unexpected (MyEnd2) ;

     // Code
     if (…) throw "Exception" ;

     // Recovering standard handler
     set_unexpected (previousUnexpected);
}

void End2 () {
     try {
           End2C ();
     }
     catch (int e) {
           cerr << "Interception " << e << endl ;
     }
}
```

Err.Canal	Error messages must be sent to the user via a dedicated input output channel, when this exists in the language. If it does not exist, a dedicated channel must be created for this purpose.
M=1;R=0 ;P=114;V=1	
Any	

Description

The channel may be any type of communication resource: a logic channel, file, console, etc.

Justification

Improves consistency in error reporting
Enhances reliability

Example

In SHELL

The error descriptor 2 is used, which corresponds to a standard error file. The user can ask that normal display be redirected in the file, while maintaining error display on-screen.

```
if a_test_that_must_succeed
then
   # Any operations
else
   print 'The test_that_must_succeed has failed!' >&2
fi
```

7.7. DYNAMIC

Dyn.OS	Task and thread management mechanisms offered by the operating system and/or the real-time kernel must be carefully analysed. Decisions regarding the use or non-use of each mechanism must be carefully discussed.
M=1;R=1;P=54;V=0	
Any	

Description

The programming environment generally offers specialised classes and services to allow multi-tasking and multi-threading to be managed. In particular, classes for managing mutual exclusion and services for inhibiting preemption, etc. are offered.
Compilation options also allow the generated code to be customised: for example, to verify or ensure that one variable will not be shared by various threads.

Justification

Multi-thread programming is highly context-specific.

Example

In IDL

On multi-processor machines, IDL authorises multi-threading, which increases calculation speed by simultaneously using the available processors. IDL automatically assesses the calculations performed by the various routines and decides which will benefit from multi-threading, according to the following parameters:

Number of elements concerned,
Processor availability,
Availability of a multi-threaded version of the routine used.

Only a certain number of IDL instructions have a multi-threaded version, and may as a result benefit from multi-threading. To obtain this list, refer to IDL online help under the heading "Services that use the thread pool".

Dyn.AttenteActive	No tasks or threads should have active waiting.
M=0;R=1;P=96;V=0	
Any	

Description

An active loop is defined here as a permanent loop around a scanning or processing activity that is never suspended by standby or waiting.
Tasks with active loops are prohibited.

Justification

Tasks with active loops are permanently active and risk monopolising the CPU at the expense of other tasks, and possibly freezing the application.
The correct operation of a program that does not respect this rule depends on the behaviour of its computer and its operating system when managing tasks. It is conditioned by the number of processors, priority management and the time share used.
The reliability of this type of program is uncertain, and relies on the machine that executes it.

Example

Not Applicable

Dyn.Abort	A program must never be abruptly ended by a task or thread termination instruction (such as exit or abort).
M=0;R=1;P=91;V=1	
Any	

Description

When a termination instruction is executed, this causes the task to be abruptly stopped. The resources used by the task may be in an incoherent state; tasks that depend on these resources may be abruptly aborted as a result.

Justification

Enhances program reliability, especially as regards data and processing consistency

Example

In ADA

The ***abort*** instruction is not effectively executed when the instruction is read, but rather is delayed until a "check-point", such as the beginning or end of a process, select instruction, ***delay***, ... This delay cannot be controlled and may lead to unexpected behaviour.

Dyn.PrioRelatives	Absolute priorities must not be used for tasks and threads, but rather relative priorities.
M=1;R=1;P=55;V=1	
Any	

Description

Real-time architecture must be designed without impacting the task sequencing algorithm. Respecting this rule guarantees application portability.

Justification

Enhances the efficiency of multi-task programs
Enhances the portability of multi-task programs

Example

In ADA

No ***Priority*** pragma is used to manage task synchronisation.
In certain real-time applications, the ***priority*** pragma may be authorised in order to optimise computer resources.

Dyn.Ressources	The resources allocated in a thread must be freed in this same thread.
M=0;R=3;P=38;V=0	
Any	

Description

For a multi-threading support, many compilers verify this point.
Exception: if a thread is written to implement an instance factory: this thread will be solely in charge of building instances, which are destroyed by other threads.

Justification

Enhances the reliability of multi-task applications

Example

In C++
When programming in *Windows* , the allocation of a COM/OLE object by a thread must be freed by the thread.

Dyn.SectionCritique	The creation and initialisation of tasks or threads must be encapsulated; they must be performed in a critical section, without any possibility of being interrupted.
M=0;R=3;P=37;V=0	
Any	

Description

Not applicable

Justification

No events must disturb task creation and initialisation.

Example

In JAVA
The start() method should be called inside the class.
Incorrect example:

```
// Implementation  of the Runnable interface.
class Display implements Runnable {
    ...
    public void run() {
        while (true) {
           // Draw.
           ...
           repaint();
        }
    }
}

// In another class, creation of the interface.
Drawing displayed = new Display("Christmas Tree");

// Creation of the object Thread.
Thread myThread = new Thread(drawing);   // no encapsulation

// Activation of thread.
myThread.start();                      // no encapsulation
...
```

Correct example:

```
// Implementation of the Runnable interface.
class Animation implements Runnable {

    // Private attribute used to store a thread
```

```
        // identifier.
        private Thread myThread;

        // Creation of a Thread object and activation of the thread.
        // Initialisations in the constructor.
        Animation(String name) {             // Scope of the constructor is
                                             // not specified.
            myThread = new Thread(this);     // Creation of a Thread object
            myThread.start();                // activation of thread.
        }
        ...
    }
    // Creation of an animation.
    // The way in which the animation object is implemented does not appear
    // from the outside.
    Animation Hello = new Animation("Hello");
```

Dyn.Partage	Variable shared between threads should be carefully analysed.
M=1;R=3;P=23;V=0	
Any	

Description

Specifically, the resources (essentially variables) shared between the main thread and secondary threads should be analysed, as well as resources shared between secondary threads.
A thread is generally implemented by a function whose launch mode is asynchronous: this rule means that data that is local to this function or to functions called by this function may be handled in a thread.
In handling of variables shared between threads, the keyword volatile should be used to inhibit compiler optimisation relating to the recognition of sub-expressions: note, however, that this attribute does not guarantee data integrity. This declaration must be completed by using semaphores or *mutex*.

Justification

Non-synchronisation between threads may lead to incoherent results or unexpected behaviour concerning the shared resources. This non-synchronisation may be very critical during the allocation or deallocation of these resources.

Example

Not Applicable

7.8 INTERFACES

Int.ExistenceFichier M=0;R=2;P=66;V=1 Any	The existence or non-existence of a file must always be verified before the file is opened or created; actions to be performed in the event of failure must be provided for.

Description

Read- or write-access to a file may be prevented for several reasons.

Justification

Enhances reliability

Example

Not Applicable

Int.CheminFichier M=2;R=0 ;P=113;V=1 Any	The access path to any file must be parameterised.

Description

The file access path may be placed in an environment variable, but other parameterisation resources may also be used (parameter files, etc.).

Justification

Facilitates upgrading.

Example

In C

```
#include   <stdlib.h>
char *Name_Directory;
Name_Directory = getenv ("REP_FILE_CONF"); // recovery of full
// path to directory containing
// configuration files
```

Int.CheminAbsolu M=2;R=1;P=56;V=0 Any	Access paths must not make any hypotheses on the the current directory.

Description

The current directory is volatile information.

Justification

Improves reliability and portability.

Example

In C

Paths to includes are independent of the file location or compilation directory

In SHELL

The current directory ('.') must never be in the search path used by a SHELL program

Int.Environement	Elements relating to program installation must be designated using specific environment variables
M=2;R=0;P=83;V=0	
Any	

Description

Not applicable

Justification

Enhances portability.

Example

In WAWE

The environment variable "WAVE_PATH" must be positioned outside of the application and must not be modified in services. This environment variable indicates the directory(ies) in which the modules and services that may be used by WAVE are located. These directories are scanned in the order of their appearance in "WAVE_PATH". This variable is comparable to the "PATH" environment variable used in UNIX.

Int.Temporaire	All temporary files created by the application must be located in dedicated areas and destroyed at the end of execution, at the latest.
M=0;R=1;P=92;V=0	
Any	

Description

Program execution must not pollute disk space.
Temporary files should be destroyed as soon as possible, especially if they are large.

Justification

Disk space is finite; it is essential that this space be conserved.

Example

Not Applicable

Int.FichierFermeture	All open files must be closed at the same algorithmic level: module, class, operation.
M=0;R=1;P=93;V=0	
Any	

Description

Not Applicable

Justification

This rule allows file opening and closing operations to be grouped together in the same service to ensure that the file has effectively been closed.
File closing is important, as it frees logic units and allows other files to be opened (the number of available logic units is limited).

Example

In PVWAWE

Files can be closed using either the "CLOSE" function or the "FREE_LUN" function, depending on the opening mode.
Processing associated with files (reading, writing) may be performed in other called services. Only OPEN and CLOSE operations must be performed in the same service.

Int.GrouperES	Input/output instructions of the same type must be grouped together.
M=1;R=0;P=112;V=0	
Any	

Description

It is better to have fewer long I/O instructions, rather than numerous short I/O instructions.

Justification

The surplus regarding the operating system kernel function calls penalises the system.

Example

In C

```
// incorrect
printf(" x = %f", Var_X);
printf(" y = %f", Var_Y);

// correct
printf("x = %f y = %f", Var_X, Var_Y);
```

7.9. QUALITY

Qa.Ressources	The software must be free of user interface details by using separate graphical resources
M=1;R=0;P=111;V=0	
Any	

Description

The graphical capacities of the target machines should not be assumed. When building a GUI, the graphical attribute values for the hardware platform should never be set.

Justification

Enhances portability.

Example

In general:

When building an GUI, the character font or font size should never be set.

In JAVA:

To determine the list of fonts available, use:

```
java.awt.Toolkit.getFontList()
```

To determine character font size, use:

```
Font.getFontMetrics()
Graphics.getFontMetrics()
```

For example:

```
String fontCourier = ... // Not set.
titleFont = new java.awt.Font(fontCourier, Font.BOLD, 12);
titleFontMetrics = getFontMetrics(titleFont);
```

In PVWAWE:

Assign font sizes and colours in a resource file.
Use constants to define widget size in pixels (size, position etc.).
Use constants to allocate resources.
Use constants to locate the position of menu items.

Qa.PortType	The portability of base types should always be a concern.
M=1;R=0;P=110;V=1	
Any	

Description

Base types (numeric, characters) generally depend on the machine or environment in which they are executed.

Justification

Enhances portability.

Example

In C:

Base types (`int`, `float`) should not be used as is. The physical size of the `int`, type integer depends on the target machine. In general, it corresponds to the most natural size on the target machine.

In ADA:

INTEGER sub-types will be defined.
In addition, a distinct type will be defined for each group of quantifiable entity, with the appropriate application constraints. As a result, the type is implemented correctly regardless of the machine. If we want the type to be represented identically (16, 32, 64, ... bits) on all machines, a representation clause must be added.

```
--  First unauthorised example
procedure COUNT_AIRCRAFT is
NO_OF_AIRCRAFT: INTEGER := 10 ;
begin
     ...
end COUNT_AIRCRAFT;


--  Second authorised example: be certain to define
--  application type only for counting aircraft
procedure COUNT_AIRCRAFT is
MAX_NO_OF_OBJECTS: constant := 100;
type AN_AIRCRAFT_COUNTER is range 0.. MAX_NO_OF_OBJECTS;
NO_OF_AIRCRAFT: AN_AIRCRAFT_COUNTER := 10 ;
begin
     ...
end COUNT_AIRCRAFT;
```

Qa.RepérerPort	The non-standard or non-portable elements used must be identified and program functioning must be adapted if need be.
M=1;R=0;P=108;V=0	
Any	

Description

Programs that must be executed on more than one target must detect and adapt themselves to targets.

Justification

Enhances portability.

Example

In SHELL

Many aspects of script functioning may be altered, depending on the operating system executing the program. For a script to be portable, these dependencies must be factorised as much as possible and isolated in a specific initialisation block. It may be wise to create a special initialisation/configuration file containing these dependencies. This file will concerns the entire project and will be read by each script.

Here is how a script might begin:

```
version=`uname -r | cut -d. -f1`
case $version in
   5) # Initialisations Solaris 2.x
      …
   4) # Initialisations SunOS
```

```
         …
      *) echo Type of system `uname -a` unknown
         exit 1
         …
   esac
```

Qa.TestRetour	Function return must be systematically tested, specifically system function return.
M=0;R=2;P=67;V=0	
Any	

Description

A function call must never appear as an independent instruction. A function must never be used only for its side effects.

Justification

The role of a function is to provide a value in the assessment of an expression. The programmer must use this function return value: if this is not the case, the programmer must use a procedure and not a function.

Example

In C:

```
// correct
State = Control_State (Var_X, Var_Y, Var_Z);
(void) printf ("State =%d", State); // Tolerated for this type of
function

// incorrect
(void) Control_State (Var_X, Var_Y, Var_Z);
// or
Control_State (Var_X, Var_Y, Var_Z);
```

Qa.Branches	In conditional instructions, the most frequent and most simple branches must be processed before the others, in order to enhance performance.
M=1;R=0;P=107;V=0	
On-board	

Description

Multiple choice type instructions verify the case according to its order of appearance in the block. Frequent and simple cases should therefore be placed before rare and complex cases.
If the portion of code concerned is not critical in regard to execution time, logical case order should be used (see Tr.Choix)

Justification

Improves execution time

Example

Not Applicable

Qa.Performances	Effectiveness can be enhanced by studying the possibilities made available by the development environment (compiler, performance analysis tools, etc.)
M=1;R=0;P=106;V=0	
Any	

Description

"*Profilers*" allow application execution to be traced. Analysis tools may then be used to examine the parts of the application that use the most resources (memory or execution time). Some software allow these analyses to be performed automatically by ensuring distributed process monitoring.
In most cases, 90% of execution time is consumed by only 10% of a program, and this, most often in areas in which we might least expect. Optimisation efforts must thus be concentrated here.

Justification

Improves effectiveness

Example

In C++:

Compilers generally propose specific options for function management. In particular:
call management (*fast call*, use of directories rather than the stack for non-recursive functions, short calls, etc.),
inline management: inhibition of expansions on option, on condition; seeking functions to be automatically put inline, etc.,
pointer representation mode to virtual member functions: previous definition of pointers, acknowledgement of multiple heritage, etc.

In JAVA:

The interpreter prof option creates a profile file in the current directory, which can subsequently be used.

Qa.Pile	Stack consumption as compared to available quantities must be carefully studied. In particular: local data, parameters and call tree depth.
M=0;R=3;P=42;V=1	
On-board	

Description

When the size allocated to the stack is critical, it may be preferable to work by side effect on global variables rather than using local data or parameter passing, which will create stack consumption. This choice must be guided by careful analysis of the call tree depth, in "worst case" type scenarios.
In C, C++ and ADA, an address or reference passing mode may also be used for data that occupies significant memory space. This would allow space to be saved in the stack (the size of the data address is smaller than the data itself).

Justification

Improves effectiveness: correct stack sizing allows RAM resources to be optimised
Enhances reliability by avoiding stack overflow

Example

Not Applicable

Qa.Interruptions	Software processing dedicated to accounting for hardware interruptions must be as brief as possible.
M=0;R=1;P=93;V=0	
Any	

Description

This software process is called an *interruption handler*. Each interruption is associated with a *handler* or *IT processing*. Interruption acknowledgement blocks the acknowledgement of other interruptions (not considering re-entry), which may be events that are important and which must be handled in a timely fashion. Therefore, in order to avoid monopolising the processor, processing times for interruptions must be kept to a minimum, and important tasks must be moved to the application excluding interruptions.

Justification

Improves response time for real-time applications.

Example

Not Applicable

Qa.Factorisation	Execution time-consuming sub-expressions must only be assessed once.
M=0;R=1;P=97;V=0	
Any	

Description

Arithmetic expressions must be factorised to the greatest extent possible; invariants must also be removed from loops.

Justification

Improves effectiveness

Example

In ADA

```
Y = 3*X*X + 2*X => 5 operations
Y = X * (3*X + 2) => 4 operations
```

Qa.Algèbre	Algebraic identities which can accelerate calculation must be used.
M=0;R=1;P=98;V=0	
Any	

Description

Algebraic identities may be used judiciously to facilitate certain calculations.

Justification

Improves effectiveness

Example

In general:

In searching for the point closest to point (X0, Y0), searching for the point that minimises the expression (X1-X0)^2 + (Y1-Y0)^2 is sufficient ; calculating the square root is not necessary.

Qa.Correlation	Correlated quantities must be calculated simultaneously.
M=1;R=1;P=57;V=0	
Any	

Description

Group together all calculations relating to the same problem.

Justification

Improves effectiveness

Example

In IDL:

IDL includes routines to simultaneously calculate correlated quantities.
The maximum value of the array table is calculated, and the minimum value is simultaneously calculated as well.

```
MaxValue = MAX(array, MIN = minValue)
```

Qa.ReutValide	Only validated services may be reused.
M=3;R=2;P=14;V=0	
Any	

Description

Only standard, validated and up-to-date components must be reused.

Justification

Enhances portability.

Example

In JAVA:

Use JFC - *Java Foundation Class* - packages from Sun, which contains among other things the Swing graphical interface classes as well as an API Java 2D implementation.
Libraries may be re-written for very specific project constraints.
Example: This may be the case for the library of basic mathematical functions other than those provided with the target arithmetic processor. This allows the application to effectively manage:
numeric behaviour (which is independent of the machine, host or target)
the number of nesting levels in the call tree (thereby minimising the size of the execution stack)
performance in terms of calculation time.

In PERL:

For developments that require the current or future possibility of being executed on various OS, direct system calls should not be performed, but rather, Perl primitives should be used as much as possible, as they are generally more portable.

```
# Do not write:
print `whoami`;
# but:
print getlogin;
```

Qa.OptionsCompil	With a compiled language, the compilation options that will highlight a maximum number of compilation warnings should be used. Each unresolved warning must be justified.
M=1;R=2;P=43;V=1	
Any	

Description

By default, compilers do not provide the maximum number of compilation warnings.

Justification

Enhances robustness.

Example

In C

Example: Using the option –Wall in the `gcc` compiler specifically locates problems that are difficult to debug: implicit conversion between signed and unsigned values or the illicit use of an allocation in a test.

In PERL:

PERL provides the programmer with a way of being warned when he performs coding operations that are not recommended (or prohibited). There are two ways to activate this check: the first is by using the "`-w`" option passed to the Perl interpreter, and the second solution involves the use of "`use warnings`".

Qa.Instrumentation	Code instrumentation (code, assertions) must be created using dedicated operations. If the language does not offer these operations, a specific module concerning them must be created.
M=2;R=1;P=49;V=1	
Any	

Description

Code instrumentation allows the rule Tr.ProgDefensive to be applied.
In the specific case of on-board software, special care will be taken when creating instrumentation.

Justification

Enhances robustness.

Example

In on-board C

Macros defined in the file "lice.h" are used for the Myriade line.

8. SUMMARY

8.1. RULE SUMMARY TABLE

The rules are summarised below, in alphabetical order.

Id. Rule	Title	Page
Don.AllocDynbord	Dynamic memory allocation is prohibited.	35
Don.AllocDynSol	If the language supports the concept, dynamic memory allocation must be used sparingly, and with caution.	36
Don.AllocEchec	If the language supports the concept of dynamic allocation, the potential failure of a memory allocation request must be systematically provided for.	36
Donc.AllocErreur	An error that occurs during processing must not cause memory to not be freed.	37
Don.AllocLiberation	All allocated memory must be freed at the same conceptual level.	37
Don.ChaineOper	Global operations for character strings (initialisation, copy, duplication, comparison, search, modification) must be performed using standard primitives provided by the language, when they exist.	35
Don.Declaration	All data used must be explicitly declared	27
Don.Enumeration	The use of constants or symbols must be preferred (enumerative, if the language allows) over the use of whole numerical data. The use of whole numerical data must be essentially limited to simple calculation or counting.	30
Don.Homonymie	The use of homonyms must be avoided except in cases of overload or explicit redefinition.	31
Don.Initialisation	Variables must be initialised before being used for the first time.	32
Don.Invariant	Constants must be defined for entities whose value is invariant.	30
Don.Localite	Local data declarations are preferred over more global declarations: data that is local to a module are preferable to global data, formal parameters are preferable to global data, local data for an operation are preferred over module-level data, and local data for an instruction block are preferable to local data for an operation.	29
Don.LocalUnique	Each local datum must have a unique use.	33
Data.PointeurNonAff	If the language supports the pointer concept, when a pointer is not associated with a specific object at declaration, a comment must specify the object that will be associated with it and, if the language allows, initialise it to null.	32
Don.Separee	Each piece of data must have a separate declaration.	27
Don.Structure	When a conceptual object must be implemented as several data, this data must be grouped in a structuring entity (class, structure, record, type) according to the possibilities provided by the language.	31
Don.TableOper	Global operations for tables (initialisation, copy, duplication, comparison) must be performed using standard primitives provided by the language, when they exist.	34
