



RuleChecker & QualityChecker Ada Reference Manual

Before using this information, be sure to read the general information under “Notices” section, on page 51.

© Copyright Kalimetrix 2014

About This Manual

Audience

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

Related Documents

Reading first the following manuals 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 a dedicated document:

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

Overview

Project Settings

Chapter 1 presents the main settings of the Logiscope *RuleChecker* & *QualityChecker* Ada projects.

Command Line Mode

Chapter 2 specifies how to run Logiscope *RuleChecker* & *QualityChecker* Ada using the Logiscope command line interface.

Standard Metrics

Chapter 3 specifies the source codes metrics computed by Logiscope *QualityChecker* