RuleChecker & QualityChecker C++ Reference Manual

Before using this information, be sure to read the general information under “Notices” section, on page 145.

About This Manual

Audience

This manual is intended for Kalimetrix Logiscope™ *RuleChecker & QualityChecker* users for C++ source code verification.

Related Documents

Reading first the following manual is highly recommended:

- *Kalimetrix Logiscope - Basic Concepts.*
- *Kalimetrix Logiscope - RuleChecker & QualityChecker - Getting Started.*

Creating new scripts to check specific / non standard programming rules is addressed in dedicated document:

- *Kalimetrix Logiscope - Adding Java, Ada and C++ scriptable rules metrics and contexts.*

Overview

C++ Project Settings

Chapter 1 presents basic concepts of *Logiscope RuleChecker & QualityChecker C++* , its input and output data, its prerequisites and its limitations.

C++ Parsing Options

Chapter 2 describes the way to adapt *Logiscope RuleChecker & QualityChecker C++* to the application. It also specifies the specifics of the C++ dialects supported by *Logiscope RuleChecker & QualityChecker C++*

Command Line Mode

Chapter 3 specifies how to run *Logiscope RuleChecker & QualityChecker C++* using a command line interface.

Standard Metrics

Chapter 4 specifies the metrics computed by *Logiscope QualityChecker C++*.

Programming Rules

Chapter 5 specifies the programming rules checked by *Logiscope RuleChecker C++* .

Customizing Standard Rules and Rule Sets

Chapter 6 describes the way to modify standard predefined rules and to create new ones with *Logiscope RuleChecker C++*.

Conventions

The following typographical conventions are used:

bold	literals such as tool names (**studio**) and file extension (***.cpp**),
bold italics	literals such as type names (***integer***),
italics	names that are user-defined such as directory names (*log_installation_dir*), notes and documentation titles,
`typewriter`	file printouts.

Contacting Kalimetrix Software Support

If the self-help resources have not provided a resolution to your problem, you can contact KalimetrixSupport for assistance in resolving product issues.

Prequisites

To submit your problem to Kalimetrix Software Support, you must have an active support agreement. You can subscribe by visiting http://www.kalimetrix.com .

- To submit your problem online (from the KalimetrixWeb site) you need to be a registered user on the Kalimetrix Support Web site : http://support.kalimetrix.com/

Submitting problems

To submit your problem to Kalimetrix Software Support:

1) Determine the business impact of your problem. When you report a problem to Kalimetrix, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting.Use the following table to determine the severity level.

Severity	Description
Block	The problem has a *critical* business impact. You are unable to use the program, resulting in a critical impact on operation. This condition requires an immediate solution.
Crash	The problem has a *significant* business impact. The program is usable, but it is severely limited
Major	The problem has a *some* business impact. The program is usable, but less significant features (not critical to operation) are unavailable.
Minor	The problem has a *minimal* business impact. The problem causes little impact on operations or a reasonable circumvention to the problem was implemented.

2) Describe your problem and gather background information, When describing a problem to Kalimetrix, be as specific as possible. Include all relevant background information so that Kalimetrix Software Support

specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?

  To determine the exact product name and version, start your product, and click **Help > About** to see the offering name and version number.

- What is your operating system and version number (including any service packs or patches)?

  - Do you have logs, traces, and messages that are related to the problem symptoms?

  - Can you recreate the problem? If so, what steps do you perform to recreate the problem?

  - Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?

  - Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.

3) Submit your problem to Kalimetrix Software Support. You can submit your problem to Kalimetrix Software Support in the following ways:

   - **Online**: Go to the Kalimetrix Software Support Web site at http://support.kalimetrix.com

Table of Contents

Chapter 1

C++ Project Settings

A Logiscope project mainly consists in:

- the list of source files to be analysed,
- applicable source code parsing options according to the compilation environment,
- the verification modules to be activated on the source code files and the associated controls (e.g. metrics to be computed, rules to be checked).

A source file is a file containing C++ source code. This file is not necessarily compilable. It only has to conform to the C++ syntax.

Logiscope C++ projects can be created using:

- **Logiscope Studio Wizard:** a graphical interface requiring a user interaction, as described in the following sub-sections introducing the Logiscope C++ project settings,
- **Logiscope Create:** a tool to be used from a standalone command line or within makefiles, please refer to Chapter *Command Line Mode* to learn how to create a Logiscope project using **Logiscope Create**.

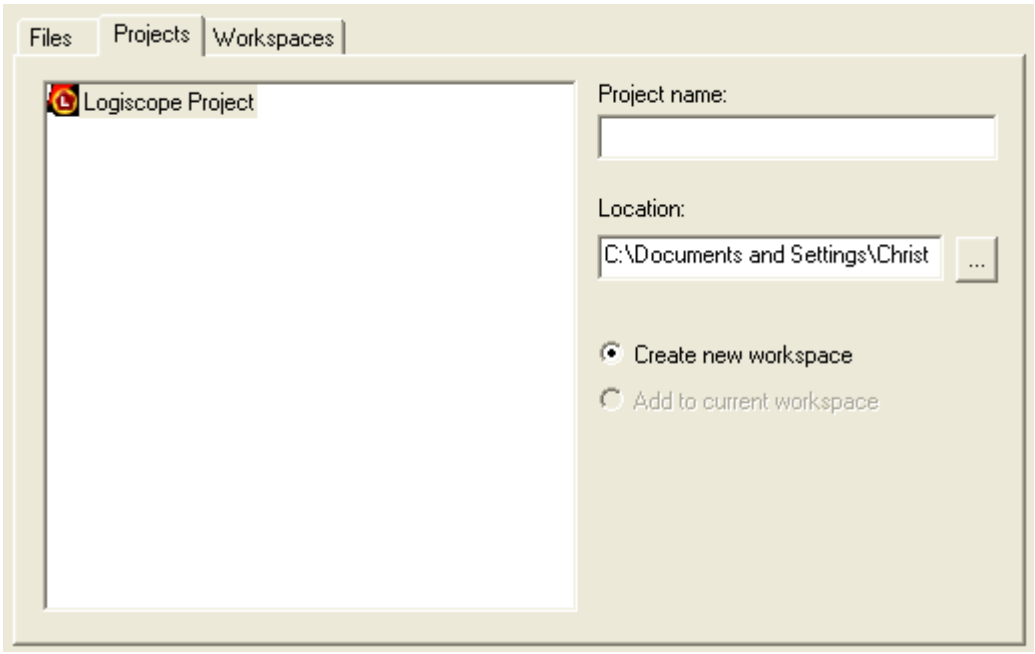
1.1 Starting a Logiscope Studio Session

To begin a Logiscope **Studio** session:

- On UNIX (i.e. Solaris or Linux):
  - launch the **vcs** binary .
- On Windows:
  - click the **Start** button and select the **Kalimetrix Logiscope <version>** item in the **Kalimetrix** Programs Group.

1.2 Creating a Logiscope Project

Once the Logiscope Studio main window is displayed, select the **New**... command in the File menu or click on the  icon, you get the following dialog box:



The **Project name:** pane allows to enter the name for the new Logiscope project to be created.

Location: allows to specify the directory where the Logiscope project and the associated Logiscope repository will be created. For more details, see the next section.

By default, the project name is automatically added to the specified location. This implies that a subdirectory named <ProjectName> is automatically created.

Defining the type of the Logiscope project

The following **Logiscope Project Definition** dialog box appears:

The **Project Language:** is the programming language in which are written the source code files to be analysed. Of course, select C++.

Note: Only one language can be selected. If your application contains source code files written in several languages e.g. C and C++ source files, you should create several distinct Logiscope projects: one for each language.

The **Project Modules:** lists the verification modules to be activated on the source files of the project .
For instance, you can select both **QualityChecker** and **RuleChecker**.

Notes: At least one module should be selected. The TestChecker module cannot be selected with an other module.

For more details on *TestChecker* module, please refer to *Kalimetrix Logiscope - TestChecker - Getting Started.*
For more details on *CodeReducer* module, please refer to *Kalimetrix Logiscope - CodeReducer - Getting Started.*

Specifying the source files to be analysed

The **Project Source Files** dialog box allows to specify what source files are to be analysed and where they are located.

Source files root directory shall specify the directory including all the source files to be analyzed.

If necessary, use the **Directories** choice to select the list of repertories covering the application source files.

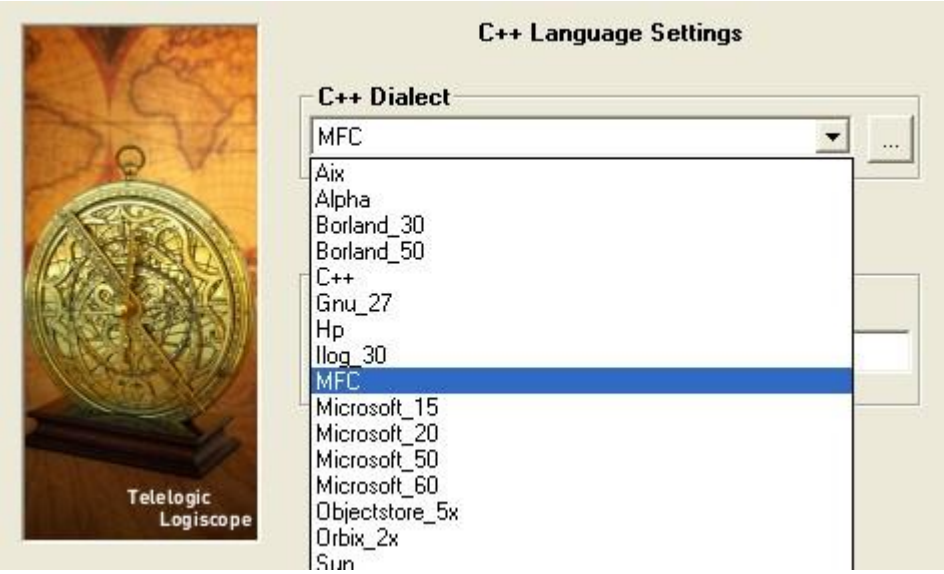
- **Include all subdirectories** means that selected files will be searched for in every sub-directory of the source file root directory.
- **Do not include subdirectories** means that only files included in the application directory will be selected.
- **Customize subdirectories to include** allows the user to select the list of directories that include application files through a new page.

Suffixes choices allow to specify applicable source, header and inline file extensions needed in the above selected directories. Extensions shall be separated with a semi-colon.

Setting Parsing Options

The next dialog box allows to set up C++ source code parsing options:

C++ Dialect: A dialect is used to specify parsing actions associated to some types, "keywords" according to the source code compiler specifics.
For more details on available dialects, please refer to the next chapter *Parsing Options.*



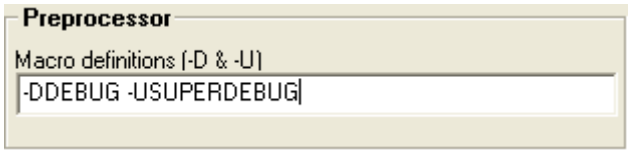
Preprocessor: The source code files to be analyzed may contain some preprocessing directives (e.g. `#ifdef`). In some cases, these directives can lead to parsing errors and warnings by breaking up the code structure.

Logiscope allows to parse C++ files taking into account part of the preprocessing directives.
In the Macro definitions (-D & -U) pane, you can define and or undefine some preprocessing options by respectively using:

- **`-D`**`<name>`: defines `<name>` as if it were in a `#define` directive.
- **`-U`**`<name>`: considers `<name>` as undefined as if it were part of an `#undef` directive.

The number of occurrences of option **-D** and/or option **-U** is unlimited.

In the example below, the `DEBUG` option is defined, so the corresponding conditional code will be parsed. The `SUPER_DEBUG` option is considered as undefined so the corresponding conditional options will not be parsed.



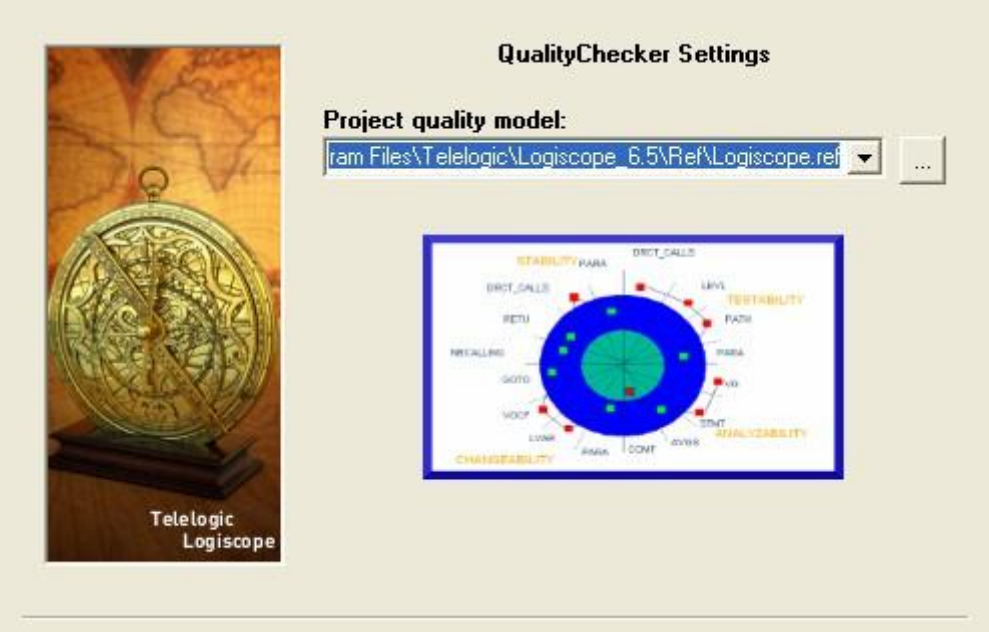
For more details on the **Preprocessor**: settings, please refer to the next chapter *Parsing Options.*

Setting QualityChecker Parameters

The next dialog box allows to specify the applicable **Project quality model:** how the *QualityChecker* module evaluates software quality characteristics (e.g. Maintainability) based on a standard factors / criteria / metrics approach.

Note: Quality models are textual files (also called Reference files). Default quality models are provided with the standard Logiscope installation. They should be customized to take into account the verification objectives and contexts applicable to the project.

For more information, see the *Kalimetrix Logiscope - Basic Concepts* manual.



For your project verification, you should define and select your own applicable quality model.

Setting RuleChecker Parameters

The **RuleChecker Settings** dialog box allows to specify the applicable **Project rule sets:** i.e. the rules / coding standards the *Logiscope RuleChecker* module shall verify on the project source files.

For more details on available rules and rule sets, please refer to the chapter *Standard Programming Rules.*

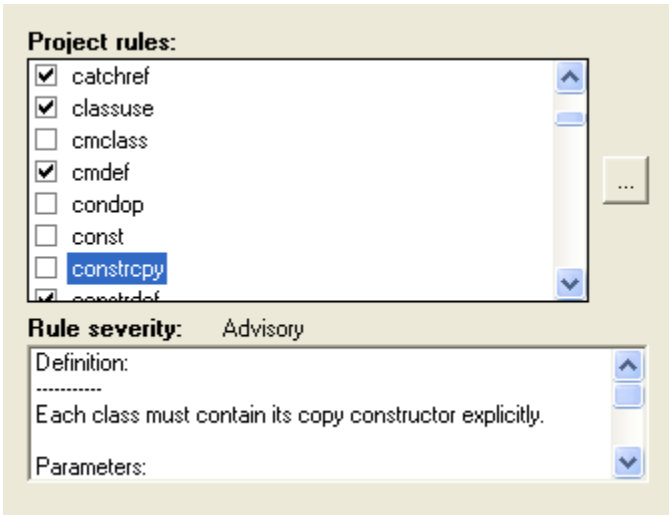
At least one rule set should be selected for the *Logiscope RuleChecker* projects.

Several rule sets can be selected. If so, *Logiscope RuleChecker* will check the union of the rules specified in all selected rule sets.

The next **RuleChecker Settings** dialog box allows to fine tune the list of **Project rules.** It is possible to select or unselect some of the rules available.

The rules that are selected are those listed in the Project rule sets selected in the previous **RuleChecker Settings** dialog box

You can check / uncheck the rules.The description of the selected rule and the rule severity are displayed in the bottom pane.



The last **RuleChecker Settings** dialog box allows to use some advanced features of the *Logiscope RuleChecker* module.

Relaxation mechanism: when the box is checked, rule violations can be relaxed using special comments in the code. For more details, please refer to the next section.

External violation import: when the box is checked, the files in the specified project folder can be used to import violations generated by an external tool.
For more details, please refer to the *Kalimetrix Logiscope - RuleChecker & QualityChecker - Getting Started* document.

Rule set file format: when the box is checked, the project rule set file (i.e. with a ".rst") extension) that is generated for the project doesn't contain any includes of other rule set files. It will contain an expanded copy of the contents of any rule sets that were used for the project.
For more details, please refer to the Chapter *Customizing Rules and Rule Sets.*

1.3 Logiscope Repository

The Logiscope Repository is the directory where Logiscope will create and maintain all internal files storing the necessary information. The Logiscope Repository is specified using the **location** pane in the Project Creation window (see previous section).

At the end of the of a Logiscope project creation process, the following files are generated in the Logiscope Repository:

- <ProjectName>**.ttw** for Logiscope workspace,
- <ProjectName>**.ttp** for Logiscope project,
- <ProjectName>**.rst** for Logiscope Rule Set.

Once a Logiscope project has been "built": i.e. the source files of the project have been parsed to extract all necessary information for code verfication, a Logiscope folder is created containing several Logiscope internal ASCII format files files among which:

- a file named **standards.chk** containing all the violations found for the source code file of the project under analysis.
- a control graph file (suffixed by **.cgr**) for each source code file,
- global analysis result files (suffixed by **.dat**, **.tab** and **.graph**).

All files stored in the Logiscope Repository are internal data files to be used by Logiscope **Studio, Viewer** and **Batch**. They are not intended to be directly used by Logiscope users. The format of these files is clearly subject to changes.

1.4 Relaxation Mechanism

When **Relaxation mechanism** is activated for a Logiscope RuleChecker project, rule violations that have been checked and that you have decided are acceptable exceptions to the rule, can be relaxed for future builds: they will no longer appear in the list of rule violations. This can be very useful when checking violations in a context where multiple reviews are performed.

The violations that have been relaxed will remain accessible for future reference in the Relaxed Violations folder.

The relaxation mechanism is based on comments inserted into the code where the tolerated violations are. There are two ways to do this, depending on whether there is a single rule violation to relax on the line, or multiple ones to relax on the given line.

Relaxing a single rule violation

If there is a single violation to relax, it can be done as a comment on the same line as the code, using the following syntax:

```
some code // %RELAX<rule_mnemonic> justification
```

where:

- `rule_mnemonic`: is the mnemonic of the rule that you want to ignore violations of on the current line.
- `justification`: is free text, allowing to justify the relaxation of the rule violation.

If justification carries over several lines, they will not be included as part of the justification of the relaxation. In order for the justification to be written on several lines, the second syntax which is presented in the next section should be used.

Relaxing several violations and/or adding a longer justification

If there are several violations to relax for a same line (several violations occurring in different places in the code at the same time cannot be relaxed), or if the justification of the violation should have several lines, the following syntax should be used.

```
// >RELAX<rule_mnemonic> justification
```

followed by any number of empty lines, comment lines, or relaxations of other rules relating to the same code line, then by the code line of the violation.

Relaxing all violations in pieces of code

If all the violations of one or more rules are to be relaxed in a given piece of code (e.g. reused code included in a newly developed file), the piece of code should be surrounded by:

```
//  {{RELAX<list_of_rule_mnemonics> justification
the piece of code
//  }}RELAX<list of rule mnemonics>
```

where:

- `list_of_rule_mnemonics`: is the list of all mnemonics of the rules that you want to ignore violations of on the piece of code.
The rule mnemonics shall be separated by a comma.

Chapter 2

C++ Parsing Options

Logiscope uses source code parsers to extract all necessary information from the source code files specified in the project under analysis.

As the source code under analysis may contains compiler specifics, this chapter first details the available options to adapt the default behavior of the Logiscope C++ parsing to such specifics. They involve:

- choosing the appropriate dialect (cf.2.2),
- configuring the Logiscope C++ parser (cf. 2.3),
- managing pre-processing directives (cf. 2.4).

2.1 Reserved Keywords

The source code shall respect the C++ syntax defined in the reference document:

"*Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++*", by Andrew Koenig, referenced X3J16/96-0225 WG21/ N1043, dated: December, 1996.

Because of the use of a parser configuration file to define type specifiers, type qualifiers and access specifiers (cf. section 2.4), the list of keywords for *Logiscope C++* is smaller than the list of keywords of the language.

The list of C++ reserved words is the following:

```
asm
break
case
catch
class
const_cast
continue
default
delete
do
dynamic_cast
extern
else
enum
false
for
goto
```

```
if
namespace
new
operator
reinterpret_cast
return
sizeof
static_cast
struct
switch
template
this
throw
true
try
typeid
typename
typeof
union
using
while
```

The $ character is not authorized in identifiers.

2.2 Choosing the Appropriate Dialect

2.2.1 Available C++ Dialects

Compilers allow specifics that may not be correctly handled by the default Logiscope C++ parsing. To consider those specifics when parsing the source code and thus avoid parsing errors and warnings, the user shall choose the appropriate C++ dialect when setting up the project.

The list of available C++ dialects is the following:

- **Aix** for the IBM C++ 3.1 dialect
- **Alpha** for the DIGITAL C++ 6.0 dialect.
- **Borland_30** for the BORLAND C++ 3.0 dialect,
- **Borland_50** for the BORLAND C++ 5.0 dialect,
- **C++** for standard ISO C++,
- **Gnu_27** for the GNU 2.7 dialect,
- **Hp** for the HP C++ dialect,
- **Ilog_30** for the Ilog 3.0 dialect,
- **MFC** for the Microsoft Fundations Classes dialect,
- **Microsoft_15** for the MICROSOFT C++ 1.5 dialect,
- **Microsoft_20** for the MICROSOFT C++ 2.0 dialect,
- **Microsoft_50** for the MICROSOFT C++ 5.0 dialect,
- **Microsoft_60** for the MICROSOFT C++ 6.0 dialect,
- **Rhapsody** for source code file generated by *IBM Rational Rhapsody*,
- **Object_5x** for the ObjectStore 5 dialect,
- **Orbix_2x** for the Orbix 5 dialect,
- **sun** for the SUN C++ dialect,

The specifics of each dialect are specified in the following sections.

2.2.2 Reference Documentation

AIX

IBM C++ Compiler User’s Guide
5/2/96 xIC 3.1 1.69

Alpha

DIGITAL C++ Version 6.0 for DIGITAL
C++ Programming Language, Third Edition, by Bjarne Stroustrup

Borland C++ 3.0

Borland TURBO C++ 3.0
User's Guide

Borland C++ 5.0

Borland C++ 5.0 Development Suite
CD User’s Guide

GNU 2.7

Info file `**gcc**' made from the Texinfo source file **gcc.texinfo**.

HP C++

The C++ Programming Language
Bjarne Stroustrup
Second Edition
Addison-Wesley Publishing Company, 1991.

Microsoft C++ 1.5

Extract related to MICROSOFT C++ 1.5 language of the compact disk
Microsoft Visual C++
Development System and Tools for Windows

Microsoft C++ 2.0, 5.0, and 6.0

Microsoft Visual C++
Development System and Tools for Windows

SUN C++

SPARCompiler (Extract on the compact disk)
C++ 4.0 Language System

2.2.3 Dialect Specifics

Aix

The following keywords are recognized:

_offsetof
_System

Alpha

The following keywords are recognized:

__builtin_sizeof
__builtin_isfloat

Borland C++ 3.0

The following keywords are recognized:

_cdecl	**cdecl**
_far	**far**
_fastcall	**fastcall**
_huge	**huge**
_interrupt	**interrupt**
_loadds	
_near	**near**
_pascal	**pascal**
_saveregs	
_seg	

The following keywords are not recognized: **_asm**

Borland C++ 5.0

The following keywords are recognized:

_cdecl	**cdecl**	
_cs		
__declspec		
__ds	**_ds**	
__es	**_es**	
__except		
__export	**_export**	
__far	**_far**	**far**
__fastcall	**_fastcall**	
__huge	**_huge**	**huge**
__import	**_import**	
__interrupt	**_interrupt**	**interrupt**

__loadds	**_loadds**	
__near	**_near**	**near**
__pascal	**_pascal**	**pascal**
__rtti		
__saveregs	**_saveregs**	
__seg	**_seg**	
_ss		
__thread		

The following keywords are not recognized:

__asm	**_asm**
__finally	**__try**

Gnu 2.7

The following keywords are recognized:

bool		
true	**false**	
explicit		
mutable		
__alignof	**__alignof__**	
__asm	**__asm__**	
__attribute	**__attribute__**	
__const	**__const__**	
__extension__		
__inline	**__inline__**	
__label__		
__signed	**__signed__**	
__typeof	**__typeof__**	**typeof**
__volatile	**__volatile__**	
__wchar_t		

The following keywords are not recognized:

<?=>?=
<?>?
signature

Overmore, the Gnu specific construction:

```
#define AAAA(prefix, string, args...) fprintf(stder, prefix string, ##args)
```

is not supported.

Microsoft C++ 1.5

The following keywords are recognized:

__based	**_based**	
__cdecl	**_cdecl**	**cdecl**

__export	**_export**	
__far	**_far**	**far**
__fastcall	**_fastcall**	
__fortran	**_fortran**	
__huge	**_huge**	**huge**
__inline	**_inline**	
__interrupt	**_interrupt**	
__loadds	**_loadds**	
__near	**_near**	**near**
__pascal	**_pascal**	
__saveregs	**_saveregs**	
__segment	**_segment**	
__segname	**_segname**	

Microsoft C++ 2.0

The following keywords are recognized:

__based	**_based**	
__cdecl	**_cdecl**	**cdecl**
__declspec	**_declspec**	
__except		
__fastcall	**_fastcall**	
__inline	**_inline**	
__int8	**_int8**	
__int16	**_int16**	
__int32	**_int32**	
__int64	**_int64**	
__leave		
__stdcall	**_stdcall**	

The following keywords are not recognized:

__finally
__try

Microsoft C++ 5.0 and 6.0

The following keywords are recognized:

__based
__cdecl
__declspec
dllexport
dllimport
__except
__fastcall
__inline
__int16
__int32

__int64
__int8
__leave
__multiple_inheritance
naked
__single_inheritance
__stdcall
thread
__virtual_inheritance

The following keywords are not recognized:

__asm
__finally
__try

Rhapsody

A source code file is is considered as a generated file if it contains one of the following:

- a comment starting by #[,
- a comment starting by ##,
- a comment containing //! Generated Date .:

Otherwise, the source code file is considered as “hand written”.

In generated file, user-written code is detected using the following rules:

- a line following a line starting by a comment starting with ## is a user code line,
- a line ending with comment starting with ## is a user code line,
- a line starting with a comment starting with #[open a block of user code ending on a line starting with a comment starting with #].

Exception: if a comment ## is followed by « auto_generated » it does not introduce user code.

2.3 Configuring the Logiscope C++ Parser

In fact, each dialect is associated to a textual file that specifies the dialect specifics: the parser configuration file.

The default parser configuration files are located in the directory **data/env_c++/** in the Logiscope installation directory.

If the C++ dialect or the C++ library used is not supported by one of the standard C++ dialects, the user can customize an existing Logiscope C++ parser configuration file to better suit the application source code syntax specifics.

2.3.1 Type - Syntax Item Association

The parser configuration file allows the description of specifics types, keywords and macros in order to improve the source code parsing. For each identifier, a type may be associated. This type corresponds to an item of the C++ syntax.

The following table details the relation between the type of identifiers and the C++ syntax.

Type	**Syntax item**
IDENTIFIER	A simple identifier. Allows to mask predefined keywords.
STORAGE_CLASS	static, extern, register, ...
TYPE_SPECIFIER	int, char, float, unsigned, ...
TYPE_NAME	Allows to specify an identifier is a type name.
TYPE_QUALIFIER	const, volatile, ...
ACCESS_SPECIFIER	private, public, protected, ...
STRING_MACRO	A macro defined as a character string "..."
EXPRESSION_MACRO	A macro defined as an expression 3, t[i], f(a, b), ...
STATEMENT_MACRO	A macro defined as a statement a = 3; , f(a, b); , ...
DECLARATION_MACRO	A macro defined as a declaration int a; , myclass obj(a,b); , ...
TYPE_MACRO	A macro defined as a typename mytempl<int> , x##_ptr, ...
OPEN_BLOCK_MACRO	A macro that replaces {
CLOSE_BLOCK_MACRO	A macro that replaces }

OPEN_LOOP_MACRO	A macro that replaces for(;;) {
CLOSE_LOOP_MACRO	A macro that replaces } corresponding to for(;;) {
COMMENT_MACRO	A macro that replaces a comment
STRING_FUNC_MACRO	A macro function defined as a character string "..."
EXPRESSION_FUNC_MACRO	A macro function defined as an expression 3, t[i], f(a, b), ...
STATEMENT_FUNC_MACRO	A macro function defined as an statement a = 3; , f(a, b); , ...
DECLARATION_FUNC_MACRO	A macro function defined as an declaration int a; , myclass obj(a,b); , ...
TYPE_FUNC_MACRO	A macro function defined as a typename mytempl<int> , x##_ptr, ...
OPEN_BLOCK_FUNC_MACRO	A macro function that replaces {
CLOSE_BLOCK_FUNC_MACRO	A macro function that replaces }
OPEN_LOOP_FUNC_MACRO	A macro function that replaces for(;;) {
CLOSE_LOOP_FUNC_MACRO	A macro function that replaces } corresponding to for(;;) {
COMMENT_FUNC_MACRO	A macro function that replaces a comment
SQL_MACRO_START	Starts an SQL embedded statement. EXEC
SQL_MACRO_TYPE	Type of SQL embedded statement. SQL, ORACLE, IAF, ...

The definition of **..._FUNC_MACRO** types allows to pass parameters to these macros that should not be allowed for function calls.

Examples:

```
LIST_MAP(mylist, char *, str)
OPER_NAME(struct)
```

2.3.2 Syntax of the Parser Configuration File

The EBNF notation is used to describe the syntax of the C++ parser configuration file.

```
<conf> ::= "%START_C_CONFIGURATION" <head> <defs>
"%END_C_CONFIGURATION"
<head> ::= "=" <init> ";"
<init> ::= {"copy" <predefined>} | {"add" <predefined>}
<predefined> ::= "EMPTY"  | "C++"
<defs> ::=
         | <defs> <def>
<def> ::= <type> ":" <idents> ";"
<type> ::= "%IDENTIFIER"
          | "%STORAGE_CLASS"
          | "%TYPE_SPECIFIER"
          | "%TYPE_QUALIFIER"
          | "%TYPE_NAME"
          | "%ACCESS_SPECIFIER"
          | "%STRING_MACRO"
          | "%EXPRESSION_MACRO"
          | "%STATEMENT_MACRO"
          | "%DECLARATION_MACRO"
          | "%TYPE_MACRO"
          | "%OPEN_BLOCK_MACRO"
          | "%CLOSE_BLOCK_MACRO"
          | "%OPEN_LOOP_MACRO"
          | "%CLOSE_LOOP_MACRO"
          | "%COMMENT_MACRO"
          | "%STRING_FUNC_MACRO"
          | "%EXPRESSION_FUNC_MACRO"
          | "%STATEMENT_FUNC_MACRO"
          | "%DECLARATION_FUNC_MACRO"
          | "%TYPE_FUNC_MACRO"
          | "%OPEN_BLOCK_FUNC_MACRO"
          | "%CLOSE_BLOCK_FUNC_MACRO"
          | "%OPEN_LOOP_FUNC_MACRO"
          | "%COMMENT_FUNC_MACRO"
          | "%CLOSE_LOOP_FUNC_MACRO"
          | "%SQL_MACRO_START"
          | "%SQL_MACRO_TYPE"
<idents> ::=
           | <idents> <ident>
<ident> ::=[a-zA-Z0-9_][a-zA-Z0-9_]*
```

Comments begin with /* and end with */. They cannot be nested.

Separators are blanks, tabulations, ends of lines, and comments.

2.4 Managing pre-processing directives

C++ source code can be analyzed either expanded or not, this means after or before use of the preprocessor. The user shall supply Logiscope with the more suitable source code according to the analysis goals. The explanations below are intended to help choosing between these two solutions.

2.4.1 Impact on Analysis Results

As a general rule, if the purpose is to assess the maintainability of the software, the non-expanded source code suits better as it is near the developer point of view. For example, a piece of source code with a low complexity but using a lot of macro calls, can have a very high complexity after the preprocessing. Analyzing preprocessed code can generate unjustified alarms. In the same way, a piece of source code with a high complexity because of the use of a lot of `#if` statements, can be very simple after preprocessing. Analyzing preprocessed code can omit to raise important alarms.

More detailed considerations have to be taken into account. Within non-expanded code, conditional statements (`#if`, `#ifdef`, ...) are considered as `if` statements. Macro calls are considered as function calls.

Analyzing non preprocessed code has an influence on measurements when the same variable, the same type or the same function is declared in both branches of a `#if ... #else ... #endif`.

In the following example, the number of declared variables is equal to 2 instead of 1.

```
...
#ifdef POSIX
void  *ptr;
#else
char  *ptr;
#endif
...
```

In the same way, the following source code has a number of functions equal to 2.

```
...
#ifdef POSIX
int fn()
{ ... }
#else
int fn()
{ ... }
#endif
...
fn();
```

Choosing between both solutions shall be done according to the analysis goals and programming styles (macro often used or not, for example).

2.4.2 Restrictions

The Logiscope way of parsing source code imposes restrictions on the use of preprocessing statements in C++ programs. A file which does not follow the restrictions may be incompletely parsed by Logiscope (this yields *syntax error...* messages).

The main limitations are:

- Only the following macro types are allowed:
  - Macros used in place of an identifier or an expression.

Example:

```
#define ZERO 0
...
a=ZERO;
```

  - Macros used in place of a statement or a declaration.

Example:

```
#define PERROR(errno)  .....
....
if (ret_code < 0)
  PERROR (7);
```

  - Macros used in place of the beginning or the end of a block.

Example:

```
#define WHEN(x) if (x) {
#define END }
....
WHEN(ret_code < 0)
...
END
```

Among the above three types of macros, only part of the first one can be parsed without using the configuration file. The following example shows the use of invalid macros.

:

```
#define IS_NEG  < 0
#define STRUCT(x)  struct x

foo () {

STRUCT(point) pt1;/* should be defined as TYPE_FUNC_MACRO */

if (i IS_NEG)     /* invalid */
  i=0;

}
```

• The preprocessing directives can only be located in a place where an instruction, a declaration or an expression can be found. A preprocessing directive must not "cut" a declaration or an instruction.

Examples of invalid source code:

```
main () {
int
#ifdef OK      /* #ifdef inside a declaration */
i;
#else
j;
#endif

#ifdef OK
if (i == 1)    /* invalid */
#else
if (i == 0)
#endif
printf("OK\n") ;
else
printf("ERROR\n") ;
}
char *day_name (int n) {
static char *name[] = {
#ifdef FRENCH    /* invalid */
"jour inconnu",
"lundi", "mardi", "mercredi", "jeudi",
"vendredi", "samedi", "dimanche"
#else
"unknown day",
"Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"
#endif
} ;
return (n < 1 || n > 7) ? name[0] : name[n];
}
```

Chapter 3

Command Line Mode

3.1 Logiscope create

Logiscope projects: i.e. "**.ttp"** file are usually built using Logiscope **Studio** as described in chapter *Project Settings* or in the *Logiscope RuleChecker & QualityChecker Getting Started* documentation.

The logiscope **create** tool builds Logiscope projects from a standalone command line or within makefiles (replacing the compiler command) .

3.1.1 Command Line Mode

When started from a standard command line, The **create** tool creates a new project file with the information provided on the command line.

For a complete description of the command line options, please refer to the Command Line Options paragraph.

When used in this mode, there are two different ways for providing the files to be included into the project:

Automatic search

This is the default mode where the tool automatically searches the files in the directories. Key options having effect on this modes are:

`-root <root_dir>` : the root directory where the tool will start the search for source files. This option is not mandatory, and if omitted the default is to start the search in the current directory.

`-recurse` : if present indicates to the tool that the search for source files has to be recursive, meaning that the tool will also search the subdirectories of the root directory.

File list

In this mode, the tool will look for the –**`list`** option which has to be followed by a file name. This provided file contains a list of files to be included into the project. The file shall contain one filename per line.

Example: Assuming a file named `filelist.lst` containing the 3 following lines:

```
/users/logiscope/samples/C++/Hangman/GenericDlg.cpp
/users/logiscope/samples/C++/Hangman/Hangman32.cpp
/users/logiscope/samples/C++/Hangman/Hangman.cpp
```

Using the command line:

```
create –audit –lang c++ aProject.ttp –list filelist.lst
```

will create a new Logiscope C++ project file aProject.ttp containing 3 files: `GenericDlg.cpp`, Hangman32.cpp and `Hangman.cpp` on which *QualityChecker* and *RuleChecker* verification modules will be activated.

3.1.2 Makefile mode

When launched from makefiles, **create** is designed to intercept the command line usually passed to the compiler and uses the arguments to build the Logiscope project.

The project makefiles must be modified in order to launch **create** instead of the compiler. In this mode, the name of the project file (".ttp" file) has to be an absolute path, otherwise the process will stop.

When used inside a Makefile, **create** uses the same options as in command line mode, except for:

-root, -recurse, -list : which are not available in this mode

-- : which introduces the compiler command.

The following lines can be introduced in a Makefile to build a Logiscope project file :

```
CREATE=create /users/projects/my.ttp –audit -rule –lang c++
CC=$(CREATE) -- gcc
CPP=$(CC) –E
...
```

In this mode, the project file building process is as follows:

1. **create** is invoked for each file by the make utility, instead of the compiler.

2. When **create** is invoked for a file it adds the file to the project, with appropriate preprocessor options if any, then **create** starts the normal compilation command which will ensure that the normal build process will continue.

3. At the end of the make process, the Logiscope project is completed and can be used either using Logiscope **Studio** or with the **batch** tool (see next section).

Note: *Before executing the makefile, first clean the environment in order to force a full rebuild and to ensure that the* ***create*** *will catch all files.*

3.1.3 Options

The **create** options are the following:

`create -lang cpp`

`<ttp_file>`	name of a Logiscope project to be created (with the .ttp extension). Path has to be absolute if the option -- is used.
`[-root <directory>]`	where <directory> is the starting point of the source search. Default is the current directory. This option is exclusive with -list option.
`[-recurse]`	if present the source file search is done recursively in subfolders.
`[-list <list_file>]`	where <list_file> is the name of a file containing the list of filenames to add to the project (one file per line). This option is exclusive with -root option.
`[-repository <directory>]`	where <directory> is the name of the directory where Logiscope internal files will be stored.
`[-no_compilation]`	avoid compiling the files if the -- option is used
`[--]`	when used in a makefile, this option introduces the compilation command with its arguments.
`[-audit]`	to activate the *QualityChecker* verification module
`[-ref <Quality_model>]`	where `<Quality_model>` is the name of the Quality Model file (".ref") to add to the project. Default is <install_dir>/Ref/Logiscope.ref
`[-rule]`	to select the RuleChecker verification module
`[-rules <rules_file>]`	where <rule_file> is the name of the rule set file (.rst) to be included into the project. Default is the RuleChecker.rst file located in the /Ref/RuleSets/<lang>/ will be used.
`[-relax]`	to activate the violation relaxation mechanism for the project.

`[-import <folder_name>]`	where <folder_name> is the name of the project folder which will contain the external violation files to be imported. When this option is used the external violation importation mechanism is activated.
`[-external <file_name>]*`	where <file_name> is the name of a file to be added into the import project folder. This option can be repeated as many times as needed. Only applicable if the -import option is activated.
`[-dial <dialect_name>]`	where <dialect_name> is one of the available C++ dialects.
`[-source <suffixes>]`	where <suffixes> is the list of accepted suffixes for the source files. Default is "*.cpp;*.cc;*.cxx".
`[-header <suffixes>]`	where <suffixes> is the list of accepted suffixes for header files. Default is "*.h;*.hxx;*.hh"
`[-inline <suffixes>]`	where <suffixes> is the list of accepted suffixes for inline files . Default is "*.inl".
`[-D<macro_name>]*`	same syntax as a preprocessor. When used, this option activates the unifdef tool when parsing the code.
`[-U<macro_name>]*`	same syntax as a preprocessor. When used, this option activates the unifdef tool when parsing the code.

3.2 Logiscope batch

Logiscope **batch** is a tool designed to work with Logiscope in command line to:

- parse the source code files specified in a Logiscope project: i.e. ".ttp" file,
- generate reports in HTML and/or CSV format automatically.

Note that before using Logiscope **batch**, a Logiscope project shall have been created:

- using Logiscope **Studio**, refer refer to Section 1 or *IBM Rational Logiscope RuleChecker & QualityChecker Getting Started* documentation,
- or using Logiscope **create**, refer to the previous section.

Once the Logiscope project is created, Logiscope **batch** is ready to use.

3.2.1 Options

The **batch** command line options are the following:

`batch`

`<ttp_file>`	name of a Logiscope project.
`[-tcl <tcl_file>]`	name of a **Tcl** script to be used to generate the reports instead of the default **Tcl** scripts.
`[-o <output_directory>]`	directory where the all reports are generated.
`[-external <violation_file>]*`	name of the file to be added into the import project folder. This option can be repeated as many times as needed. This option is only significant for *RuleChecker* module for which the external violation importation mechanism is activated
`[-nobuild]`	generates reports without rebuilding the project. The project must have been built at least once previously.
`[-clean]`	before starting the build, the Logiscope build mechanism removes all intermediate files and empties the import project folder when the external violation importation mechanism is activated.
`[-addin export -format csv]`	generates the reports in csv format available using the **file/export** command.
`[-addin <addin> options]`	where `addin` nis the name of the addin to be activated and `options` the associated options generating the reports.

`[-table]`	generates tables in predefined HTML reports instead of slices or charts. By default, slices or charts are generated (depending on the project type). This option is available only on Windows as on Unix there are no slices or charts, only tables are generated.
`[-noframe]`	generates HTML reports with no left frame.
`[-v]`	displays the version of the **batch** tool.
`[-h]`	displays help and options for **batch**.
`[-err <log_err_folder>]`	directory where troubleshooting files **batch.err** and **batch.out** should be put. By default, messages are directed to standard output and error.

3.2.2 Examples of Use

Considering a previously created Logiscope project named **MyProject.ttp** where:

- *RuleChecker* and *QualityChecker* verification modules have been activated,
- the Logiscope Repository is located in the folder **MyProject/Logiscope,**

(Refer to the previous section or to the *RuleChecker & QualityChecker Getting Started* documentation to learn how creating a Logiscope project).

Executing the command on a command line or in a script:

```
batch MyProject.ttp
```

will:

- perform the parsing of all source files specified in the Logiscope project **MyProject.ttp,**
- run the standard TCL script **QualityReport.tcl** located in <l*og_install_dir*>**/Scripts** to generate the standard *QualityChecker* HTML report named **MyProjectquality.html** in the default **MyProject/Logiscope/reports.dir** folder.
- run the standard TCL script **RuleReport.tcl** located in <*log_install_dir*>**/Scripts** to generate the standard *RuleChecker* HTML report named **MyProjectrule.html** in the default **MyProject/Logiscope/reports.dir** folder.

Chapter 4

Standard Metrics

Logiscope QualityChecker C++ proposes a set of standard source code metrics. Source code metrics are static measurements (i.e. obtained without executing the program) to be used to assess software attributes (e.g. complexity, self-descriptiveness) or characteristics (e.g. Maintainability, Reliability) of the C++ functions, classes, modules, application under evaluation.

The metrics can be combined to define new metrics more closely adapted to the quality evaluation of the source code. For example, the "comments frequency" metric, well suited to evaluate quality criteria such as self-descriptiviness or analyzability, can be defined by combining two basic metrics: "number of comments" and "number of statements".

The user can associate threshold values with each of the quality model metrics, indicating minimum and maximum reference values accepted for the metric.

For more details on Source Code Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

Source code metrics apply to different domains (e.g. line counting, control flow, data flow, calling relationship) and the range of their scope varies.

The scope of a metric designates the element of the source code the metric will apply to. The following scopes are available for *Logiscope QualityChecker C++*.

- The *Function scope*: the metrics are available for each member and non-member function defined in the source files specified in the Logiscope Project under analysis.
- The *Class scope*: the metrics are available for each C++ class defined in the header and source files specified in the Logiscope Project under analysis. Classes contain member functions and member data.
- The *Module scope*: the metrics are available for each C++ header or source file specified in the Logiscope Project under analysis.
- The *Application scope*: the metrics are available for the set of C++ header and source files specified in the Logiscope Project .

4.1 Function Scope

4.1.1 Line Counting

For more details on Line Counting Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

lc_cline Total number of lines

Definition Total number of lines in the function.

lc_cloc Number of lines of code

Definition Total number of lines containing executable code in the function.

lc_cblank Number of empty lines

Definition Number of lines containing only non printable characters in the function.

lc_ccomm Number of lines of comments

Definition Number of lines of comment in the function.

lc_ccpp Number of preprocessor statements

Definition Number of preprocessor directives (e.g. `#include`, `#define`, `#ifdef`) in the function.

lc_csbra Number of lines with lone braces

Definition Number of lines containing only a single brace character : i.e. “{“ or “}” in the function.

lc_pro_c Number of lines in Pro*C

Definition Number of lines written in Pro*C in the function.

lc_bcom Number of comment blocks.

Definition Number of comment blocks used between a function's header and the closing curly bracket (Blocks of COMments). Several consecutive comments are counted as a single comment block.

Example

```
funct() ;
{
  /* this is a comment */
  printf ("----------------------") ;
  /* this is a second */
  /* comment          */
  printf ("----------------------") ;
  /* this is a third
     comment          */
}
```

lc_bcom= 3

Alias BCOM

lc_bcob Number of comment blocks before

Definition 1 if there is a block of comments used just before a function (Blocks of COmments Before). 0 either.

Example

```
/* this comment is not counted      */
/* as a comment before the function */
int i;
/* this one is counted
   as a comment                      */
/* before the function               */
funct() ;
{
  printf ("----------------------") ;
  printf ("----------------------") ;
}
```

lc_bcob = 1

Alias BCOB

lc_parse Number of lines not parsed

Definition Number of lines which cannot be parsed in a function because of syntax errors or of some particular uses of macros.

4.1.2 Lexical and syntactic items

lc_algo Number of syntactic entities in algorithms

Definition Number of syntactic entities inside statements of a function that are not counted as declarations.

lc_decl Number of syntactic entities in declarations

Definition Number of syntactic entities in the declaration part of a function.

lc_stat Number of Statements

Also called STMT.

Definition Number of statements in the function body

Without an optional parameter, following statements are counted:

- Control statements: `break`, statement block, `continue`, `do`, `for`, `goto`, `if`, labels, `return`, `switch`, `while`, `case`, `default`,
- Statements followed by `;` ,
- Empty statement.

This metric can be parametrized to count the statements in a familiar way:

- if no parameter is provided, all statements listed above are counted,
- if the parameter **"no_null_stat"** is provided, block statements, empty statements and labeled statements (including `case` and `default` labels in `switch` statements) are omitted.

lc_synt Number of syntactic entities

Definition Number of syntactic entities used in the function.

Note **lc_synt** is the sum of **lc_decl** and **lc_algo**.

4.1.3 Data Flow

dc_consts Numbers of declared constants

Definition Number of constants in a function declared by:

- the `#define` statement,

- variables having a simple type declared as `const`,
- `enum` elements.

dc_types Number of declared types

Definition Number of types declared in a function with the `typedef`, `struct`, `class` or `enum` statement.

dc_vars Number of declared variables

Definition Number of variables declared in a function.

dc_lvars Number of local variables

Also called LVAR.

Definition Total number of variables declared in a function (Local VARiables).

dc_clas_var Number of class-type local variables

Also called LVARop.

Definition Number of class type variables which are local to a function. This metric shows a specific type of coupling between classes.

dc_other_clas_var Number of other class-type local variables

Definition Number of class type variables which are local to a function, where the class is different from the current class.
If the function being analyzed is a non-member function, the value is 0.

ic_param Number of parameters

Also called PARA.

Definition Number of parameters of a function.

ic_parvar Variable number of parameters

Definition Equals 1 if the function has a variable number of parameters, 0 otherwise.

ic_paradd Number of parameters passed by reference

Also called PARAadd.

Definition Number of parameters passed by reference of a function. If the function returns a value, then the returned value is considered as a passed by reference parameter.

ic_parcl Number of class-type parameters

Also called PARAc.

Definition Number of class-type parameters of a function. If the function returns a class-type value then the returned value is considered as a class-type parameter. This metric shows a specific type of coupling between classes.

ic_par_othercl Number of other class-type parameters

Definition Number of class-type parameters of a function, where the class is different from the current class. If the function being analyzed is a non-member function, then the value is 0.

ic_parval Number of parameters passed by value

Also called PARAval.

Definition Number of parameters passed by value of a function.

ic_usedp Number of parameters used

Also called U_PARA.

Definition Number of function parameters used in a function body. A parameter is said to be used wherever it appears in the function code. Combined with the number of function parameters, this metric is a good indicator of the consistency of the function's interface.

ic_vare Number of uses of external attributes

Also called VARe

Definition Number of uses of attributes defined outside the class. An attribute is said to be "external" if it belongs to another class.

All attribute occurrences are counted.

ic_vari Number of uses of internal attributes

Also called VARi.

Definition Number of uses of attributes defined in the class. An attribute is said to be "local" if it belongs to the class of the function being analyzed.

All attribute occurrences are counted.

ic_varpe Number of distinct uses of external Aattributes

Also called VAR_PATHSe.

Definition Number of distinct times attributes defined outside the class are used. An attribute is said to be "external" if it belongs to another class.

Different uses of the same attribute count for one.

ic_varpi Number of distinct uses of local attributes

Also called VAR_PATHSi.

Definition Number of times the distinct class attributes are used. An attribute is said to be "local" if it belongs to the class of the function being analyzed.

Different uses of the same attribute count for one.

4.1.4 Halstead Metrics

For more details on Halstead Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

n1 Number of distinct operators

Also called ha_dopt.

Definition Number of different operators used in a function.

This metric can be parametrized to count the operators in a familiar way:

- if no parameter is provided, operators are counted between the beginning of the function’s definition and its closing curly bracket,
- if the parameter **"in_body"** is provided, operators are only counted in the function body (that is between the function’s opening and closing curly brackets).

For the use of this parameter, see Chapter Customizing Metrics & Rules.

The following are operators:

- Expressions:
  - Unary operators:

+ -	unary plus or minus
++ --	pre-/post- increment or decrement
!	negation
~	complement of 1 or destructor
*****	indirection
&	address
sizeof	sizeof
throw	throw
new	new
::new	global scope new
delete	delete
::delete	global scope delete
delete []	array delete
::delete []	global scope array delete
.	dot
->	arrow
()	expression in parenthesis

- Binary Operators:

+ - * / %	arithmetic operators
<< >> & | ^	bitwise operators
> < <= >= == !=	comparison operators
&& ||	logical operators
->* .*	pointer to member operators

- Ternary conditional operator: **?:**
- Assignment operators: **= *= /= %= += -= >>= <<= &= ^= |=**
- Other operators:

(...)	cast	(ex: **(**float**)**1)
dynamic_cast	cast	(ex: **dynamic_cast**<T>(v))
static_cast	cast	(ex: **static_cast**<T>(v))
reinterpret_cast	cast	(ex: **reinterpret_cast**<T>(v))
const_cast	cast	(ex: **const_cast**<T>(v))
[]	subscripting	(ex: a**[**i**]**)

::	(global) scope	(ex: X**::**i, **::**i)
...()	function call	(ex: func**(**1**)**)
(.., .., ..)	expressions list	(ex: func**(**1**,**2**,**3**)**)
this		

- Statements:

IF	**ELSE**	**WHILE( )**	**DO WHILE( )**
RETURN	**FOR(;;)**	**SWITCH CASE**	**BREAK**
CONTINUE	**GOTO label**		**DEFAULT**
LABEL			
{ }	(compound)		
;	(empty statement)		

- Declarations:

ASM	(ex: **asm**("foo"))
EXTERN	(ex: **extern** "C" { ... })
; (empty declaration)	
(member) declaration	(ex: **int i; int i = 1;**)
type specifier	(ex: **int**)
storage class	(ex: **auto**, **register**, **static**, **extern**, **mutable**)
enumerator specifier	(ex: **enum** X **{** ... **}**;)
enumerator-list	(ex: **enum** X {a, b, c};)
enumerator-definition	(ex: **enum** X **{**a=1**,** b=2**}**;)
typename	(ex: typedef **typename** X::a b;)
namespace definition	(ex: **namespace** N { ... })
using declaration	(ex: **using** A::x;)
using directive	(ex: **using namespace** M;)

- Declarators:

	function declarator	(ex: **int func();**)
[ ]	array declarator	(ex: int tab**[**5**];)**
*****	pointer declarator	(ex: int ***i;**)
&	reference declarator	(ex: int**&** i;)
::*	pointer to member declarator	(ex: int X**::*** i;)
(.., .., ..)	parameter-declaration-list	(ex: int func**(**int i**,** char *j**);**)

{.., .., ..}	initializer-list	(ex: int tab[] = **{**1**,** 3**,** 5**}**;)
	type qualifier	(ex: **const**, **volatile**)
	type identifier	(ex: sizeof(**int**), new (**int**))

- Classes:

class keys	class struct union
access specifiers	private public protected

- Derived classes:

base classes	(ex : class Z **:** public X **,** public Y)

- Special member functions

:	constructor initializer	(ex: C::C(): A() { ...} try **:** i(f(ii)), d(id))
.., .., ..	member initializer list	(ex: i(f(ii))**,** d(id))
	member initializer id	(ex: i(f(ii)))

- Overloading: **operator ...**
- Templates:

template parameters	(ex: template<class K, class V>)
type parameter	(ex: template<class K = int> template<template<class T> class K = int>)
template name	(ex: T1<T2>)
template argument list	(ex: T<T1,T2,T3>)
explicit instantiation	(ex: template A::operator void*();)
explicit specialization	(ex: template <> A::operator char*() { return 0; })

- Exceptions:

throw (.., ..)	exception specification (ex: int func() throw(X,Y)
try { ... }	try block
catch (..) { ... }	handler

- Preprocessing directives:

#define	**#undef**	
#if	**#ifdef**	**#ifndef**
#elif	**#else**	**#endif**

#line	**#error**	**#pragma**
#	**#include**	
#define func(.., .., ..)	macro arguments	

N1 Total number of operators

Also called ha_topt.

Definition Total number of operators used in a function.

Note The function area where operators are counted depends on the parameter of the **n1** metric (see above).

n2 Number of distinct operands

Also called ha_dopd.

Definition Number of different operands used in a function.

This metric can be parameterized to count the operands in a familiar way:

- if no parameter is provided, operands are counted between the beginning of the function's definition and its closing curly bracket,
- if the parameter **"in_body"** is provided, operands are only counted in the function's body (that is between the function's opening and closing curly brackets).

For the use of this parameter, see Chapter Customizing Metrics & Rules.

The following are operands:

- Literals:
  - Decimal literals (ex: 45, 45u, 45U, 45l, 45L, 45uL)
  - Octal literals (ex: 0177, 0177u, 0177l)
  - Hexadecimal literals (ex: 0x5f, 0X5f, Ox5fu, 0x5fl)
  - Floating literals (ex: 1.2e-3, 1e+4f, 3.4l)
  - Character literals (ex: 'c', L'c', 'cd', '\a', '\177', '\x5f')
  - String literals (ex: "hello", L" world\n")
  - Boolean literals (true or false)
- Identifiers (variable names, type names, function names, etc.)
- File names in #include clauses (ex: #include <stdlib.h>, #include "foo.h")
- Operator names:

new	**delete**	**new[]**	**delete[]**					
+	**-**	*****	**/**	**%**	**^**	**&**	**|**	**~**
!	**=**	**<**	**>**	**+=**	**-=**	***=**	**/=**	**%=**

^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	,	->*	->
()	[]							

N2 Total number of operands

Also called ha_topd.

Definition Total number of operands used in a function.

Note The function area where operands are counted depends on the parameter of the **n2** metric (see above).

n Halstead vocabulary

Definition Halstead vocabulary of the function: n = n1 + n2

Alias ha_voc

N Halstead length

Definition Halstead length of the function: N = N1 + N2

Alias ha_olg

CN Halstead estimated length

Definition Halstead estimated length of the function:
$CN = n1 * \log_2(n1) + n2 * \log_2(n2)$

Alias ha_elg

V Halstead volume

Definition Halstead volume of the function: $V = N * \log_2(n)$

Alias ha_vol

L Halstead level

Definition Halstead level of the function: L = (2 * n2) / (n1 * N2)

Alias ha_lev

D Halstead difficulty

Definition Halstead difficulty of the function: D = 1/L

Alias ha_dif

I Halstead intelligent content

Definition Halstead intelligent content of the function: I = L * V

Alias ha_int

E Halstead mental effort

Definition Halstead mental effort of the function: E = V / L

Alias ha_eff

4.1.5 Structured Programming

In structured programming:

- a function shall have a single entry point and a single exit point,
- each iterative of selective structures shall have a single exit point: i.e. no `goto`, `break`, `continue` or `return` statement in the structure.

Structured programming improves source code maintainability.

ct_bran Number of destructuring statements

Definition Number of destructuring statements in a function (`break` and `continue` in loops, and `goto` statements).

ct_break Number of break and continue branchings

Definition Number of `break` or `continue` statements used to exit from loop structures in the function.

`break` statements in `switch` structures are not counted.

ct_exit Number of out statements

Definition Number of nodes associated with an explicit exit from a function (*`return`*, *`exit`*).

Alias N_OUT

ct_goto Number of gotos

Definition Number of *goto* statements.
Alias GOTO

ESS_CPX Essentiel complexity

Definition Cyclomatic number of the “reduced” control graph of the function.
The “reduced” control graph is obtained by removing all structured constructs from the control graph of the function.
A structured contruct is a selective or iterative structure that does not contains auxiliary exit statements: `goto, break, continue` or `return`.

Justification When the Essentiel Complexity is equal to 1, the function complies with the structured programming rules.
Note that the **ct_exit** and **ct_bran** metrics already provide such an information on the structuring of the function with more details.

4.1.6 Control Flow

For more details on Control Graph Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

ct_decis Number of decisions

Definition Number of selective statements in a function : `if, switch`
Alias N_STRUCT

ct_degree Maximum degree

Definition Maximum number of edges departing from a node.

ct_edge Number of edges

Definition Number of edges of the control graph of a function.

ct_nest Maximum nesting level

Definition Maximum nesting level of control structures in a function.

ct_node Number of nodes

Definition Number of nodes of the control graph of a function.

ct_loop Number of loops

Definition Number of iterative statements in a function (pre- and post- tested loops): `for`, `while`, `do while`,

ct_path Number of non-cyclic paths

Definition Number of non-cyclic execution paths of the control graph of the function.

Note Since version 6.6.1, ct_npath replaces the previous ct_path metric, now considered as deprecated due to inaccurate results in some contexts but kept only for non regression purpose.

ct_raise Number of exceptions raised

Definition Number of occurrences of the *`throw`* clause within a function body.

Alias N_RAISE

ct_try Number of exceptions handlers

Definition Number of *`try`* blocks in a function.

Alias N_EXCEPT

ct_vg Cyclomatic number (VG)

Definition Cyclomatic number of the control graph of the function.

Alias VG, ct_cyclo

DES_CPX Design complexity

Definition Cyclomatic number of the “design” control graph of the function.
The “design” control graph is obtained by removing all constructs that do not contain calls from the control graph of the function.

4.1.7 Relative Call Graph

For more details on Call Graph Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

cg_entropy Relative call graph entropy

Definition SCHUTT entropy of the relative call graph of the function.

Alias ENTROPY

cg_ hiercpx Relative call graph hierarchical complexity

Definition Average number of components per level(i.e. number of components divided by number of levels) of the relative call graph of the function..

Alias HIER_CPX

cg_levels Relative call graph levels

Definition Depth of the relative call graph of the function..

Alias LEVELS

cg_strucpx Relative call graph structural complexity

Definition Average number of calls per component: i.e. number of calling relations between components divided by the number of components) of the relative call graph of the function..

Alias STRU_CPX

cg_testab Relative call graph testability

Definition Mohanty system testability of the relative call graph of the function.

Alias TESTBTY

dc_calls Number of direct calls

Definition Number of direct calls in a function.
Different calls to the same function count for one call.

Alias DRCT_CALLS

dc_calle Number of external calls

Definition Number of calls to functions defined outside the class.
All call occurrences are counted.
A function is said to be "defined outside" the class if the function does not belong to the same class as the function being analyzed. If the function being analyzed is a non-member function, then all functions called by the function being analyzed are considered as "defined outside" the class.

Alias CALLe

dc_calli Number of internal calls

Definition Number of calls to functions defined within the class.
All call occurrences are counted.
A function is said to be "defined in" the class if the function belongs to the same class as the function being analyzed. If the function being analyzed is a non-member function, then there is no function "defined in" the class (the value is 0).

Alias CALLi

dc_calling Number of callers

Definition Number of functions calling the designated function.

Alias NBCALLING

dc_callpe Number of external direct calls

Definition Number of distinct calls to functions defined outside the class of the function being analyzed (see **dc_calle** above).
Different calls to the same function count for one call.