Don.TablePrincipe	The processing principal (line x column or column x line) for double entry tables must be defined.	33
Don.Typage	Data must be systematically and explicitly typed.	28
Don.TypeAnonyme	Anonymous types must not be used.	28
Don.Utilisee	All data that is defined must be used; a datum that is no longer used must be deleted.	33
Dyn.Abort	A program must never be abruptly ended by a task or thread termination instruction (such as exit or abort).	59

Id. Rule	Title	Page
Tr.ParSortie	All output parameters for a procedure must have received a value before the first processing condition, by initialisation by default, if necessary. The same is true for variables used to return the value of a function.	51
Tr.ProgDefensive	Defensive programming, which involves the use of pre-conditions and post-conditions, should be preferred.	45
Tr.RecursifBord	Recursivity is prohibited.	44
Tr.RecursifSol	Recursive operations must not be used unless they are conceptually simpler than an equivalent iterative operation.	43
Tr.Residus	No programming residue must exist as comments in the code: an instruction that is no longer used must be deleted.	46
Tr.TestEgalite	Use of the equality or difference test must be replaced by inequality where possible.	38

8.2. "COMMON" TRACEABILITY

This table provides the correspondence between the rules set out in this document and the rules found in language manuals. It contains the same number of lines as rules in this document, and as many columns as there are language manuals. The cells are empty if the common rule is not mentioned in the language manual; otherwise, cells contain the list of rules in the language manual which are covered by the common rule.

Rule / Language (Version)	ADA (5)	ON-BOARD ADA (3)	C (6)	ON-BOARD C (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Don.AllocDynbord		MEM (1)		EMBED-malloc								
Don.AllocDynSol	Memory.Allocation											
Don.AllocEchec							Excep.Allocation					
Donc.AllocErreur							Excep.Free					
Don.AllocLiberation			CO.DAT-FreeDyn								MEM (1)	Pointer.FREE-MEMORY;Routines.VARIABLESHEAP
Don.ChaineOper										Prog_CompStr_1		
Don.Declaration			CO.DAT-Vis	CO.DAT-Vis	CO.DAT(1);CO.DAT(4)	DECL(1)				Prog_DeclVar_1		
Don.Enumeration	Types.Enumerated	DECL(7)		CO.DAT-Lit	CO.DAT(1);CO.DAT(2)	DATA(1)	Const.Define;Semantic.Enum	MAINT-Const				Constant.DEFINITION
Don.Homonymie	Identifiers.Homonyms	IDENT (10)	CO.DAT-VarRedef			NAME(2)						Routines.UNIQUE-NAME
Don.Initialisation	Variables.Initialisation	DECL (9)	CO.DAT-IniVar	CO.DAT-IniVar	CO.DAT(5)	DATA(7)	Data.InitLocal	VARLOC-Initialisation	ENV-5		COMMON(3)	
Don.Invariant	Constants.Definition	DECL (10)		CO.DAT-Lit		DATA(3)	Const.Literal					Expression.INVARIANT;Constant.DEFINITION
Don.Localite	Variables.Block	MISC(6)				DECL(10);DECL(14)	Data.Local;Data.Proximity	VARLOC-Proximity;OPTIM-AccessVars	VAR-5;ENV-1		COMMON(4)	CommonBlock.AVOID;;CommonBlock.POSITIONAL-PARAMETER-1
Don.LocalUnique								VARLOC-Utilisation				
Data.PointeurNonAff						DECL(13)						
Don.Separee							Data.DeclSeparate	VARLOC-Line				
Don.Structure			CO.DAT-Lit;CO.DAT-TypConst			DATA(5)						Structure.MINIMISATION
Don.TableOper	Instructions.CopyTables											Table.EQUALITY
Don.TablePrincipe					CO.DAT(9)						TABLE(5)	Table. PATH
Don.Typage			CO.DAT-TypVar	EMBED-types-use		DECL(2)						
Don.TypeAnonyme	Types.Declaration	DECL (1)	CO.TY-Def;CO.TY-CompLit									
Don.Utilisee					CO.DAT(6)	DATA(9)				Prog_InitVar_1		
Dyn.Abort	Tasks.Abort	TASK (8)			CD.SG(3)							
Dyn.AttenteActive	Tasks.ActiveLoop	TASK (6)										
Dyn.OS		KERNEL (5)					Thread.Configuration					Routines.MULTITHREADING
Dyn.Partage							Thread.Sharing					
Dyn.PrioRelatives	Tasks.Priority	TASK (3)										
Dyn.Ressources							Thread.Resources					

Rule / Language (Version)	ADA (5)	ON-BOARD ADA (3)	C (6)	ON-BOARD C (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Dyn.SectionCritique		KERNEL (4)						THREAD-Encapsulate				
Err.Canal									I/O-5			