Alias CALL_PATHSe

dc_callpi Number of internal direct calls

Definition Number of distinct calls to functions defined in the class of the function being analyzed (see **dc_calli** above).
Different calls to the same function count for one call.

Alias CALL_PATHSi

dc_stat_call Number of calls to static members

Definition Number of calls to static member functions in a function.

IND_CALLS Relative call graph call-paths

Definition Number of call paths in the relative call graph of the function.

4.2 Class Scope

4.2.1 Comments

cl_bcob Number of comment blocks before

Also called BCOBc.

Definition Number of blocks of comments located between a class header and the curly bracket of the previous class or between a class header and the beginning of the file.

cl_bcom Number of comment blocks

Also called BCOMc.

Definition Number of comment blocks in a class. Consecutive comments are counted as belonging to the same block. Comments located outside the class are not counted.

4.2.2 Data Flow

cl_base_priv Number of private base classes

Definition Number of declared classes from which a class inherits, whose names appear after the `private` keyword.

cl_base_prot Number of protected base classes

Definition Number of declared classes from which a class inherits, whose names appear after the `protected` keyword.

cl_base_publ Number of public base classes

Definition Number of declared classes from which a class inherits, whose names appear after the `public` keyword.

cl_base_virt Number of virtual base classes

Definition Number of declared classes from which a class inherits, whose names appear after the `virtual` keyword.

cl_clas_frnd Number of friend classes

Definition Number of classes declared in a class definition, whose names appear after the `friend` keyword.

cl_cobc Coupling between classes

Also called COBC, cl_dep_deg

Definition Coupling between classes is the sum of:

- the number of inherited classes (see in **in_data_class** Number of Direct Base Classes),
- the number of class type attributes for the class (see **cl_data_class** below),
- two times the number of calls to static member functions for class methods (see in **dc_stat_call** Number of Calls to Static Member Functions).
- two times the number of class-type parameters for the class methods,
- three times the number of class-type local variables for the class methods (see in **dc_clas_var** Number of Class Type Local Variables).

$$cl\_cobc = in\_dbases + cl\_data\_class + \sum_{methods} (2 \times dc\_stat\_call + 2 \times ic\_parcl + 3 \times dc\_clas\_var)$$

cl_data_class Sum of class-type attributes

Definition Number of class-type attributes for the class.

Alias LACT

cl_data_priv Number of private attributes

Definition Number of data members declared in the `private` section of a class.

Alias LAPI, cl_field_priv

cl_data_prot Number of protected attributes

Definition Number of data members declared in the `protected` section of a class.

Alias LAPO, cl_field_prot

cl_data_publ Number of public attributes

Definition Number of data members declared in the `public` section of a class.

Alias LAPU, cl_field_publ

cl_data_stat Number of static data members

Definition Number of data members declared after the `static` keyword in a class.

cl_data_inh Number of inherited attributes

Definition Number of public or protected attributes in the base classes of a class, which are not overridden in that class.

cl_dep_meth Number of dependent methods

Definition Number of methods within the class depending on other classes. A method is said to be dependent if:

- it calls a non-member function or other class methods (see in **dc_calle** Number of Calls to functions Defined outside the Class),
- it uses an attribute which belongs to a different class (see in **ic_vare** Number of Times External Attributes are used),
- it has a class instance parameter which belongs to a different class (see in **ic_par_othercl** Number of Other Class Type Parameters),
- it declares a class instance variable which belongs to a different class (see in **dc_other_clas_var** Number of other Class Type Local Variables).

$$\text{cl\_dep\_meth} = \sum_{\text{methods}} \begin{cases} 1 & \text{if } dc\_calle + ic\_vare + ic\_par\_othercl + dc\_other\_clas\_var > 0 \\ 0 & \text{otherwise} \end{cases}$$

Alias NMD

cl_rfc Response for a class

Definition Number of methods that can be invoked in response to a message to an object of the class or by some method in the class.
This includes all methods accessible within the class hierarchy.

cl_type Number of local types

Definition Number of types declared in a class.

cl_const Number of local constants

Definition Number of constants declared in a class. Constants are data members declared with the keyword `const`, like `const type name ...`, or `type * const name ...` (constant pointer), or `type C::* const name` (constant pointer to member) for instance (but not pointers to constant).

cl_genp Number of of parameters for templates

Definition Number of parameters declared in a class for classes that are templates. If cl_genp has the value 0 the class is not a template.

cl_oper_conv Number of conversion operators

Definition Number of conversion operators declared in a class declaration.

cl_oper_std Number of standard operators

Definition Number of operators declared in a class, whose names belong to a certain list being a parameter of the metric (by default, this list is empty).

cl_oper_affc Number of assignment operators

Definition Number of operators declared in a class, whose names belong to a certain list which is a parameter of the metric (by default, this list contains `"="`, `"+="`, `"-="`, `"*="`, `"/="`, `"%="`, `"^="`, `"&="`, `"|="`, `"<<="`, `">>="`, `"+"`, `"-"`, `"*"`, `"/"` and `"[]"`).

cl_oper_spec Number of special operators

Definition Number of operators declared in a class, whose names belong to a certain list which is a parameter of the metric (by default, this list contains `"->"`, `"()"`, `","`, `"->*"`, `"new"`, `"delete"`, `"new[]"`, and `"delete[]"`).

4.2.3 Statistical Aggregates of Function Metrics

cl_func_priv Number of private methods

Definition Number of methods declared in the `private` section of a class.
Alias LMPL, cl_meth_priv

cl_func_prot Number of protected methods

Definition Number of methods declared in the `protected` section of a class.
Alias LMPO, cl_meth_prot

cl_func_publ Number of public methods

Definition Number of methods declared in the `public` section of a class.

Alias LMPU, cl_meth_publ

cl_func_virt Number of virtual methods

Definition Number of methods declared after the `virtual` keyword in a class.

cl_func_pure Number of abstract methods

Definition Number of methods declared after the `virtual` keyword and followed by `=0` in a class.

Alias LMABS

cl_func_cons Number of constant methods

Definition Number of methods declared after the `const` keyword in a class.

cl_func_inln Number of inline methods

Definition Number of methods declared after the `inline` keyword in a class.

cl_func_excp Number of methods handling or raising exceptions

Definition Number of methods declared in a class declaration in which:

- the body of the function is a `try` block, or
- the function body contains a `try` block, or
- exceptions are specified using the `throw` keyword.

cl_func_frnd Number of friend functions

Definition Number of methods declared after the `friend` keyword in a class.

cl_func_inh Number of inherited methods

Definition Number of public or protected methods in the base classes of a class, which are not overridden in that class.

cl_func_over Number of overridden methods

Definition Number of inherited methods which a class overrides.

Justification High values for **cl_func_over** tend to indicate design problems. Sub-classes should generally add to and extend the functionality of the parent classes rather than overriding them.

Alias LMRE

cl_data_vare Sum of uses of external attributes

Definition Total number of times attributes defined in other classes (ic_varpe) are used by the class methods.

Alias LMVAR_PATHSe

cl_data_vari Sum of uses of internal attributes

Definition Total number of times the class's attributes (ic_varpi) are used by the class methods.

Alias LMVAR_PATHSi

cl_fpriv_path Sum of paths of private methods

Definition Sum of non-cyclic execution paths (cl_path) of the private methods of the class.

Alias LMPIPATH

cl_fprot_path Sum of paths of protected methodss

Definition Sum of non-cyclic execution paths (cl_path) of the protected methods of the class.

Alias LMPOPATH

cl_fpubl_path Sum of paths of public methods

Definition Sum of non-cyclic execution paths (cl_path) of the public methods of the class.

Alias LMPUPATH

cl_func_calle Sum of external calls

Definition Total number of calls from the class methods to non-member functions or member functions of other classes (dc_callpi).

Alias LMCALL_PATHSe

cl_func_calli Sum of internal calls

Definition Total number of calls from class methods to member functions of the same class (dc_callpi).

Alias LMCALL_PATHSi

cl_usedp Sum of parameters

Definition Total number of parameters (ic_usedp) used in the class methods.

Alias LMU_PARA

The two following metrics have been introduced by Shyam R. Chidamber and Chris F. Kemerer in "*A Metrics Suite for Object Oriented Design*" (IEEE Transactions on Software Engineering, vol 20, pp. 476-493, June 1994).

cl_wmc Weighted Methods per Class

Definition Sum of static complexities of class methods.
Static complexity is represented in this calculation by the cyclomatic numbers (VG).

Alias LMVG, cl_cyclo

cl_locm Lack of cohesion of methods

Definition Percentage of methods that do not access a specific attribute of a class averaged over all attributes in that class.

$$cl\_locm = \frac{\sum_{i=1}^{TA} (1 - Ac(A_i))}{TA}$$

where:

$$Ac(A_i) = \frac{\sum_{j=1}^{TM} is\_accessed(A_i, M_j)}{TM}$$

and:

$$is\_accessed(A_i, M_j) = \begin{cases} 1 & \text{¤ } M_j \text{ accesses } A_i \\ 0 & \text{otherwise} \end{cases}$$

4.2.4 Inheritance Tree

in_bases Number of base classes

Definition Number of classes from which a class inherits directly or not
If multiple inheritance is not used, the value of **in_bases** is equal to the value of **in_depth**.

Alias in_inherits

in_dbases Number of direct base classes

Definition Number of classes from which a class directly inherits.
Note A value of **in_dbases** upper than 1 denotes multiple inheritance.
Alias MII, in_dinherits

in_depth Depth of the inheritance tree

Definition Maximum length of an inheritance chain starting from a class.

in_derived Number of derived classes

Definition Total number of classes which inherit from a class directly or indirectly.

in_noc Number of children

Definition Number of classes which inherit directly from a class.
Alias NOC, in_dderived

in_reinh Number of classes inherited several times

Definition Number of classes which directly inherit from a class.

4.2.5 Use Graph

cu_level Depth of use

Definition Maximum length of a chain of use starting from a class (not counting use loop).

cu_cdused Number of direct used classes

Definition Number of classes used directly by a class.

cu_cused Number of used classes

Definition Number of classes used by the current class directly or not.

cu_cdusers Number of direct user classes

Definition Number of classes which use directly a class.

cu_cusers Number of user classes

Definition Total number of classes which use directly or not a class.

4.3 Module Scope

4.3.1 Line Counting

For more details on Line Counting Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

md_blank Number of empty lines

Definition Number of lines containing only non printable characters in the module.

md_comm Number of lines of comments

Definition Number of lines of comments in the module.
Alias LCOM

md_cpp Number of preprocessor statements

Definition Number of preprocessor directives (e.g. `#include`, `#define`, `#ifdef`) in the module.

md_line Total number of lines

Definition Total number of lines in the module.

md_loc Number of lines of code

Definition Total number of lines containing executable code in the module.

md_sbra Number of lines with lone braces

Definition Number of lines containing only a single brace character : i.e. “{“ or “}” in the module.

md_pro_c Number of lines in Pro*C

Definition Total number of lines of PRO*C in the module.

4.3.2 Lexical and syntactic items

md_algo Number of syntactic entities in algorithms

Definition Number of syntactic entities inside statements that are not counted as declaration in the file.

md_decl Number of syntactic entities in declarations

Definition Number of syntactic entities in the declaration part of the module (function headers and declaration.

md_synt Number of syntactic entities

Definition Total number of syntactic entities in the module.

md_stat Number of statements

Definition Total number of executable statements in the functions defined in the module.

4.3.3 Data Flow

md_consts Number of declared constants

Definition Number of constants declared in the module.

md_expfn Number of exported functions

Definition Number of non-static global functions defined in the module.

md_expva Number of exported variables

Definition Number of non-static global variables defined in the module.

md_impmo Number of imported modules

Definition Number of modules included inside a module.

md_types Number of declared types

Definition Number of types declared in the module.

md_vars Number of declared variables

Definition Number of variables declared in the module.

4.3.4 Halstead Metrics

For more details on Halstead Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

md_n1 Number of distinct operators

Definition Number of distinct operators referenced in the module.
See metric n1 in Function Scope section for the specification of operators.

md_n2 Number of distinct operands

Definition Number of distinct operands referenced in the module.
See metric n2 in Function Scope section for the specification of operands.

md_N1 Total number of operators

Definition Total number of operators referenced in the module.

md_N2 Total number of operands

Definition Total number of operands referenced in the module.

md_n Halstead vocabulary

Definition Halstead vocabulary of the module.
$n = n1 + n2$

md_N Halstead length

Definition Halstead observed length of the module.
$N = N1 + N2$

md_CN Halstead estimated length

Definition Halstead estimated length of the module.
$\hat{N} = n1 * \log_2(n1) + n2 * \log_2(n2)$.

md_V Halstead volume

Definition Halstead Program Volume
$V = N * \log_2(n)$

md_L Halstead level

Definition Halstead Program Level
L = (2 * n2) / (n1 * N2)

md_D Halstead difficulty

Definition Halstead Program Difficulty
D = 1/L

md_I Halstead intelligent content

Definition Halstead Intelligent Content
I = L * V

md_E Halstead mental effort

Definition Halstead Intelligent Content
E = V / L

4.4 Application Scope

Metrics presented in this section are based on the set of C++ header and source files specified in Logiscope Project under analysis. It is therefore recommended to use these metrics values exclusively for a complete application or for a coherent subsystem.

4.4.1 Line Counting

For more details on Line Counting Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

ap_sline Total number of lines

Definition Total number of lines in the application source files.

ap_sloc Number of lines of code

Definition Total number of lines containing executable in the application source files.

ap_sblank Number of empty lines

Definition Total number of lines containing only non printable characters in the application source files.

ap_scomm Number of lines of comments

Definition Totam number of lines of comments in the application source files.

ap_scpp Number of preprocessor statements

Definition Number of preprocessor directives (e.g. `#include`, `#define`, `#ifdef`). in the application source files.

ap_ssbra Number of “brace” lines

Definition Number of lines containing only a single brace character (“{“ or “}”) in the application source files.

4.4.2 Application Aggregates

ap_clas Number of application classes

Definition Number of classes defined in the header and source files.

Alias LCA

ap_func Number of application functions

Definition Number of member and non-member functionsdefined in the header and source files.

Alias LMA

ap_stat Number of statements

Definition Number of executables statements (i.e. lc_stat) of all the functions defined in the application functions.

ap_cbo Coupling between objects

Definition Sum of the relationships from class to class other than inheritance relationships.

$$ap\_cbo = \sum_{classes} (cl\_func\_calls + cl\_data\_class)$$

Alias CBO

ap_mdf Number of defined methods

Definition Number of defined member functions in the application.

Alias MDF

ap_nmm Number of member functions

Definition Number of member functions in the application.

Alias NMM

ap_npm Number of public methods

Definition Number of public methods in the application.

Alias NPM

ap_vg Sum of cyclomatic numbers

Definition Sum of cyclomatic numbers (i.e. ct_vg) for all the functions defined in the application.

Alias VGA, ap_cyclo

4.4.3 Application Call Graph

For more details on Call Graph Metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

ap_cg_cycle Call graph recursions

Definition Number of recursive paths in the call graph for the application's functions. A recursive path can be for one or more functions.

Alias GA_CYCLE

ap_cg_edge Call graph edges

Definition Number of edges in the call graph of application functions.

Alias GA_EDGE

ap_cg_leaf Call graph leaves

Definition Number of functions executing no call.
In other words, number of leaves nodes in the application call graph.

Alias GA_NSS

ap_cg_levl Call graph depth

Definition Depth of the Call Graph: number of call graph levels.

Alias GA_LEVL

ap_cg_maxdeg Maximum callers/called

Definition Maximum number of calling/called for nodes in the call graph of application functions.

Alias GA_MAXDEG

ap_cg_maxin Maximum callers

Definition Maximum number of "callings" for nodes in the call graph of Application functions.

Alias GA_MAX_IN

ap_cg_maxout Maximum called

Definition Maximum number of called functions for nodes in the call graph of Application functions.

Alias GA_MAX_OUT

ap_cg_node Call graph nodes

Definition Number of nodes in the call graph of Application functions. This metric cumulates Application's member and non-member functions as well as called but not analyzed functions.

Alias GA_NODE

ap_cg_root Call graph roots

Definition Number of roots functions in the application call graph.

Alias GA_NSP

4.4.4 Inheritance Tree

ap_inhg_cpx Inheritance tree complexity

Definition Thecomplexity of the inheritance tree is defined as a ratio between:

- the sum for all of the graph levels of the number of nodes on the level times the level weight index,
- the number of graph nodes.
- Basic classes are on the top level and leaf classes on the lower levels

Alias GH_CPX

ap_inhg_edge Inheritance graph edges

Definition Number of inheritance relationships in the application.

Alias GH_EDGE

ap_inhg_leaf Number of final class

Definition Number of final classes in the inheritance tree of the application.
A class is said to be a final class if it has no child class.

Alias GH_NSP

ap_inhg_levl Depth of inheritance tree

Definition The Depth of the Inheritance Tree (DIT) is the number of classes in the longest inheritance link.

Alias GH_LEVL

ap_inhg_maxdeg Maximum Number of derived/inherited classes

Definition Maximum number of inheritance relationships for a given class. This metric applies to the Application's inheritance graph.

Alias GH_MAX_ DEG

ap_inhg_maxin Maximum Number of derived classes.

Definition Maximum number of derived classes for a given class in the inheritance graph.

Alias GH_MAX_ IN

ap_inhg_maxout Maximum Number of inherited classes.

Definition Maximum number of inherited classes for a given class in the inheritance graph.

Alias GH_MAX_ OUT

ap_inhg_node Inheritance tree classes

Definition Number of classes present in the inheritance tree of the application.

Alias GH_NODE

ap_inhg_pc Protocol complexity

Definition Depth of the Inheritance Tree times the maximum number of functions in a class of the inheritance tree over the total number of functions in the inheritance tree

$$ap\_inhg\_pc = ap\_inhg\_levl \times \frac{MAX\ (LMPI + LMPO + LMPU)}{SUM\ (LMPI + LMPO + LMPU)}$$

Alias GH_PC

ap_inhg_root Number of basic classes

Definition Number of basic classes in the application. A class is said to be basic if it does not inherit from any other class.

Alias GH_NSS

ap_inhg_uri Number of repeated inheritances

Definition Repeated inheritances consist in inheriting twice from the same class. The number of repeated inheritances is the number of inherited class couples leading to a repeated inheritance.

Alias GH_URI

4.4.5 MOOD Metrics

The MOOD (Metrics for Object Oriented Design) set of metrics described in this chapter has been introduced by Fernando Brito e Abreu in "*Object-Oriented Software Engineering: Measuring and Controlling the Development Process*" (Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, October 1994).

Their definitions have been refined since their first introduction. The MOOD metrics computed by *Logiscope C++ QualityChecker* conform to the latest definitions and the corresponding C++ bindings described in "*Evaluating the Impact of Object-Oriented Design on Software Quality*" (Proceedings of the Third International Software Metrics Symposium, IEEE, Berlin, Germany, March 1996).

ap_mhf Method Hiding Factor (MHF)

Definition

$$ap\_mhf = \frac{\sum_{i=1}^{TC}\left[\sum_{m=1}^{M_d(C_i)}(1 - V(M_{mi}))\right]}{\sum_{i=1}^{TC} M_d(C_i)}$$

where:

$$V(M_{mi}) = \frac{\sum_{i=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1}$$

and:

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 \Leftrightarrow \begin{cases} j \neq i \\ C_j \text{ may call } M_{mi} \end{cases} \\ 0 \quad \text{otherwise} \end{cases}$$

The MHF numerator is the sum of the invisibilities of all methods defined in all classes. The invisibility of a method is the percentage of the total classes from which this method is not visible.

The MHF denominator is the total number of methods defined in the project.

The following C++ bindings are used to compute this metric:

MOOD		C++
TC	total classes	total number of classes
	methods	constructors; destructors; function members; operator definitions
$M_d(C_i)$	methods defined (not inherited)	all methods declared in the class including virtual (deferred) ones
$V(M_{mi})$	visibility - percentage of the total classes from which the method M_{mi} is visible	≡ 1 for methods in public clauses; ≡ 0 for methods in private clauses; = $DC(C_i)/(TC-1)$ for methods in protected clauses ($DC(C_i)$ = descendants of C_i)

ap_ahf Attribute Hiding Factor (AHF)

Definition

$$ap\_ahf = \frac{\sum_{i=1}^{TC}\left[\sum_{m=1}^{A_d(C_i)}(1 - V(A_{mi}))\right]}{\sum_{i=1}^{TC} A_d(C_i)}$$

where:

$$V(A_{mi}) = \frac{\sum_{i=1}^{TC} is\_visible(A_{mi}, C_j)}{TC - 1}$$

and:

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1 \Leftrightarrow \begin{cases} j \neq i \\ C_j \text{ may reference } A_{mi} \end{cases} \\ 0 \quad \text{otherwise} \end{cases}$$

The AHF numerator is the sum of the invisibilities of all attributes defined in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is not visible.

The AHF denominator is the total number of attributes defined in the project.

The following C++ bindings are used to compute this metric:

MOOD		C++
$A_d(C_i)$	attributes defined (not inherited)	data members
$V(A_{mi})$	visibility - percentage of the total classes from which the attribute A_{mi} is visible	= 1 for attributes in public clauses; = 0 for attributes in private clauses; = $DC(C_i)/(TC-1)$ for attributes in protected clauses ($DC(C_i)$ = descendants of C_i)

ap_mif Method Inheritance Factor (MIF)

Definition

$$ap\_mif = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

where:

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

The MIF numerator is the sum of inherited methods in all classes of the project.

The MIF denominator is the total number of available methods (locally defined plus inherited) for all classes.

The following C++ bindings are used to compute this metric:

MOOD		C++
$M_a(C_i)$	available methods	function members that can be invoked in association with C_i
$M_d(C_i)$	methods defined	function members declared within C_i
$M_i(C_i)$	inherited methods	function members inherited (and not overridden) in C_i

ap_aif Attribute Inheritance Factor (AIF)

Definition

$$ap\_aif = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

where:

$$A_a(C_i) = A_d(C_i) + A_i(C_i)$$

The AIF numerator is the sum of inherited attributes in all classes of the project.

The AIF denominator is the total number of available attributes (locally defined plus inherited) for all classes.

The following C++ bindings are used to compute this metric:

MOOD		C++
$A_a(C_i)$	available attributes	data members that can be invoked associated with C_i
$A_d(C_i)$	attributes defined	data members declared within C_i
$A_i(C_i)$	inherited attributes	data members inherited (and not overridden) in C_i

ap_pof Polymorphism Factor (POF)

Definition

$$ap\_pof = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

where:

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

The POF numerator is the sum of overriding methods in all classes. This is the *actual number of possible different polymorphic situations*. Indeed, a given message sent to a class can be bound, statically or dynamically, to a named method implementation. The latter can have as many shapes (morphos) as the number of times this same method is overridden (in that class's descendants).

The POF denominator represents the *maximum number of possible distinct polymorphic situations* for that class as the sum for each class of the number of new methods multiplied by the number of descendants. This value would be maximum if all new methods defined in each class would be overridden in all of their derived classes.

The following C++ bindings are used to compute this metric:

MOOD		C++
$DC(C_i)$	descendants count	number of classes descending from C_i
$M_n(C_i)$	new methods	function members declared within C_i that do not override inherited ones
$M_o(C_i)$	overriding methods	function members declared within C_i that override (redefine) inherited ones

ap_cof Coupling Factor (COF)

Definition

$$ap\_cof = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$$

where:

$$is\_client(C_i, C_j) = \begin{cases} 1 \Leftrightarrow C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 \quad \text{otherwise} \end{cases}$$

The COF denominator stands for the *maximum possible number of couplings* in a system with TC classes.

The client-supplier relation (represented by C_c ® C_s) means that C_c (*client* class) contains *at least one* non-inheritance reference to a feature (method or attribute) of class C_s (*supplier* class). The COF numerator then represents the *actual number of couplings not imputable to inheritance*.

Client-supplier relations can have several shapes:

Client-supplier shapes	**C++**
regular message passing	call to the interface of a function member in another class
"forced" message passing	call to a visible or hidden function member in another class by means of a friend clause
object allocation and deallocation	call to a class constructor or destructor
semantic associations among classes with a certain arity (e.g. 1:1, 1:n or n:m)	reference to a supplier class as a data member or as a formal parameter in a function member interface

Chapter 5

Programming Rules

This chapter describes the default programming rules that can be checked using Logiscope *C++ RuleChecker*.

They are made available through various standard Rule Sets provided with the Logiscope distribution. See §5.1.

Rule Sets and rules can be tailored to better fit to the effective coding standards and quality requirements applicable to the project or organisation: e.g. naming rules.
For more details, see Chapter Customizing Metrics & Rules.

5.1 Rule Sets

Logiscope *C++ RuleChecker* comes with several standard Rule Sets

They are made available through various Rules Set

The rules specified in section 5.2 can be customized by modifying parameters in the project Rule Set file (see Chapter Customizing Metrics & Rules).

The **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation has published a list of rules for the use of the C++ programming language for critical systems :

- *MISRA-C++:2008 Guidelines for the use of the C++ language critial systems* - June 2008 [MISRA-C++:2008].

Logiscope RuleChecker C++ provides the MISRAC++2008 Rule Set including some of the key MISRA C++ programming rules:
- Please refer to [MISRA-C++:2008] for a complete specification of the rules.

5.2 Rule Scripts

5.2.1 Basic Rules

asscal Assignment inside function calls

Description Assignment operators (=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --) shall not be used inside function calls.

Justification Removes ambiguity about the evaluation order.

asscon Assignment inside conditions

Description Assignment operators (=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --) shall not be used inside conditional expression in control statements `if`, `while`, `for` and `switch`.

Justification An instruction such as
`if (x=y) {...`
is ambiguous and unclear. One might think the author wanted to write
`if (x==y) {...`

Example:

```
// do not write
if (x -= dx) { ...
for (i=j=n; --i > 0; j--) {
..

// write
x -= dx;
if (x) { ...
for (i=j=n; i > 0; i--, j--)
{ ...
```

assexp Assignment inside expressions

Description Inside an expression:

- an *lvalue* has to be assigned only once,
- with multiple assignments, an assigned *lvalue* can appear only where it has been assigned.

Justification Removes ambiguity about the evaluation order.

Example:

```
// do not write
i = t[i++];
a=b=c+a;
i=t[i]=15;
```

blockdecl Declarations in Blocks

Description Declarations must appear at the beginning of blocks.

Justification Makes the code easier to read.

boolean Use Proper Boolean Expressions

Description The tests in control structures must contain proper boolean expressions.

Justification Makes the code easier to understand.

Example:

```
// do no write
while (1) {
  if (test) {
    for (i=1; function_call(i); i++) {

// write
AlwaysTrue = true;
while (AlwaysTrue == true) {
  if (test == true) {
    for (i=1; function_call(i); i++) {
```

brkcont Break and Continue Forbidden

Description `Break` and `continue` instructions are forbidden inside conditional expressions in control statements (`for`, `do`, `while`).

Nevertheless, the `break` instruction is allowed in the block instruction of the `switch` statement.

Justification Like a `goto`, these instructions break down code structure. Prohibiting them in loops makes the code easier to understand.

classuse Hidden class uses

Description Following expressions are not allowed: `u.v.a`, `u.v.f()`, `u.g().a`, `u.g().f()`, as well as expressions using the `->` operator.

Justification Prevents from calling a class method not known in the user class (hidden use), through calls in series.

Example:

```
// do not write
myWindow.itsButton.push();
```

Manipulate the *myWindow* object from the *Window* class; access to the *itsButton* attribute; directly call the *push* method on it. But only the *Window* class and its interface, containing *itsButton*, are normally known, and not the *itsButton* attribute class, neither its public methods (including *push*).

Example:

```
// do not write Error-
>pos.line;
```

There is a hidden use of *line*, which is not known from *Error*.

condop No ternary operator

Description The ternary conditional operator `? ... : ...` must not be used.

Justification Makes the code easier to read.

constrdef Default constructor

Description Each class must contain its default constructor explicitly.

Justification Makes sure the author has thought about the way to initialize an object of the class.

Example:

```
// write
class aClass {
...
aClass();
...
};
```

ctrlblock Blocks in Control Statements

Description Block statements shall always be used in control statements (`if`, `for`, `while`, `do`).

Justification Removes ambiguity about the scope of instructions and makes the code easier to read and to modify.

Example:

```
// do not write
if (x == 0) return;
else
    while (x > min)
        x--;

// write
if (x == 0) {
    return;
} else {
    while (x > min) {
      x--;
    }
}
```

delarray Use Delete [] For Array

Description Empty brackets must be used for delete when de-allocating arrays.

Justification Reliability: Ensures that the appropriate amount of memory is freed.

Example:

```
int *table = new int[7];
delete table;      // violation
delete [10] table; // violation
delete [] table;   // ok
```

Limitations There are some limitations to this rule when delete is used followed by a variable name.
These limitations do not apply in the case where delete is followed by a number in brackets.

This rule is not violated in the case of "complex" types:
Example 1:

```
int ** myarray = new int[2];
myarray[0] = new int[10];

delete myarray;     // violation
delete myarray[0];  // no violation
```

Example 2:

```
class A
{
  public:
    int *tab;
...
};
A var;
var.tab = new int[10];
delete var.tab;        // no violation
```

The rule is also not violated when the new operation is hidden:

```
int * create_array(int nb)
{
   return (new int[nb]);
}
...
int * myarray = create_array(10);
delete myarray;    // no violation
```

destr Destructor

Description Each class must contain its destructor explicitly.

Justification Reliability: being sure that the author has thought about the way to destroy an object of the class.

Example:

```
// write
class aClass {
...
~aClass(aClass &object);
...
};
```

fntype Function Types

Description Each function has to declare its type. If nothing is returned, it must be declared of `void` type.

Justification Portability.

forinit Initialize For Loop Counter In For Head

Description Loop counters (in for loops) are to be initialized in the initialization statement within the loop. The loop counter is determined by the third element of the loop head, which is most frequently used to increment the loop counter.
In all the following examples, `i` is the loop counter.

Justification This way the loop counter is certain to have been initialized, and with a value that is visible alongside with the loop condition and increment. The loop is easier to understand and to control.

Example:

```
for (int i = 0; i < 10; i++) ... // ok
for (int i; i < 10; i++) ...   // violation
for (int j = 0; j < 10; i++) ... // violation
for (int j = 10; i < j; i++) ... // violation
for (int j = 1; i < funct(j); i+=j) ... // violation
```

frndclass Friend Classes

Description If friend classes are used, they must be declared at the beginning of the class (before member declaration).

funcptr No Function Pointers

Description Do not use function pointers.

globinit Global Variable Initialization

Description Global variables must be initialized when they are defined.

Justification Not all compilers give the same default values. Unexpected behaviour can be avoided with better control over variable values. Initializing global variables when they are declared ensures that they are initialized before being used.

imptype Do Not Use Implicit Typing

Description Function, parameter, attribute or variable types must be declared explicitly.

This rule applies to non-ANSI compliant C++ code and should be turned off when using an ANSI compliant C++ compiler.

Justification Improves code portability.

Example:

```
// write
void aFunction(int value);

// do not write
aFunction(value);
```

macroparenth Parenthesis in Macro Definitions

Description Each occurrence of the macro parameters shall be enclosed in parenthesis (or braces) inside the macro definition.

Justification Makes the code easier to read.

Example:

```
// do not write
#define GET_NAME(obj,ind) obj->name[ind]

// write
#define GET_NAME(obj,ind) (obj)->name[ind]
```

mfunc Inline Functions instead of Macro-functions

Description Use inline functions instead of macro-functions.

Justification In comparison with macro-functions, inline functions enable the checking of their parameters types and do not allow side effects (such as `MIN(++i, j)` with the below example).

Example:

```
// write
inline char *GetName(aClass &object) {
return(object.name); }
inline min (int i, int j) { return (i<j)?i:j; }

// do not write
#define GetName(s) ((s)->name)
#define MIN(i,j) ((i)<(j)) ? (i) : (j)
```

multiass No Multiple Assignment

Description Assignment operators (=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --) must not be used more than once in each statement (declarations are also checked).

Justification Removes ambiguity about the evaluation order.

Example:

```
// do not write
b = c = 5;
a = (b++ * c) + 5;

// write
c = 5;
b = c;
b++;
a = (b * c) + 5;
```

nostruct Keyword Struct Not Allowed

Description The keyword struct may not be used. If the parameter is specified, only C-style structs may be used.

Parameters An optional string may be used (cstruct) to enable C-style structs to be used. When the cstruct rule is used, the possibilities allowed in C++ in a struct (such as access specifiers: private for example, or methods) are not to be used.

notemplate Avoid Using Templates

Description Do not use templates.

Justification Efficiency.

nothrow No Throw Instructions

Description No exceptions may be raised by the user (the keyword throw may not be used).

nounion No Union

Description The keyword union is not allowed.

parse Parse Error

Description This rule identifies module parts that could not be parsed.

Justification Enables to determine which portions of code have been analyzed and which portions of code have been rejected by Logiscope *C++ RuleChecker*.

pmfrtn Do Not Return Pointer To Member Data

Description Member functions must not return a pointer or a non-const reference to member data.

Justification Helps to ensure that data encapsulation is respected.

ptraccess Pointer Access

Description Use the ptr->fld syntax instead of the (*ptr).fld syntax.

ptrinit Pointers Initialization

Description Each auto variable that is explicitly declared as a pointer (using "*"), must be initialized when declared.

Justification Makes sure pointer variables are correctly initialized before being used.

Example:

```
// write
int* y=&x;
...

// do not write
int *y ;
*y=&x ;
 ...
```

rtnlocptr Do Not Return Pointer To Local Variable

Description Functions must not return a pointer to a non-static local variable.

Justification This avoids dangling references of the pointer to the variable after its lifetime.

sgdecl A Single Variable per Declaration

Description Variable declarations have the following formalism:

```
type variable_name;
```

It is forbidden to have more than one variable for the same type declarator.

Justification Makes the code easier to read.

Example:

```
// write
int width;
int length;

// do not write
int width, length;
```

sglreturn A Single Return per Function

Description Only one return instruction is allowed in a function.

Justification Maintainability : a basic rule for structured programming.

slcom Use // Comments

Description /* */ comments are forbidden. Use only // comments.

Justification Makes the code easier to read.

slstat One Statement per Line

Description There must not be more than one statement per line.

A statement followed by a curly bracket (`instr {`) or a curly bracket followed by a statement (`{ instr`) is allowed in the same line, but not both of them (`instr { instr`).

Justification Makes the code easier to read.

Example:

```
// write
x = x0;
y = y0;
while (IsOk(x)) {
   x++;
}

// do not write
x = x0; y = y0;
while (IsOk(x)) {x++;}
while (IsOk(x)) {x++;
}
```

typeinher Inheritance Type

Description The inheritance type (public, protected, private) must be specified.

Justification Analysability

Example:

```
class inherclass : public Base1, private Base2
{...
```

vararg Variable Number of Arguments

Description Functions with a variable number of arguments are not allowed.

Parameters of `va_list` type and `...` are forbidden in function declarations.

Justification Makes the code easier to understand.

voidptr No Void Pointer

Description The void pointer (void *) should not be used.

varinit All Variables Must Be Initialized Before Being Used

Description All variables must be initialized before they are used, without counting on the default value attributed by the compiler. Global variables, parameters of a function in the function body, and data fields of a class in its methods are considered to be initialized.

Justification Not all compilers give the same default values.
Unexpected behaviour can be avoided with better control over variable values.

Limitations This rule is not violated in the following cases:

- If an array, a struct or a class are used, they will be considered initialized as soon as a part of them has been initialized.
  For example:

```
int a[2];
int b[2] = {6, 7};
int h;

a[0] = b[0];   // no violation
h = a[1];      // no violation

struct {
    int i;
    int j;
} e, f;
e.i = 0;
g = e;      // no violation
```

This rule is violated in the following cases where initialization is uncertain:

- Using a variable in a function call is considered as "being used": if it is not initialized, the rule will be violated. This will occur whatever the use of the function, even initializing the variable.
- In cases including a conditional initialization, the rule is violated even though the variable may well be initialized.

```
int i, j, k;
j = func();
if (j)
   i = 0;
k = i;       // violation
```

- This applies even when there is an else branch:

```
int i, j, k;
j = func();
if (j)
   i = 0;
else
   i = 5;
k = i;      // violation
```

  where initialization is certain.

- In the case of a loop, for example:

```
int j, k;
for (int i=0; i<glob; i++)
 {
  j=func(i);
 }
k = j;    // violation
```

  where `glob` is a global variable, depending on the value of `glob`, `j` will have been initialized or not: the rule is violated, whether the loop condition occurs or not.

5.2.2 Customizable Rules

The rules specified in this section have parameters that can be modified to better fit to the effective programming rules applicable to the project or organisation: e.g. naming rules.

Please, refer to the Chapter *Customizing Metrics & Rules* for more details on how tailoring rule checking using Rule Set file.

ansi Function Declarations in ANSI Syntax

Description Function declaration and definition shall be written in ANSI syntax.

It is possible to select two options among the following:

- **name**: parameters shall be named and their type indicated in function declaration,
- **void**: empty parameter lists are forbidden.

By default, both options are selected.

Parameters A list of character strings composed of chosen options listed above.

Justification Makes the code easier to read and improves its portability.

Example:

```
// do not write
f(int, char*);
f();

// write
f(int a, char *b);
f(void);
```

cmclass A Single Class per Code File

Description In a code file, every function must belong to the same class.

A C function is considered to belong to the main class.
The first funtion encountered in the file sets the class for that file.

By default, a code file has one of the suffixes *.cc, *.cxx, *.cpp, *.C or *.c.

Parameters A string representing the types of modules (metric type) that should be considered as code files.

Justification Makes the code easier to read.

Limitation Friend functions of a class that don’t have a scope are considered to belong to the main class.

cmdef Classes in Code File

Description A code file must not contain any class declaration.

A C function is considered to belong to the main class.

By default, a code file has one of the suffixes *.cc, *.cxx, *.cpp, *.C or *.c.

Parameters A string representing the types of modules (metric type) that should be considered as code files.

Justification Makes the code easier to read.

const Literal Constants

Description Numbers and strings have to be declared as constants instead of being used as literals inside a program.

Specify allowed literal constants. By default allowed literal constants are "", " ", "0" and "1".

Parameters A list of character strings representing allowed literal constants.
A special parameter can be used: LOG_SWITCH_CONST. If present, it must be the first parameter of the list. When activated it allows constants to be used in switch cases.

Justification Makes maintenance easier by avoiding the scattering of constants among the code, often with the same value.

Note In the case of constants used in initializing lists (concerning array and `struct` structures), only the first five violations are detected.

Example:

```
// do not write
char tab[100];
int i;
...
if (i == 7) {
    p = "Hello World.\n";
}

// write
#define TAB_SIZE 100
enum i_val { ok =7; ko =11};
const char HelloWorld[] = "Hello World.\n";
char tab[TAB_SIZE];
i_val i;
...
if (i == ok) {
    p = HelloWorld;
}
```

constrcpy Copy Constructor

Description Each class must contain its copy constructor explicitly.

Parameters The string "dynalloc" which, if used, indicates that the rule has to be checked only if there is a class member which is a pointer

Justification Makes sure the author has thought about the way to copy an object of the class.

Example:

```
// write
class aClass {
...
aClass(const aClass &object); // "const" is optional
...
};
```

dmaccess Access to Data Members

Description The class interface must be purely functional: data members definitions can be limited.

By default, only the data members definition in the public part of a class are forbidden.

Parameters A list of character strings corresponding to the forbidden access specifiers for the data members.

Justification The good way to modify the state of an object is via its methods, not its data members. The data members of a class should be private or at least protected.

exprcplx Expressions Complexity

Description Expressions complexity must be smaller than a limit given as a parameter. This complexity is calculated with the associated syntactic tree, and its number of nodes.

By default, the maximum authorized complexity level is 13.

Parameters A number representing the maximum authorized complexity level.

Justification Makes the code easier to read.

Example:

For instance, this expression:

```
(b+c*d) + (b*f(c)*d)
```

is composed of 8 operators and 7 operands.

The associated syntactic tree has 16 nodes, so if the limit is under 16, there will be a rule violation.

exprparenth Parentheses in Expressions

Description In expressions, every binary and ternary operator shall be put between parentheses.

It is possible to limit this rule by using the **partpar** option. The following rule is then applied: when the right operand of a "+" or "*" operator uses the same operator, omit parentheses for it. In the same way, omit parentheses in the case of the right operand of an assignment operator. Moreover, omit parentheses at the first level of the expression.

By default, the **partpar** option is selected.

Parameters The character string "**partpar**", which, if used, allows programmers not to put systematically parentheses, according to the rule above.

Justification Reliability, Maintainability: Removes ambiguity about the evaluation priorities.

Example:

```
// do not write
result = fact / 100 + rem; // Violation
// write
result = ((fact / 100) + rem); // Ok
// or write, with the partpar option
result = (fact / 100) + rem;
// with the partpar option, write
result = (fact * ind * 100) + rem + 10 + power(coeff,c);
// instead of
result = ((fact * (ind * 100)) + (rem + (10 + power(coeff,c))));
```

funcres Reserved Functions

Description Certain names cannot be used for the declaration or definition of functions, and for function calls.

By default, no function names are forbidden.

Parameters A list of character strings representing the function names considered as reserved.