Err.FinOperation	Exceptions.Regrouping											
Err.Impression	Exceptions.Failure	EXCEPT (1)								Prog_TraceErr_1		
Err.IntegriteDonnee							Excep.Integrity					
Err.Mecanisme	Exceptions.Failure	EXCEPT (5)	CD.PRO-ErrMgt			EXEP(1)	Excep.Strategy	METH-Return	ERR-1;ERR-2;ERR-3	Prog_CodeErr_1	ERROR(1);ERROR(2)	Routines.REPORT; Errors.ON_ERROR-ON_IOERROR;Errors.CATCH
Err.Nom	Identifiers.Exceptions	IDENT (9)										
Err.Operation		INSTR (5) ;EXCEPT (3)					Excep.Proce	EXCEP-CatchUse				
Err.ToutesTraitées				EMBED-Interruptions			Excep.Terminate					
Err.TraitementDiff	Exceptions.Failure	EXCEPT (1)										
Id.ClasseType	Identifiers.Types	IDENT (2)	CO.TY-NameTyp					NAME-Class			STRUCT(8)	CommonBlock.NAMING
Id.ConstSignif	Identifiers.Constants	DECL (11)	CO.DAT-NameConst;CD.PP-NameMacro					NAME-Constant				
Id.Fonction	Identifiers.Functions	IDENT (6)	CP.SG-NameFunc					NAME-AccessAttribute	FILE-3	Name_IdFunc_1		Routines.NAMING
Id.IdentRegle	Identifiers.Underscore	IDENT (2) ; IDENT (3) ;IDENT (12) ; FILE(2)			CO.PRE(6)	NAME(4)	Name.General	STYLE-Language; NAME-Default		Name_idCons_1		CommonBlock.NAMING
Id.IdentSignif	Identifiers.Naming; Identifiers.Descriptiveness	IDENT (1)			CO.PRE(6)		Name.General	NAME-Explicit		Name_DescriId_1	STRUCT(7)	
Id.NomDonnee												
Id.NomParFormel	Identifiers.FormalParam	IDENT (13)										
Id.Pointeur	Identifiers.Pointers	IDENT (8)										
Id.Procedure	Identifiers.Procedures	IDENT (5)	CP.SG-FuncNam;CD.PP-MacroName				Name.PrivateData	NAME-AccessAttribute	FILE-3			Routines.NAMING
Id.Tache	Identifiers.Procedures	IDENT (5)										
Id.VarSignif	Identifiers.Variables	IDENT (4)	CO.DAT-VarName						VAR-2	Name_IdVar_1		Variable.NAMING1
Id.VarType	Identifiers.Variables		CO.DAT-VarName							Name_IdVar_1	STRUCT(8);STRUCT(9)	Variable.NAMING1
Int.CheminAbsolu									ENV-3			
Int.CheminFichier	File.AccessPath		CD.IO-FileParam									
Int.Environement								PORTAB-ConstFlat	ENV-4		STRUCT(11)	
Int.ExistenceFichier												
Int.FichierFermeture											I/O(3)	I_O.CLOSURE
Int.GrouperES			CO.IO-IOGroup		CO.IO(3)							
Int.Temporaire									I/O-1;I/O-2			
Org.Couplage	Packages.Linking		CO.DAT-NbGlobVar			MOD(2)					COMMON(2)	CommonBlock.SHARING
Org.DonneesOper	Packages.Design	PACKAGE (1)				DECL(5)		ORGANI-Package	FILE-5			

Rule / Language (Version)	ADA (5)	ON-BOARD ADA (3)	C (6)	ON-BOARD C (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Org.Duplication			CD.PP-InlineFunc	EMBED-Inline			Function.Inline;Meta.Techniques;Meta.Coding					
Org.Masquage	Packages.Specification	PACKAGE (3);RISQ (3)				MOD(4)	Encap.MemberData	CLASS-DataProtect				
Org.MatérielIndep			CD.DV-SeparPort			PORT(2)		PORTAB-InOutErr; PORTAB-GUICapa, PORTAB-Limit		Prog_SysSpec_1		
Org.Module					CD.SG(1)	PRES(4)	Orga.Order;Orga.PresFunc			Pres_OrgMod_1; Pres_OrgScript_1; Pres_OrgFunc_1	STRUCT(2);STRUCT(3):STRUCT(4);STRUCT(6)	Presentation.ROUTINE; Presentation.STRUCTURES-CONTROL; Presentation.FILE-BATCH;Presentation.MODULE
Org.ModuleNom	Identifiers.Packages;File.Naming	IDENT (7) ; FILE(2)	CP.SG-FileRole			NAME(1):MOD(1)	Name.Files		FILE-1	Name_IdMod_1; Name_IdScript_1	STRUCT(1)	Naming.SUFFIX
Org.MultiLang											COMM(5)	
Org.Principal					CD.SG(5)	PROG(1)						
Pr.Aeration	Presentation.Spacing	MISC(10)	CO.EX-UnaryOp;CO.EX-BinaryOp		CO.PRE(5)			DOC-Layout		Pres_Space_1;Pres_NoSpace_1;Pres_LineSep_1	STRUCT(6)	
Pr.CartDonnée			CO.PRE-CommVar					DOC-Layout			STRUCT(6)	
Pr.CartStd	Presentation.Header	PRES (7)	CP.PRE-Box		CO.PRE(1);CO.PRE(2)		Organ.Header	DOC-Layout	COMT-1;COMT-2	Pres_FuncHeader_1	STRUCT(6)	Presentation.ROUTINE
Pr.CommFonc	Comments.Autodoc;Comments.Interpretation;Types.Comments	COMM (1)	CO.PRE-CommFunc		CO.PRE(11)							Instruction.COMMENT
Pr.CommIdent	Comment.Indentation	COMM (2)	CO.PRE-CommIdent;CO.PRE-CommFBlo		CO.PRE(10)			DOC-Layout				
Pr.Indentation	Presentation.Indentation	PRES(1)	CO.PRE-Indent						FILE-6	Pres_Indent_1;Pres_Bracket_1;Pres_AlignCode_1;Pres_PosElse_1	STRUCT(6)	