Justification Portability: Prevents from the use of system functions that are non portable or dangerous.

Example:

```
// if the system function is forbidden, do not write
int
system(char *command);
int
system(char *command)
{
...
}
system("cp file /tmp");
```

goto Goto Statement

Description The `goto` statement must not be used.

By default, all `goto` statements are forbidden.

Parameters A list of strings specifying labels which are authorized with the `goto` statement.

Justification Maintainability Insures that structured programming rules are respected, so the code is easier to understand. The `goto` statement often reveals an analysis error and its systematic rejection improves the code structure.

Headercom Module Header Comment

Description Modules must be preceded by a header comment.

It is possible to define a format for this comment depending on the type of the module as it is defined in metric type.

By default, a header comment with the name of the file, its author, its date and possible remarks is required for header and code files (see below example).

Parameters Two lists of character strings: the first one for the header files, and the second for the code files. Each list begins with the string "HEADER" or "CODE", followed by strings representing the associated regular expressions.

Justification Makes the code easier to read.

Example of the default required header comment:

```
////////////////////////////////////////
/////
// Name: program
// Author: Andrieu
// Date: 08/07/96
// Remarks: example of comments
////////////////////////////////////////
/////
```

headercom Function and Class Header Comments

Description Functions and classes must be preceded by a comment.

It is possible to define a format for this comment depending on the type of the function definition or declaration, or class definition (func_glob_def, func_glob_decl, func_stat_def, func_stat_decl, class).

By default, only a comment beginning with "/*" is required for functions or classes.

Parameters Five lists of character strings concerning the five cases listed above. Each list begins with one of the five strings (func_glob_def for instance), followed by a string representing the regular expression.

Justification Makes the code easier to read.

hmclass A Single Class Definition per Header File

Description A header file must not contain more than one class definition.

Nested classes are tolerated.

By default, a header file corresponds to the filter *.{h,hh,H,hxx,hpp}.

Parameters A string representing types of modules (metric type) that should be considered as header files.

Justification Makes the code easier to read.

hmdef Header File Contents

Description Header files may not contain some of language statements (data and function definitions).

The forbidden language items are function definitions (func-stat-def, func-glob-def) and data definitions (var-stat, var-glob).

By default, a header file corresponds to the filter *.{h,hh,H,hxx,hpp}.

Parameters A string representing types of modules (metric type) that should be considered as header files.

Justification The implementation of a class should not be found in header files.

hmstruct Header File Structure

Description The main structure of header files should be:

```
#ifndef <IDENT>
#define <IDENT>
...
#endif
```

or

```
#if !defined (<IDENT>)
#define <IDENT>
...
#endif
```

where `<IDENT>` is an identifier built from the name of the header file.

The comparison is made only on alphanumeric characters and is not case sensitive.

The part of the filename taken into account is between the MINth and the MAXth characters (including them). This character string should be found in the identifier according to the above comparison rules.

By default, the MIN value is 1 and the MAX value is 999 and a header file corresponds to the filter *.{h,hh,H,hxx,hpp}.

Parameters A MINMAX couple of values giving the part of the filename to take into account, and a list of character strings giving the list of file types to be considered as header files for this rule. The types are those defined by the metric type.

Justification Prevents multiple inclusions of header files.

Example:

```
// if the parameter is MINMAX 4 9,the following contents
// of file div_audit_env.h is correct
#ifndef AUDIT_H
#define AUDIT_H
...
#endif
```

identfmt Identifier Format

Description The identifier of a function, type or variable declared in a module must have a format corresponding to the category of the declaration.

By default, the only restrictions concern the constants and the macros, which must have no lower case letter.

Parameters A list of couples of character strings; the first string of the couple represents the declaration category name, the second one the regular expression associated to that category.

Justification Makes the code easier to understand.

identl Identifier Length

Description The length of a function, type or variable identifier has to be between a minimum and a maximum value.

By default, the methods and functions must have between 4 and 25 characters, the types, variables, constants, macros and classes between 5 and 25, and the other identifiers between 1 and 25.

Parameters A list of couples of character strings; the first string of the couple represents the declaration category name, the second one the MINMAX expression associated.

Justification Makes the code easier to read.

identres Reserved Identifiers

Description Some identifiers may be forbidden in declarations. For instance, names used in compilation directives or in libraries.

By default, there are no reserved identifiers.

Parameters A list of character strings representing reserved identifiers.

Justification Improves code portability.

incltype Included Modules Type

Description Only some types of modules are allowed to be included in other modules.

By default, header modules can be included in header and code modules.

Parameters Lists of lists of character strings, each list being comprised of a string representing a type of module (metric type), followed by strings representing the types of modules that may be included in it.

Justification Improves code structuring.

inldef Inline Functions Declaration and Definition

Description Inline functions must be declared in their class and defined outside of it.

Parameters The string “private” which is an optional parameter. When the parameter is used, private inline functions must be defined in the class definition file (.cpp file), other inline functions must be defined in the class declaration file (.h file).

Justification Makes the code easier to read.

macrocharset Characters Used in Macros

Description Some characters may be forbidden in the writing of the definitions of macro-functions and macro-constants (not in their name).

The two cases are treated separately.

By default, no characters are forbidden in macros.

Parameters A list of two couples of character strings; the first string of the couple is "constant" or "function", and the second one a string composed by the associated forbidden characters.

Justification Improves code portability.

mconst Macro Constant Usage

Description The usage of macro constants shall be limited.

It is possible to choose between three options:

- **var**: global or static variables are used for string constants, other constants could be defined by macros (this is the default option),

Example:

```
// write
const char *string = "Hello world!\n";
#define value 3

// do not write
#define string "Hello world!\n"
```

- **const**: `const` data are always used instead of macros,

Example:

```
// write
const char *string = "Hello world!\n";
const int value = 3;

// do not write
#define string "Hello world!\n"
#define value 3
```

- **nodefine**: only compilation flags and macro functions are allowed.

Example:

```
// write
#define VERBOSE
#define min(x,y) ((x)<(y)?(x):(y))

// do not write
#define value 3
#define current_value f(tab[0])
```

Parameters One of the three character strings explained above.

Justification Limits the use of macro-constants.

mname File Names

Description A file name and the name of the class declared or defined in this file must be closely related.

The comparison is made only on alphanumeric characters and is not case sensitive.

The extension of the file name is not taken into account.

The part of the file name taken into account to correspond to the name of the class is between the MIN and the MAX characters (these included). This character string should be found in the identifier according to the above comparison rules.

By default, the part of the file name taken into account is between the characters 1 and 5.

Parameters A MINMAX couple of values giving the part of the file name to take into account.

Justification Makes the application easier to understand.

Example:

```
if the MINMAX parameters are 4 and 10, and the file name
is
    My_Graph_Node.h
then the part of the file name that should be found in
the class name is:
    GRAPHN
(the first 10 characters: My_Graph_N,
minus the first 3: Graph_N,
minus non alphanumeric characters: GraphN)

Then, the class name that the file is based upon could
be one of the following declarations
    class CLA_Graph_Node { ...}
    class Graph_Node { ...}
    class Graph_Node_Def { ...}
    class graphnode { ...}
But not the following ones
    class Graph { ...}
    class NodeGraph { ...}
```

nopreproc No Pre-processing Instructions

Description No pre-processing instructions may be used, except for those specified in the parameter list.

Parameters A list of strings defining the exceptions to this rule. The list can be empty. By default, only #line and # alone may not be used.
“define”: #define may be used
“include”: #include may be used
“if”: #if, #ifdef and #ifndef may be used
“pragma”: #pragma may be used
“undef”: #undef may be used
“line”: #line may be used
“error”: #error may be used
“none”: # may be used alone

Justification Makes the code easier to read and understand.

operass Assignment Operator

Description Each class must explicitly contain at least one assignment operator.

Parameters The string "dynalloc" which, if used, indicates that the rule has to be checked only if there is a class member which is a pointer

Justification Makes sure the author has thought about the way to assign an object of the class.

Example:

```
// write
class aClass {
...
operator = (const aClass &object); // "const" is optional
...
};
```

parammode Parameters Mode

Description In function definitions, the parameters mode used (`IN`, `OUT` or `INOUT`) must be indicated.

By default, the three modes "IN", "OUT" and "INOUT" are authorized.

Parameters A list of character strings representing the authorized keywords (their order does not matter).

Justification Enables to control parameter passing.

Example:

```
// write
int Multiply(IN Matrix *m, IN Vector *v, OUT Matrix *result);
```

sectord "public", "private" and "protected" Sections Order

Description In a class declaration, sections defined by the access specifiers must follow a particular order, given in the parameters of the rule.

An empty string can be used (in the first position), representing the first section without any specifier.

Note Class definitions have not to contain all the access specifiers defined in the standard.

By default no particular order is given.

Parameters A list of character strings representing the access specifiers in the wanted order.

Justification Makes the code easier to read.

Example:

```
// if the standard has the following strings in this order:
// "", "private", "protected" and "public",
// following declarations are allowed
class aClass {
     int i ;
  protected:
     void p();
     };
class aClass {
  protected:
     int i ;
  public:
     void p();
     };

// and not the following ones:
class aClass  {
   protected:
     ...;
   private:
     ... ;
   };
class aClass  {
   protected:
     ...;
   protected:
     ... ;
   };
```

sgancstr Single Ancestor

Description All classes must have a same direct or indirect ancestor. The ancestor can be specified as a parameter.

Parameters A string representing the name of the ancestor. The parameter is optional.

swdef default within switch

Description A `default` case is mandatory within a `switch` in order to cover unexpected cases.

By default, the `default` case has to be the last one.

Parameters The character string "last", which, if used, specifies that the `default` case has to be the last one.

Justification All cases must be provided for in a `switch`.

swend End of Cases in a "switch"

Description Each case in a `switch` shall end with `break`, `continue`, `goto`, `return` or `exit`. Several consecutive case labels are allowed.

By default, such instructions are not mandatory for the last case.

Parameters The character string "nolast", which, if used, allows not to have one of these instructions in the last case.

Justification Makes the code easier to understand and reduces the risk of errors.

varstruct Struct and Union Variables

Description Variables must not be directly declared using a `struct` or an `union` structure.

An intermediate type must be automatically used.

Parameters The string "nostruct" which, if used, prevents from declaring a `struct` or `union` variable except in a `typedef` structure.

This option has no meaning in C++ programs, where class declarations are always allowed outside a `typedef` structure.

Justification Makes the code easier to understand.

Example:

```
// write
typedef struct {
    ...
} typeName;
typeName varName;
struct structName;
typedef struct structName {
     ...
     struct structName *ptr;
} typeName;
typeName varName;

// do not write
struct {
    ...
} varName;
// do not write, if the "nostruct" option is used
struct structName {
    ...
};
struct structName varName;
```

typeres Reserved Types

Description Some types may be forbidden for variables or functions.

It is possible to define the list of types that are forbidden for variables (`extern`, `static`, and automatic variables) and the list of types that are forbidden for functions.

The type specifiers and qualifiers are forbidden in any order and even if they are merged with other specifiers or qualifiers.

These types are allowed in `typedef` definition.

Parameters Two lists of strings beginning by the keywords "data" or "function". The other items of the list are strings containing the forbidden groups of type specifiers or type qualifiers separated by spaces (' ').

Justification Not relying on predefined types improves code portability.

5.2.3 MISRA-C++ 2008 Programming Rules

The **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation has published a list of rules for the use of the C++ programming language for critical systems :

- *MISRA-C++:2008 Guidelines for the use of the C++ language critial systems* - June 2008 [MISRA-C++:2008].

Logiscope RuleChecker C++ provides the MISRAC++2008 Rule Set including some of the key MISRA C++ programming rules:
- Please refer to [MISRA-C++:2008] for a complete specification of the rules.

MISRA_0_1_1

A project shall not contain unreachable code.

MISRA_0_1_10

Every defined function shall be called at least once.

MISRA_0_1_3

A project shall not contain unused variables.

MISRA_0_1_4

A project shall not contain non-volatile POD variables having only one use.

MISRA_0_1_5

A project shall not contain unused type declarations.

MISRA_2_10_1

Different identifiers shall be typographically unambiguous.

MISRA_2_10_3

A typedef name (including qualification, if any) shall be a unique identifier.

MISRA_2_10_4

A class, union or enum name (including qualification, if any) shall be a unique identifier.

MISRA_2_10_5

The identifier name of a non_member object or function with static storage duration should not be reused.

MISRA_2_13_1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

MISRA_2_13_2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

MISRA_2_13_4

Literal suffixes shall be upper case.

MISRA_2_3_1

Trigraphs shall not be used.

MISRA_2_7_1

The character sequence /* shall not be used within a C-style comment.

MISRA_2_7_2

Sections of code shall not be "commented out" using C-style comments.

MISRA_2_7_3

Sections of code should not be "commented out" using C++ comments.

MISRA_3_1_1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

MISRA_3_1_3

When an array is declared, its size shall either be stated explicitly or defined implicitly

by initialization.

MISRA_3_9_2

typedefs that indicate size and signedness should be used in place of the basic numerical types.

Note: Renaming of the standard **typeres** rule with the following parameters:

LIST "data" "char" "int" "short" "long" "float" "double" "long double" END LIST

LIST "function" "char" "int" "short" "long" "float" "double" "long double" END LIST

MISRA_5_0_2

Limited dependence should be placed on C++ operator precedence rules in expressions.

Note: Renaming of the standard **exprparenth** rule with the parameter “partpar”:

MISRA_5_14_1

The right hand operand of a logical && or || operator shall not contain side effects.

MISRA_5_18_1

The comma operator shall not be used.

MISRA_5_2_1

Each operand of a logical && or || shall be a postfix-expression.

MISRA_5_2_10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

MISRA_5_2_11

The comma operator, && operator and the || operator shall not be overloaded.

MISRA_5_2_4

C_style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

MISRA_5_2_5

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

MISRA_5_3_3

The unary & operator shall not be overloaded.

MISRA_6_2_2

Floating_point expressions shall not be directly or indirectly tested for equality or inequality.

MISRA_6_2_3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white_space character.

MISRA_6_3_1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Note: Renaming of the standard **ctrlblock** rule.

MISRA_6_4_1

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

MISRA_6_4_2

All if ? else if constructs shall be terminated with an else clause.

MISRA_6_4_4

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

MISRA_6_4_5

An unconditional throw or break statement shall terminate every non-empty switch-

clause.

MISRA_6_4_6

The final clause of a switch statement shall be the default-clause.

MISRA_6_4_8

Every switch statement shall have at least one case- clause.

MISRA_6_5_1

A for loop shall contain a single loop-counter which shall not have floating type.

Limitation: Loop-counters modified within the for body are not accounted for when evaluating this rule

MISRA_6_5_2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

MISRA_6_5_3

The loop-counter shall not be modified within condition or statement.

MISRA_6_5_4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

MISRA_6_6_1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

MISRA_6_6_2

The goto statement shall jump to a label declared later in the same function body.

MISRA_6_6_4

For any iteration statement there shall be no more than one break or goto statement used

for loop termination.

MISRA_6_6_5

A function shall have a single point of exit at the end of the function.

MISRA_7_3_2

The identifier main shall not be used for a function other than the global function main.

Parameter: A list of non usable function names .
For instance, on a Windows platform,

LIST "main" "WinMain" "DllMain" END LIST

Default: Only “main” is considered.

MISRA_7_3_3

There shall be no unnamed namespaces in header files.

Parameter: A list of accepted header files extensions.

ex. LIST ".h" ".hpp" END LIST

Default: Only files with extensions “.h”, “.hxx” and “.hh” are considered as header files.

MISRA_7_3_4

Using-directives shall not be used.

MISRA_7_3_6

Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

MISRA_7_4_3

Assembly language shall be encapsulated and isolated.

MISRA_7_5_1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

MISRA_7_5_2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA_8_0_1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Note: Renaming of the standard **sgdecl** rule.

MISRA_8_3_1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

MISRA_8_4_1

Functions shall not be defined using the ellipsis notation.

Note: Renaming of the standard **vararg** rule.

MISRA_8_4_3

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
Limitations*:* The rule only checks that all non-void functions have at least one return statement, and that all of its return statements return an expression

MISRA_8_5_1

All shall have a defined value before they are used.

MISRA_8_5_3

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

MISRA_9_5_1

Unions shall not be used.

Note: Renaming of the standard **nounion** rule.

MISRA_9_6_2

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.
Note: In addition of wchar_t, forbidden types (because potentially signed or unsigned) are int, char, short, long.

MISRA_9_6_4

Named bit-fields with signed integer type shall have a length of more than one bit.

MISRA_10_1_1

Classes should not be derived from virtual bases.

MISRA_10_2_1

All accessible entity names within a multiple inheritance hierarchy should be unique.

MISRA_10_3_1

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

MISRA_10_3_2

Each overriding virtual function shall be declared with the virtual keyword.

MISRA_10_3_3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

MISRA_11_0_1

Member data in non-POD class types shall be private.

MISRA_12_1_2

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.

Limitation: The rule checks that base class constructors are called in the initialization list, separated by colons. It does not check that base class constructors are called within the constructor body.

MISRA_12_1_3

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

The fundamental types are:

- bool
- char "unsigned char" "signed char"
- short "short int" "signed short" "unsigned short"
- int "signed int" "unsigned int"
- __int8 int16 int32 int64
- long "signed long" "unsigned long" "long long" "signed long long" "unsigned long long"
- float double "long double"
- __wchar_t

Parameter: A list of additional types.
ex. LIST "_UI8" "_UI16" "_UI32" "_UI64" END LIST

MISRA_14_7_1

All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.

MISRA_15_0_2

An exception object should not have pointer type.

MISRA_15_0_3

Control shall not be transferred into a try or catch block using a goto or a switch statement.

MISRA_15_1_1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

MISRA_15_1_2

NULL shall not be thrown explicitly.

MISRA_15_1_3

An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

MISRA_15_3_1

Exceptions shall be raised only after start-up and before termination of the program.

MISRA_15_3_7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

MISRA_15_5_1

A class destructor shall not exit with an exception.

MISRA_15_5_3

The terminate() function shall not be called implicitly.

MISRA_16_0_1

#include directives in a file shall only be preceded by other preprocessor directives or comments.

MISRA_16_0_2

Macros shall only be #define'd or #undef'd in the global namespace.

MISRA_16_0_3

#undef shall not be used.

MISRA_16_0_4

Function-like macros shall not be defined.

Note: Renaming of the standard **mfunc** rule.

MISRA_16_0_5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

MISRA_16_0_6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

MISRA_16_0_8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

MISRA_16_1_1

The defined preprocessor operator shall only be used in one of the two standard forms.

MISRA_16_1_2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

MISRA_16_2_1

The pre-processor shall only be used for file inclusion and include guards.

MISRA_16_2_2

C++ macros shall only be used for include guards, type qualifiers, or storage class specifiers.

MISRA_16_2_3

Include guards shall be provided.

MISRA_16_2_4

The ', ", /* or // characters shall not occur in a header file name.

MISRA_16_2_5

The \ character should not occur in a header file name.

MISRA_16_2_6

The #include directive shall be followed by either a <filename> or "filename" sequence.

MISRA_16_3_1

There shall be at most one occurrence of the # or ## operators in a single macro definition.

MISRA_16_3_2

The # and ## operators should not be used.

MISRA_17_0_2

The names of standard library macros and objects shall not be reused.

Parameter: The list of non reusable names.
Default:

LIST

"NULL" "EXIT_SUCCESS" "EXIT_FAILURE" "RAND_MAX" " max" " min" "_MAX_PATH" "_MAX_DRIVE" "_MAX_DIR" "_MAX_FNAME" "_MAX_EXT" "_OUT_TO_DEFAULT" "_OUT_TO_STDERR" "_OUT_TO_MSGBOX" "_REPORT_ERRMODE" "errno" "_doserrno" " argc" " argv" " wargv" "_environ" "_wenviron" "_pgmptr" "_wpgmptr" "size_t" "wchar_t" "_onexit_t" "div_t" "ldiv_t"

END LIST

MISRA_17_0_3

The names of standard library functions shall not be overridden.

Parameter: The list of non overridden names.

MISRA_17_0_5

The setjmp macro and the longjmp function shall not be used.

Note: Renaming of the standard **funres** rule with the following parameter:

LIST “setjmp” “longjmp” END LIST

MISRA_18_0_1

The C library shall not be used.

Parameter: The list of non reusable names.

MISRA_18_0_2

The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Note: Renaming of the standard **funres** rule with the following parameter:

LIST "atof" "atoi" "atol" END LIST

MISRA_18_0_3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Note: Renaming of the standard **funres** rule with the following parameter:

LIST “abort” “exit” “getenv” “system” END LIST

MISRA_18_0_4

The time handling functions of library <ctime> shall not be used.

MISRA_18_0_5

The unbounded functions of library <cstring> shall not be used.

Note: Renaming of the standard **funres** rule with the following parameter:

LIST "strcpy" "strcmp" "strcat" "strchr" "strspn" "strcspn" "strpbrk" "strrchr" "strstr" "strtok" "strlen" END LIST

MISRA_18_2_1

The macro offsetof shall not be used.

Note: Renaming of the standard **funres** rule with the following parameter:

LIST “offsetof” END LIST

MISRA_18_4_1

Dynamic heap memory allocation shall not be used.

MISRA_18_7_1

The signal handling facilities of <csignal> shall not be used.

MISRA_19_3_1

The error indicator errno shall not be used.

MISRA_27_0_1

The stream input/output library <cstdio> shall not be used.

5.3 Scott Meyers Rules

The following rules come from two books written by Scott Meyers: "*Effective C++: 50 Specific Ways to Improve Your Programs and Designs*" (Addison-Wesley, second edition, 1997, ISBN: 0-201-92488-9) and "*More Effective C++: 35 New Ways To Improve Your Programs And Designs*" (Addison-Wesley, first edition, 1996, ISBN: 0-201-63371-X).

assignthis Check for Assignment to "self" in Operator "="

This rule relates to Item 17 in "Effective C++".

Description Inside the definition of an assignment operator:
- the equality between the parameter and `this` or `*this` shall be checked;
- in case of equality, `*this` must be returned..

Justification Ensures that self-assignment will work.

cast Prefer C++-style Casts

This rule relates to Item 2 in "More Effective C++"

.

Description Use the C++-style casts (`static_cast`, `const_cast`, `dynamic_cast` and `reinterpret_cast`) instead of the general-purpose C-style cast.

Justification The C-style cast does not allow to make a distinction between the different types of casts and it is not easy to detect.

catchref Catch Exceptions by Reference

This rule relates to Item 13 in "More Effective C++"

.

Description In `catch` clauses references to exceptions must be indicated.

Justification Improves code efficiency.

constrinit Prefer Initialization to Assignment in Constructors

This rule relates to Item 12 in "Effective C++".

Description Non static data members must be initialized inside the member initialization list of the constructor(s) of the class.

Justification Improves code efficiency.

convnewdel Adhere to Convention when Writing "new" and "delete" Operators

This rule relates to Item 8 in "Effective C++".

Description Convention for writing operator `new`:
- the type of the return value shall be `void *`;
- the type of the first parameter shall be `size_t`.

Convention for writing operator `delete`:
- the type of the return value shall be `void`;
- the type of the first parameter shall be `void *`;
- in case of a second parameter, its type shall be `size_t`.

Parameters The string "static" which, if used, indicates that operator `new` and operator `delete` shall be declared `static`.

Justification Keeps the consistency with the default `new` and `delete` operators.

dataptr Data of Pointer Type

This rule relates to Item 10 in "More Effective C++".

Description Class members which are pointers to objects are not allowed.

Justification Prevents resource leaks in constructors and simplifies destructors definitions.

delifnew Write Operator "delete" if you Write Operator "new"

This rule relates to Item 10 in "Effective C++".

Description If operator `new` is declared inside a class, then operator `delete` shall be also declared inside the same class.

Justification `new` and `delete` operators work together.

excepspec Exception Specifications

This rule relates to Item 14 in "More Effective C++"

Description Do not use exception specifications.

Justification Prevents violations of exception specifications, which are dangerous.

inlinevirt Inline Virtual Functions

This rule relates to Item 24 in "More Effective C++"

Description Virtual functions shall not be declared `inline`.

Justification Improves code efficiency.

multinher Multiple Inheritance Only Allowed for Inheriting Abstract Classes

This rule relates to Item 43 in "More Effective C++".

Description If multiple inheritance is used, the classes inherited must be abstract, that is to say that they must contain at least one pure virtual method.

Justification Makes the overall design less complicated and the code easier to understand.

Example:

```
1st case:
     A and B are not abstract classes (they contain no pure
     virtual methods). C inherits A and B: the rule is vio-
     lated.

2nd case:
     A and B are abstract classes (they contain at least one
     pure virtual method each). C inherits A and B: the rule
     is not violated.

     Current limitation of this case:
     If class C remains abstract (A and/or B's pure virtual
     methods are not redefined in C) and if a class D inherits
     C and another abstract class, the rule will be violated
     for D, although it inherits only abstract classes.

3rd case:
     A is abstract, B is not, C is (has a pure virtual func-
     tion), and inherits A and B. C violates the rule, but is
     abstract for inheriting classes.
```

nonleafabs Make non-leaf classes abstract

This rule relates to Item 33 in "More Effective C++".

Description Non-leaf classes shall be abstract.

Justification Helps assignment do what most programmers expect and improves the design of classes.

normalnew Avoid Hiding the "Normal" Form of "new"

This rule relates to Item 9 in "Effective C++".

Description If operator `new` is declared one or several times inside a class, at least one of these declarations shall follow the "normal" form:
- the type of the first parameter shall be `size_t`;
- all other parameters, if any, shall have a default value.

Justification Lets the usual invocation form of `new` available.

overload Never overload "&&", "||" and "," operators

This rule relates to Item 7 in "More Effective C++".

Description "&&", "`||`" and "`,`" operators must not be overloaded.

Justification Makes the code do what most programmers expect.

prepost Distinguish between Prefix and Postfix Forms of Increment and Decrement Operators

This rule relates to Item 6 in "More Effective C++"

Description Increment and decrement operators must be declared in the same manner as in the following example:

```
class Example {
 public:
  Example& operator++();           // prefix ++
  const Example operator++(int);   // postfix ++
  Example& operator--();           // prefix --
  const Example operator--(int);   // postfix --
}
```

Justification Keeps the consistency with built-in types.

refclass References of Classes

This rule relates to Item 22 in "Effective C++".

Description Every parameters of class type shall be passed by reference.

Justification Improves the efficency of the code.

returnthis Return "*this" in Assignment Operators

This rule relates to Item 15 in "Effective C++"

Description Inside the definition of an assignment operator, the return value shall be `*this`.

Justification Allows chains of assignments and type conversions.

tryblock Try Blocks

This rule relates to Item 15 in "More Effective C++".

Description Do not use try blocks.

Justification Efficiency.

trydestr Try Blocks in Destructors

This rule relates to Item 11 in "More Effective C++".

Description If it is explicit, the definition of a destructor must contain a `try` and `catch` block.

Justification Prevents the call of `terminate` in case of exception propagation, and helps ensure that destructors do everything they are supposed to do.

virtdestr Virtual destructors

This rule relates to Item 14 in "Effective C++".

Description Destructors of base classes must be declared `virtual`.

Justification Ensures that base and derived destructors are called before memory deallocation.

Chapter 6

Customizing Standard Rules and Rule Sets

Logiscope RuleChecker is an open-ended tool for which it is possible to customize standard rule checking or even write new personal rule checking scripts to better fit to your verification process.

This chapter presents how to customise Rule Sets and modify standard rules scripts to adapt them to specifics of user coding standards / verification requirements.

To develop a new rule script, please refer to *Kalimetrix Logiscope - Adding Ada, Java and C++ scriptable rules, metrics and contexts* advanced guide.

6.1 Modifying the Rule Set

A Rule Set is user-accessible textual file containing the specification of the programming rules to be checked by Logiscope *RuleChecker.* A Rule Set file extension is **".rst".**

Specifying one or more Rule Set files is mandatory when setting up a Logiscope *RuleChecker* project.

The Rule Sets allow to adapt *Logiscope RuleChecker* verification to a specific context taking into the applicable coding standard.

- Rule checking can be activated or de-activated.
- Some rules have parameters that allow to customize the verification. Changing the parameters changes the behaviour of the rule checking. See next section.
- The default name of a standard rule can be changed to match the name and/or identifier specified in the applicable coding standard.
  The same standard rule can even be used twice with different names and different parameters.
- The default severity level of a rule can be modified.
- A new set of severity levels with a specific ordering: e.g. "Mandatory", "Highly Recommended", "Recommended" can be specified.

All these actions can be done by editing the Logiscope Rule Set(s) and changing the corresponding specifications. For more information on how to use and modify rule sets

in Logiscope projects, please refer to:

- *Kalimetrix Logiscope RuleChecker & QualityChecker Getting Started*.
- *Kalimetrix Logiscope RuleChecker & QualityChecker Basic Concepts*.

The standard Rule Set files should be in the **RuleSets\C++** folder:

1. in the standard Logiscope Reference: i.e. the **Ref** folder of the Kalimetrix Logiscope installation directory ,
2. in one of the directories specified in the environment variable LOG_REF_ENV. The syntax of LOG_REF_ENV is dir1;dir2;…;dirn (directory names separated by semi-colons) on Windows and dir1:dir2:…:dirn (directory names separated by colons) on Unix and Linux.

To change the default behavior of a rule set, it is highly recommended to first make your own rule set, for example from a copy of default Rule Set files provided with Logiscope.

6.2 Customizing Standard Rule Scripts

The precise definition of these rules has been given in previous chapter.

ansi Function Declarations in ANSI Syntax

By default, the parameters **name** and **void** are both put:

```
STANDARD ansi ON LIST "name" "void" END LIST END STANDARD
```

To check that the parameters are named and their type indicated, just put the **name** parameter:

```
STANDARD ansi ON LIST "name" END LIST END STANDARD
```

To forbid the empty parameter lists, just put the **void** parameter:

```
STANDARD ansi ON LIST "void" END LIST END STANDARD
```

cmclass A Single Class per Code File

By default, the type of modules considered as code files is **CODE**, which corresponds to the suffixes ***.cc**, ***.cxx**, ***.cpp**, ***.C** or ***.c**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmclass ON LIST "CODE" END LIST END STANDARD
```

Change the definition of the **CODE** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.CC" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
```

```
END METRIC
STANDARD cmclass ON LIST "CODE" END LIST END STANDARD
```

Or choose to add a new module type (**MY_CODE**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "MY_CODE" "*.CC" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmclass ON LIST "MY_CODE" END LIST END STANDARD
```

cmdef Classes in Code File

By default, the type of modules considered as code files is **CODE**, which corresponds to the suffixes ***.cc**, ***.cxx**, ***.cpp**, ***.C** or ***.c**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmdef ON LIST "CODE" END LIST END STANDARD
```

Change the definition of the **CODE** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.CC" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmdef ON LIST "CODE" END LIST END STANDARD
```

Choose to add a new module type (**MY_CODE**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "MY_CODE" "*.CC" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmdef ON LIST "MY_CODE" END LIST END STANDARD
```

const Literal Constants

By default, the allowed literal constants are "", " ", "**0**" and "**1**":

```
STANDARD const ON LIST """""" """ """ "0" "1" END LIST END STANDARD
```

To allow the literal constant **MY_CST**, but forbid the constant **1**:

```
STANDARD const ON LIST """""" """ """ "0" "MY_CST" END LIST END STANDARD
```

constrcpy Copy Constructor

By default the "**dynalloc**" parameter is not put:

```
STANDARD constrcpy ON END STANDARD
```

To look for the copy constructor only if there is a class member which is a pointer:

```
STANDARD constrcpy ON "dynalloc" END STANDARD
```

convnewdel Adhere to Convention when Writing "new" and "delete" Operators

By default the "**static**" parameter is not put:
```
STANDARD convnewdel ON END STANDARD
```
To declare `static new` and `delete` operators :
```
STANDARD convnewdel ON "static" END STANDARD
```

dmaccess Access to Data Members

By default, only the data members in the public part of a class are forbidden:
```
STANDARD dmaccess ON LIST "public" END LIST END STANDARD
```
To forbid the data members in the public and protected part of a class:
```
STANDARD dmaccess ON LIST "public" "protected" END LIST END STANDARD
```

exprcplx Expressions Complexity

By default, the maximum authorized complexity level is 13:
```
STANDARD exprcplx ON MINMAX 0 13 END STANDARD
```
To change this value to 16, for example:
```
STANDARD exprcplx ON MINMAX 0 16 END STANDARD
```

exprparenth Parenthesis in Expressions

By default, the **partpar** parameter is put:
```
STANDARD exprparenth ON "partpar" END STANDARD
```
For a stricter rule, remove this parameter:
```
STANDARD exprparenth ON END STANDARD
```

funcres Reserved Functions

By default, no function names are forbidden:
```
STANDARD funcres ON LIST END LIST END STANDARD
```
To forbid the functions **system** and **malloc**, for example:
```
STANDARD funcres ON LIST "system" "malloc" END LIST END STANDARD
```

goto Goto Statement

By default, all `goto` statements are forbidden:
```
STANDARD goto ON LIST END LIST END STANDARD
```
To authorize the statements `goto ok;` and `goto error;`:
```
STANDARD goto ON LIST "ok" "error" END LIST END STANDARD
```

Headercom Module Header Comments

It is possible to define a format for the header comment depending on the type of the module as it is defined in metric **type**.

The format of the comment is defined as a list of regular expressions that shall be found

in the header comment in the order of declaration.

Formats are defined by regular expressions. The regular expression language is a subset of the one defined by the Posix 1003.2 standard (Copyright 1994, the Regents of the University of California).

A regular expression is comprised of one or more non-empty branches, separated by the "|" character.

A branch is one or more atomic expressions, concatenated.

Each atom can be followed by the following characters:

- * - the expression matches a sequence of 0 or more matches of the atom,
- + - the expression matches a sequence of 1 or more matches of the atom,
- ? - the expression matches a sequence of 0 or 1 match of the atom,
- {i} - the expression matches a sequence of i or more matches of the atom,
- {i,j} - the expression matches a sequence of i through j (inclusive) matches of the atom.

An atomic expression can be either a regular expression enclosed in "()", or:

- [...] - a brace expression, that matches any single character from the list enclosed in "[]",
- [^...] - a brace expression that matches any single character not from the rest of the list enclosed in "[]",
- . - it matches any single character,
- ^ - it indicates the beginning of a string (alone it matches the null string at the beginning of a line),
- $ - it indicates the end of a string (alone it matches the null string at the end of a line).

For more details, please refer to the related documentation.

Example:

```
".+_Ptr" matches strings like "abc_Ptr", "hh_Ptr", but not "_Ptr",
"T[a-z]*" matches strings like "Ta", "Tb", "Tz",
"[A-Z][a-z0-9_]*" matches strings like "B1", "Z0", "Pp", "P_1_a".
```

By default, a header comment with the name of the file, its author, its date and possible remarks is required for files of the **HEADER** and **CODE** type (for the signification of these types, see in Paragraph , *cmclass A Single Class per Code File*):

```
STANDARD Headercom ON
LIST "HEADER"          "Name: [a-z]*" "Author: [A-Z][a-z]*"
                       "Date: [0-9][0-9]/[0-9][0-9]/[0-9][0-9]"
                       "Remarks:" END LIST
LIST "CODE"            "Name: [a-z]*" "Author: [A-Z][a-z]*"
                       "Date: [0-9][0-9]/[0-9][0-9]/[0-9][0-9]"
                       "Remarks:" END LIST
END STANDARD
```

Example of required header:

```
////////////////////////////////////////////
// Name: program
// Author: Andrieu
// Date: 08/07/96
// Remarks: example of comments
////////////////////////////////////////////
```

headercom Function and Class Header Comments

It is possible to define a format for the comment preceding a function or a class, depending on the type of the function definition or declaration, or class definition (**func_glob_def**, **func_glob_decl**, **func_stat_def**, **func_stat_decl, class**).

The format of the comment is defined as a list of regular expressions (see in Paragraph , *Headercom Module Header Comments*) that shall be found in the comment in the order of declaration.

By default, only a comment beginning with "/*" is required for functions or classes:

```
STANDARD headercom ON
LIST "class"            "///*" END LIST
LIST "func_glob_def"    "///*" END LIST
LIST "func_glob_decl"   "///*" END LIST
LIST "func_stat_def"    "///*" END LIST
LIST "func_stat_decl"   "///*" END LIST
END STANDARD
```

Here is another example, with different required comments depending on the item type:

```
STANDARD headercom ON
LIST "class"            "Name of the class:"
                        "Filename:"
END LIST
LIST "func_glob_def"    "Definition of the extern function:"
                        "Author: [A-Z][a-z]*"
END LIST
LIST "func_glob_decl"   "Declaration of the extern funciton:"
                        "Date: [0-9][0-9]/[0-9][0-9]/[0-9][0-9]"
END LIST
LIST "func_stat_def"    "Definition of the static function:"
                        "Remarks:"
END LIST
LIST "func_stat_decl"   "Declaration of the static function:"
                        "Purpose:"
END LIST
END STANDARD
```

hmclass A Single Class Definition per Header File

By default, the type of modules considered as header files is **HEADER**, which corresponds to the filter ***.{h,hh,H,hxx,hpp}**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmclass ON LIST "HEADER" END LIST END STANDARD
```

Change the definition of the **HEADER** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmclass ON LIST "HEADER" END LIST END STANDARD
```

Or choose to add a new module type (**MY_HEADER**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "MY_HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmclass ON LIST "MY_HEADER" END LIST END STANDARD
```

hmdef Header File Contents

By default, the type of modules considered as header files is **HEADER**, which corresponds to the filter ***.{h,hh,H,hxx,hpp}**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmdef ON LIST "HEADER" END LIST END STANDARD
```

Change the definition of the **HEADER** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmdef ON LIST "HEADER" END LIST END STANDARD
```

Or choose to add a new module type (**MY_HEADER**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "MY_HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmdef ON LIST "MY_HEADER" END LIST END STANDARD
```

hmstruct Header File Structure

By default, the MIN value is 1 and the MAX value is 999 and the type of modules considered as header files is **HEADER**, which corresponds to the filter ***.{h,hh,H,hxx,hpp}**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmstruct ON MINMAX 1 999 LIST "HEADER" END LIST END STANDARD
```

Change the definition of the **HEADER** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmstruct ON MINMAX 1 999 LIST "HEADER" END LIST END STANDARD
```

Or choose to add a new module type (**MY_HEADER**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h,hh,H,hxx}" END LIST
LIST "MY_HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmstruct ON MINMAX 1 999 LIST "MY_HEADER" END LIST END STANDARD
```

Change the MINMAX values:

```
STANDARD hmstruct ON MINMAX 4 9 LIST "HEADER" END LIST END STANDARD
```

identfmt Identifier Format

It is possible to define a format for each of the categories listed below:

NAME	DESCRIPTION	DEFAULT
type	type name	any
type_obj	object type name	type, any
type_array	array type name	type, any
type_array_obj	object array type name	type_array, type_obj, type, any
type_ptr	pointer type name	type, any
type_ptr_obj	object pointer type name	type_obj, type_ptr, type, any
type_ref	reference type	type_ptr, type, any
type_ref_obj	object reference type	type_obj, type_ref, type_ptr, type, any
variable	variable name	any
variable_obj	object variable name	variable, any
variable_array	array variable name	variable, any
variable_array_obj	object array variable name	variable_obj, variable_array, variable, any
variable_ptr	pointer variable name	variable, any
variable_ptr_obj	object pointer variable name	variable_obj, variable_ptr, variable, any
variable_ref	reference variable name	variable_ptr, variable, any
variable_ref_obj	object reference variable name	variable_obj, variable_ref, variable_ptr, variable, any
type_func	function type name	function, type, any

type_struct	structured type name	type, any
type_struct_item	structure item name	variable, any
type_struct_item_obj	object structure item name	type_struct_item, variable_obj, variable, any
type_struct_item_array	array structure item name	type_struct_item, variable_array, variable, any
type_struct_item_array_obj	object array structure item name	type_struct_item_obj, type_struct_item_array, type_struct_item, variable_array, variable_obj, variable, any
type_struct_item_ptr	pointer structure item name	type_struct_item, variable_ptr, variable, any
type_struct_item_ptr_obj	object pointer structure item name	type_struct_item_obj, type_struct_item, variable_ptr, variable_obj, variable, any
type_struct_item_ref	reference structure item name	type_struct_item_ptr, type_struct_item, variable_ptr, variable_ref, variable, any
type_struct_item_ref_obj	object reference structure item name	type_struct_item_ptr_obj, type_struct_item_obj, type_struct_item_ref, type_struct_item_ptr, type_struct_item, variable_obj, variable_ptr, variable_ref, variable, any
type_union	union type name	type, any
type_union_item	union item name	variable, any
type_union_item_obj	object union item name	type_union_item, variable_obj, variable, any
type_union_item_array	array union item name	type_union_item, variable_array, variable, any
type_union_item_array_obj	object array union item name	type_union_item_obj, type_union_item_array, type_union_item, variable_obj, variable_array, variable, any
type_union_item_ptr	pointer union item name	type_union_item, variable_ptr, variable, any

type_union_item_ptr_obj	object pointer union item name	type_union_item_obj, type_union_item, variable_obj, variable_ptr, variable, any
type_union_item_ref	reference union item name	type_union_item_ptr, type_union_item, variable_ref, variable_ptr, variable, any
type_union_item_ref_obj	object reference pointer union item name	type_union_item_ptr_obj, type_union_item_obj, type_union_item_ref, type_union_item_ptr, type_union_item, variable_obj, variable_ref,variable_ptr, variable, any
enum	enumerated type name	type, any
const_enum_item	enumerated type item name	const, any
class	class name	type, any
class_attr	class attribute name	variable, any
class_attr_obj	class object attribute name	class_attr, variable_obj, variable, any
class_attr_array	class array attribute name	class_attr, variable_array, variable, any
class_attr_array_obj	class object array attribute name	class_attr_obj, class_attr, variable_obj, variable_array, variable, any
class_attr_ptr	class pointer attribute name	variable_ptr, class_attr, variable, any
class_attr_ptr_obj	class object pointer attribute name	class_attr_obj, class_attr_ptr, class_attr, variable_obj, variable_ptr, variable, any
class_attr_ref	class reference attribute name	class_attr_ptr, class_attr, variable_ref, variable_ptr, variable, any
class_attr_ref_obj	class object reference attribute name	class_attr_ptr_obj, class_attr_obj, class_attr_ref, class_attr_ptr, class_attr, variable_obj, variable_ref, variable_ptr, variable, any
method	class method name	function, class_attr, any
namespace	name space name	any

function	function name	any
const	constant name	any
const_obj	constant object name	const, any
const_array	constant array name	const, any
const_array_obj	constant object array name	const_obj, const, any
const_ptr	constant pointer name	const, any
const_ptr_obj	constant object pointer name	const_obj, const, any
const_ref	constant reference name	const_ptr, const, any
const_ref_obj	constant object reference name	const_ptr_obj, const_obj, const_ref, const_ptr, const, any
var_stat	static variable name	variable, any
var_stat_obj	static object variable name	variable_obj, var_stat, variable, any
var_stat_array	static array variable name	variable_array, var_stat, variable, any
var_stat_array_obj	static object array variable name	variable_obj, variable_array, var_stat, variable, any
var_stat_ptr	static pointer variable name	var_stat, variable_ptr, variable, any
var_stat_ptr_obj	static object pointer variable name	var_stat_obj, var_stat_ptr, var_stat, variable_obj, variable_ptr, variable, any
var_stat_ref	static reference variable name	var_stat_ptr, var_stat, variable_ref, variable_ptr, variable, any
var_stat_ref_obj	static object reference variable name	var_stat_ptr_obj, var_stat_obj, var_stat_ref, var_stat_ptr, var_stat, variable_obj, variable_ref, variable_ptr, variable, any
var_glob	global variable name	variable, any
var_glob_obj	global object variable name	variable_obj, var_glob, variable, any
var_glob_array	global array variable name	variable_array, var_glob, variable, any
var_glob_array_obj	global object array variable name	variable_obj, variable_array, var_glob, variable, any
var_glob_ptr	global pointer variable name	var_glob, variable_ptr, variable, an

var_glob_ptr_obj	global object pointer variable name	var_glob_obj, var_glob_ptr, var_glob, variable_obj, variable_ptr, variable, any
var_glob_ref	global reference variable name	var_glob_ptr, var_glob, variable_ref, variable_ptr, variable, any
var_glob_ref_obj	global object reference variable name	var_glob_ptr_obj, var_glob_obj, var_glob_ref, var_glob_ptr, var_glob, variable_obj, variable_ref, variable_ptr, variable, any
var_auto	automatic variable name	variable, any
var_auto_obj	automatic object variable name	var_auto, variable_obj, variable, any
var_auto_array	automatic array variable name	var_auto, variable_array, variable, any
var_auto_array_obj	automatic object array variable name	var_auto, variable_obj, variable_array, variable, any
var_auto_ptr	automatic pointer variable name	var_auto, variable_ptr, variable, any
var_auto_ptr_obj	automatic object pointer variable name	var_auto_obj, var_auto_ptr, var_auto, variable_obj, variable_ptr, variable, any
var_auto_ref	automatic reference variable name	var_auto_ptr, var_auto, variable_ref, variable_ptr, variable, any
var_auto_ref_obj	automatic object reference variable name	var_auto_ptr_obj, var_auto_obj, var_auto_ref, var_auto_ptr, var_auto, variable_obj, variable_ref, variable_ptr, variable, any
macro	macro name	any
macro_func	function macro name	macro, function, any
macro_const	macro constant name	macro, const, any
macro_flag	macro flag name	macro, any
parameter	parameter name	variable, any
parameter_obj	object parameter name	parameter, variable_obj, variable, any

parameter_array	array parameter name	parameter, variable_obj, variable_array, variable, any
parameter_array_obj	object array parameter name	parameter_obj, parameter_array, parameter, variable_array_obj, variable_obj, variable_array, variable, any
parameter_ptr	pointer parameter name	parameter, variable_ptr, variable, any
parameter_ptr_obj	object pointer parameter name	parameter_obj, parameter_ptr, parameter, variable_ptr_obj, variable_obj, variable_ptr, variable, any
parameter_ref	reference parameter name	parameter_ptr, parameter, variable_ref, variable_ptr, variable, any
parameter_ref_obj	object reference parameter name	parameter_ptr_obj, parameter_obj, parameter_ref, parameter_ptr, parameter, variable_obj, variable_ref, variable_ptr, variable, any

The third column represents inherited categories: for instance, for no distinction between the **macro-func**, the **macro-const** and the **macro-flag** categories, just define a particular format for the **macro** categories, which is inherited by the previous ones.

A special keyword **any** is used to define the default value for all identifier categories not explicitly defined.

The format of the identifier is defined by a regular expression (see in Paragraph , *Headercom Module Header Comments*).

By default, the only restrictions concern the constants and the macros, which must have no lower case letter:

```
STANDARD identfmt ON
LIST "any"                          ".*"
     "type"                         ".*"
     "type_obj"                     ".*"
     "type_array"                   ".*"
     "type_array_obj"               ".*"
     "type_ptr"                     ".*"
     "type_ptr_obj"                 ".*"
     "type_ref"                     ".*"
     "type_ref_obj"                 ".*"
     "variable"                     ".*"
     "variable_obj"                 ".*"
     "variable_array"               ".*"
     "variable_array_obj"           ".*"
     "variable_ptr"                 ".*"
     "variable_ptr_obj"             ".*"
```

```
"variable_ref"                  ".*"
"variable_ref_obj"              ".*"
"type_func"                     ".*"
"type_struct"                   ".*"
"type_struct_item"              ".*"
"type_struct_item_obj"          ".*"
"type_struct_item_array"        ".*"
"type_struct_item_array_obj"    ".*"
"type_struct_item_ptr"          ".*"
"type_struct_item_ptr_obj"      ".*"
"type_struct_item_ref"          ".*"
"type_struct_item_ref_obj"      ".*"
"type_union"                    ".*"
"type_union_item"               ".*"
"type_union_item_obj"           ".*"
"type_union_item_array"         ".*"
"type_union_item_array_obj"     ".*"
"type_union_item_ptr"           ".*"
"type_union_item_ptr_obj"       ".*"
"type_union_item_ref"           ".*"
"type_union_item_ref_obj"       ".*"
"enum"                          ".*"
"const_enum_item"               ".*"
"class"                         ".*"
"class_attr"                    ".*"
"class_attr_obj"                ".*"
"class_attr_array"              ".*"
"class_attr_array_obj"          ".*"
"class_attr_ptr"                ".*"
"class_attr_ptr_obj"            ".*"
"class_attr_ref"                ".*"
"class_attr_ref_obj"            ".*"
"method"                        ".*"
"namespace"                     ".*"
"function"                      ".*"
"const"                         "[A-Z0-9_]*"
"const_obj"                     "[A-Z0-9_]*"
"const_array"                   "[A-Z0-9_]*"
"const_array_obj"               "[A-Z0-9_]*"
"const_ptr"                     "[A-Z0-9_]*"
"const_ptr_obj"                 "[A-Z0-9_]*"
"const_ref"                     "[A-Z0-9_]*"
"const_ref_obj"                 "[A-Z0-9_]*"
"var_stat"                      ".*"
"var_stat_obj"                  ".*"
"var_stat_array"                ".*"
"var_stat_array_obj"            ".*"
"var_stat_ptr"                  ".*"
"var_stat_ptr_obj"              ".*"
"var_stat_ref"                  ".*"
"var_stat_ref_obj"              ".*"
"var_glob"                      ".*"
"var_glob_obj"                  ".*"
"var_glob_array"                ".*"
"var_glob_array_obj"            ".*"
"var_glob_ptr"                  ".*"
"var_glob_ptr_obj"              ".*"
"var_glob_ref"                  ".*"
"var_glob_ref_obj"              ".*"
"var_auto"                      ".*"
"var_auto_obj"                  ".*"
"var_auto_array"                ".*"
"var_auto_array_obj"            ".*"
"var_auto_ptr"                  ".*"
"var_auto_ptr_obj"              ".*"
"var_auto_ref"                  ".*"
```

```
     "var_auto_ref_obj"              ".*"
     "macro"                         "[^a-z]*"
     "macro_const"                   "[^a-z]*"
     "macro_flag"                    "[^a-z]*"
     "macro_func"                    "[^a-z]*"
     "parameter"                     ".*"
     "parameter_obj"                 ".*"
     "parameter_array"               ".*"
     "parameter_array_obj"           ".*"
     "parameter_ptr"                 ".*"
     "parameter_ptr_obj"             ".*"
     "parameter_ref"                 ".*"
     "parameter_ref_obj"             ".*"
END LIST END STANDARD
```

For the class attributes to begin with "m_", the class pointer attributes to begin with "m_p", the constants and the macros to have no lower case letter and no underscore at the beginning and the end, the global variables to begin with "g_", the global pointer variables to begin with "g_p" and all other identifiers not to begin or end with an underscore:

```
STANDARD identfmt ON
LIST "any"                   "[^_](.*[^_])?$"
     "class_attr"            "m_.*[^_]$"
     "class_attr_ptr"        "m_p.*[^_]$"
     "const"                 "[A-Z0-9]([A-Z0-9_]*[A-Z0-9])?$"
     "var_glob"              "g_.*[^_]$"
     "var_glob_ptr"          "g_p.*[^_]$"
     "macro"                 "[A-Z0-9]([A-Z0-9_]*[A-Z0-9])?$"
END LIST END STANDARD
```

identl Identifier Length

The possible categories of identifiers are the same as for the **identfmt** rule (see in Paragraph , *identfmt Identifier Format*).

By default, the methods and functions must have between 4 and 25 characters, the types, variables, constants, macros and classes between 5 and 25, and the other identifiers between 1 and 25:

```
STANDARD identl ON
LIST "any"                  MINMAX 1 25
     "type"                 MINMAX 5 25
     "type_ptr"             MINMAX 5 25
     "variable"             MINMAX 5 25
     "variable_ptr"         MINMAX 5 25
     "type_func"            MINMAX 5 25
     "type_struct"          MINMAX 5 25
     "type_struct_item"     MINMAX 5 25
     "type_union"           MINMAX 5 25
     "type_union_item"      MINMAX 5 25
     "enum"                 MINMAX 5 25
     "const_enum_item"      MINMAX 5 25
     "class"                MINMAX 5 25
     "class_attr"           MINMAX 5 25
     "class_attr_ptr"       MINMAX 5 25
     "method"               MINMAX 4 25
     "namespace"            MINMAX 5 25
     "function"             MINMAX 4 25
     "const"                MINMAX 5 25
     "const_ptr"            MINMAX 5 25
     "var_stat"             MINMAX 1 25
     "var_stat_ptr"         MINMAX 1 25
     "var_glob"             MINMAX 5 25
```

```
     "var_glob_ptr"       MINMAX 5 25
     "var_auto"           MINMAX 1 25
     "var_auto_ptr"       MINMAX 1 25
     "macro"              MINMAX 5 25
END LIST END STANDARD
```

identres Reserved Identifiers

By default, there are no reserved identifiers:

```
STANDARD identres ON LIST END LIST END STANDARD
```

To forbid the identifiers "true" and "false":

```
STANDARD identres ON LIST "true" "false" END LIST END STANDARD
```

incltype Included Modules Type

By default, **HEADER** modules can be included in **HEADER** and **CODE** modules:

```
STANDARD incltype ON
LIST "HEADER"          "HEADER" END LIST
LIST "CODE"            "HEADER" END LIST
END STANDARD
```

To also allow **CODE** modules to be included in **CODE** modules:

```
STANDARD incltype ON
LIST "HEADER"          "HEADER" END LIST
LIST "CODE"            "HEADER" "CODE" END LIST
END STANDARD
```

For the signification of the **CODE**, **HEADER**, ... types, see in Paragraph , *cmclass A Single Class per Code File*).

inldef Inline Functions Declaration and Definition

By default, the "private " parameter is not active:

```
STANDARD inldef ON END STANDARD
```

To indicate that private inline functions must be defined in the class definition file (.cpp file) and other inline functions in the class declaration file (.h file):

```
STANDARD inldef ON "private" END STANDARD
```

macrocharset Characters Used in Macros

By default, no characters are forbidden in macros:

```
STANDARD macrocharset ON LIST "constant" "" "function" "" END LIST END
  STANDARD
```

To forbid the characters @#!&/[]{}~‘’ in macro-constants and #@%.\ in macro-functions:

```
STANDARD macrocharset ON LIST "constant" "@#!&/[]{}~‘’" "function"
  "#@%.\" END LIST END STANDARD
```

mconst Macro Constant Usage

By default, the **var** option is selected:

```
STANDARD mconst ON "var" END STANDARD
```

To have the **const** option instead:

```
STANDARD mconst ON "const" END STANDARD
```

To have the **nodefine** option instead:

```
STANDARD mconst ON "nodefine" END STANDARD
```

mname File Names

By default, the part of the class name taken into account is between the characters 1 and 5:

```
STANDARD mname ON MINMAX 1 5 END STANDARD
```

To have instead the characters 4 and 10:

```
STANDARD mname ON MINMAX 4 10 END STANDARD
```

nopreproc No Pre-processing Instructions

By default, only #line and # alone may not be used:

```
STANDARD nopreproc ON LIST "define" "include" "if" "pragma" "undef"
"error" END LIST END STANDARD
```

To allow only #define, #line and # alone:

```
STANDARD nopreproc ON LIST "define" "line" "none" END LIST END STANDARD
```

nostruct Keyword Struct Not Allowed

By default, C-style structs are forbidden:

```
STANDARD nostruct ON END STANDARD
```

To allow C-style structs and then forbid C++-style structs (such as access specifiers: private for example, or methods):

```
STANDARD nostruct ON "cstruct" END STANDARD
```

operass Assignment Operator

By default the "**dynalloc**" parameter is not put:

```
STANDARD operass ON END STANDARD
```

To look for the assignment operator only if there is a class member which is a pointer:

```
STANDARD operass ON "dynalloc" END STANDARD
```

parammode Parameters Mode

By default, the three modes "**IN**", "**OUT**" and "**INOUT**" are authorized:

```
STANDARD parammode ON LIST "OUT" "INOUT" "IN" END LIST END STANDARD
```

To authorize only the mode "**IN**":

```
STANDARD parammode ON LIST "IN" END LIST END STANDARD
```

sectord "public", "private" and "protected" Sections Order

By default no particular order is given:

```
STANDARD sectord ON LIST END LIST END STANDARD
```

To authorize the first section to be without any specifier, and then the specifiers to be in the order private, protected and public:

```
STANDARD sectord ON LIST "" "private" "protected" "public" END LIST END
  STANDARD
```

sgancstr Single Ancestor

By default no ancestor is specified:

```
STANDARD sgancstr ON END STANDARD
```

To indicate a particular ancestor, name it:

```
STANDARD sgancstr ON "father" END STANDARD
```

swdef "default" within "switch"

By default, the `default` case has to be the last one:

```
STANDARD swdef ON "last" END STANDARD
```

To have only a `default` case, whatever its position:

```
STANDARD swdef ON END STANDARD
```

swend End of Cases in a "switch"

By default, an instruction `break`, `continue`, `goto`, `return` or `exit` is not mandatory for the last `switch` of a `case`:

```
STANDARD swend ON "nolast" END STANDARD
```

To impose such an instruction at the end of all the cases of a `switch`:

```
STANDARD swend ON END STANDARD
```

typeres Reserved Types

By default, there are no reserved types:

```
STANDARD typeres ON
LIST END LIST
LIST END LIST END STANDARD
```

To forbid the types `int`, `char` and `register  double` for variables and the types `unsigned int` and `double` for functions:

```
STANDARD typeres ON
LIST "data" "int" "char" "register double" END LIST
LIST "function" "unsigned int" "double" END LIST
END STANDARD
```

varstruct Struct and Union Variables

By default, the **nostruct** option is not selected:

```
STANDARD varstruct ON END STANDARD
```

To have the **nostruct** option:

```
STANDARD varstruct ON "nostruct" END STANDARD
```

6.3 Renaming Rules

It is possible to rename standard rules to have as many versions of them as needed. The renamed rules have their own set of parameters, and their own definition. Creating rules in this way allows to have multiple versions of the same rule using different parameters. It also enables adapting the names of the rules that are provided to your naming standard and their definitions to the description you are used to seeing.

The rule used to create a new one can be a built-in rule, a user rule or even an already renamed rule.

Rule Script Format

A rule verification script containing a renamed rule description should be created. It should be named *rule_name*.std, where *rule_name* is the name of the rule being created. The contents of the file should follow the following format:

```
.NAME <long_name>
.DESCRIPTION <user_description>
.COMMAND rename <mnemonic_of_the_renamed_rule>
```

where:

<long_name> is free text, that can include spaces. It is a more detailed title of the rule. It will appear as an explanation of the rule name in Logiscope.

<user_description> is the description of the rule, that will be available in Logiscope.

<mnemonic_of_the_renamed_rule> is the name of the standard rule that the new rule is based upon.

Example of a renamed rule (rename of the goto rule):

```
.NAME No goto at all
.DESCRIPTION
In our standard the goto statement is absolutely forbidden.
.COMMAND rename goto
```

Rule Script Location

The rule script should be placed in one of the following places in the **Rules\Ada** folder :

1. in the standard Logiscope Reference: i.e. the **Ref** folder of the Kalimetrix Logiscope installation directory;
2. in one of the directories in the environment variable LOG_REF_ENV.
   The syntax of LOG_REF_ENV is dir1;dir2;…;dirn (directory names separated by semi-colons) on Windows and dir1:dir2:…:dirn (directory names separated by

colons) on Unix and Linux.

Rule Verification Activation

The new rule must be added into the Rule Set file (**.rst**) using the following syntax:

```
STANDARD new_std RENAMING old_std ON parameters END STANDARD
```

where

`new_std` is the name of the rule being created.

`old_std` is the name of the existing rule.

parameters (optional) is the list of parameters, as for any other Logiscope rule.

Example:

```
STANDARD mygoto RENAMING goto ON LIST "test" END LIST END STANDARD
```

6.4 Creating New Rule Scripts

New rule verification scripts can also be created entirely using Tcl scripts.

More about this can be found in the dedicated *Kalimetrix Logiscope - Adding Ada, Java and C++ scriptable rules, metrics and contexts* advanced guide.

Notices

The licensed program described in this document and all licensed material available for it are provided by Kalimetrix under terms of the Kalimetrix Customer Agreement, Kalimetrix International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-Kalimetrix products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Kalimetrix has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Kalimetrix products. Questions on the capabilities of non-Kalimetrix products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Copyright license

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to Kalimetrix, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. Kalimetrix, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
© (your company name) (year). Portions of this code are derived from Kalimetrix Corp. Sample Programs. © Copyright Kalimetrix Corp. _enter the year or years_.

Trademarks

Kalimetrix, the Kalimetrix logo, Kalimetrix.com are trademarks or registered trademarks of Kalimetrix, registered in many jurisdictions worldwide. Other product and services names might be trademarks of Kalimetrix or other companies.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, FrameMaker, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.
AIX and Informix are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
HP and HP-UX are registered trademarks of Hewlett-Packard Corporation.
Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
Macrovision and FLEXnet are registered trademarks or trademarks of Macrovision Corporation.

Microsoft, Windows, Windows 2003, Windows XP, Windows Vista and/or other Microsoft products referenced herein are either trademarks or registered trademarks of Microsoft Corporation.

Netscape and Netscape Enterprise Server are registered trademarks of Netscape Communications Corporation in the United States and other countries.
Sun, Sun Microsystems, Solaris, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.
Pentium is a trademark of Intel Corporation.
ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S Patent and Trademark Office.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Other company, product or service names may be trademarks or service marks of others.