Pr.Instruction	Presentation.SimpleInstr		CO.PRE-MultInstr;CO.PRO-InstrLim		CO.PRE(4);CO.PRE(7)	PRES(3)				Pres_LongLine_1		Expression.PRESENTATION; Presentation.STRUCTURES-CONTROL
Pr.LongLine	Presentation.LgLine;Presentation.Truncation	PRES (3);PRES (4)	CO.PRE-LineLim							Pres_LongLine_1; Pres_ComLong_1		
Qa.Algèbre												Expression.IDENTITY
Qa.Branches												Instruction.CASE/SWITCH-CLASSIFICATION
Qa.Correlation												Expression.REGROUPING
Qa.Factorisation								OPTIM-SubExpr; OPTIM-InvLoop				Expression.FACTORISING
Qa.Instrumentation Qa.Interruptions				EMBED-Proce_interrup								

Rule / Language (Version)	ADA (5)	ON-BOARD ADA (3)	C (6)	ON-BOARD C (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Qa.OptionsCompil				EMBED-OptionComp						Prog_UseWarn_1		
Qa.Performances							Function.Conf	OPTIM-9010				
Qa.Pile				EMBED-Stack;EMBED-NbArg;EMBED-StackSize;EMBED-VarAuto								
Qa.PortType	Types.Pre-defined	DECL (3)		EMBED-Types_int_float			Type.RedefTypeBase					
Qa.RepérerPort												
Qa.Ressources								PORTAB-GUIFonts			GUI(4);GUI(8);GUI(9);GUI(10)	
Qa.ReutValide			CP.DV-Reuse	EMBED-Library				PORTAB-DependAP		Prog_PortCallSys_1		
Qa.TestRetour		KERNEL (2)	CD.PRO-RetUse								STRUCT(16)	Routines.REPORT
Tr.Booleen	Expressions.ComplexCondition											
Tr.BoucleSortie	Instructions.Exit		CO.PRO-BreakLoop		CO.PRO(9)	FLC(5);FLC(7)			CTRL-3			Instruction.FOR-BREAK
Tr.CalculStatique	Expressions.Static	MISC(2)										
Tr.Choix	Instructions.MultipleChoice	INSTR (1)		EMBED-switch_case	CO.PRO(5)	FLC(3)						
Tr.ComparaisonStrict	Instructions.FloatingEquality		CO.DAT-CompFloat		CO.EX(2)	EXP(2)						Variable.EQUALITY
Tr.ComparConst			CO.PRE-CompConst									
Tr.ControleRacc	Expressions.ShortcutCtrl			EMBED-for								
Tr.DoubleNeg	Expressions.DbleNegations											
Tr.FonctionSortie	SubProgram.Return	SPROG(9)	CD.PRO-Exit1		CD.SG(3)						STRUCT(15)	Routines.EXIT
Tr.Goto	Instructions.Goto	INSTR (3)	CO.PRO-Goto		CO.PRO(6);CO.PRO(7);CO.PRO(5)	FLC(9)	Control.Goto				STRUCT(18)	Instruction.GOTO
Tr.MelangeType			CO.TY-Conv		CO.TY(4)	DATA(10)						
Tr.ModifCompteur			CO.PRO-ForInd		CO.PRO(9)	FLC(6)		CONTR-ForParam	CTRL-2	Prog_ModForeach_1		Instruction.FOR-CONSERVATION
Tr.ModifCondSortie			CO.PRO-ForCond		CO.PRO(9)	FLC(6)		CONTR-ForCondition				
Tr.ModifConst											TYPE(3)	Constant.CONSERVATION
Tr.ModifParSortie		SPROG(4)		EMBED-ArgValue ;EMBED-ArgAddress	CO.PA(1);CO.PA(6)		Function.ConstRefer				STRUCT(17)	SystemVariable.CONSERVATION;PositioningParameter.NATURE
Tr.ModifVarGlobal	SubProgr.Function; SubProgr.GlobalVar	MISC(5)			CO.PA(8)	PAS(9)			COMT-4		STRUCT(17)	SystemVariable.CONSERVATION;Routines.MODIFICATION-PARAMETER
Tr.OrdreChoix	Instructions.EnumerationChoice;Instructions.OtherChoice	INSTR (2)	!CO.PRO-DefaultCase			FLC(4);FLC(8)		!CONTR-Default	CTRL-1			
Tr.OrdreParFormel	SubProg.ParOrder	SPROG(1)		EMBED-ArgValue	CO.PA(3)	PAS(8)					PARAM(1)	

Rule / Language (Version)	ADA (5)	ON-BOARD ADA (3)	C (6)	ON-BOARD C (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Tr.ParamOptionnel	!SubProg.ParamBy Default	!SPROG (5)				SPRO(5)					!PARAM(3);!PARAM(4);PARAM(5)	
Tr.Parenthèses	Expressions.Priority Order	EXPR (1)			CO.EX(1)	EXP(1)					INSTR (1)	Expression.PARENTHESES
Tr.ParSortie	SubProg.OutValue	SPROG(6)										
Tr.ProgDefensive	SubProg.Defensive Tests						Function.PrePostCond		ARGS-3;CTRL-8		STRUCT(16)	
Tr.RecursifBord		RISQ (4)										
Tr.RecursifSol	SubProg.Recursivity					SPRO(4)						Routines.ITERATIVE
Tr.Residus					CD.SG(4)					Pres_DebugClean_1		
Tr.TestEgalite		MISC(8)										

STANDARDS REFERENCE PRODUCED BY:

Centre National d'Etudes Spatiales

Inspection Générale Direction de la Fonction Qualité

18 Avenue Edouard Belin

31401 TOULOUSE CEDEX 9

Tel.: +33 (0)5 61 27 31 31 - Fax: +33 (0)5 61 28 28 49

CENTRE NATIONAL D'ETUDES SPATIALES

Headquarters: 2 pl. Maurice Quentin 75039 Paris cedex 01 / Tel: +33 (0)1 44 76 75 00 / Fax: 01 44 46 76 76
Paris Trade & Companies Registry No. B 775 665 912 / Business Registration No: 775 665 912 00082 / Business Sector Code 731Z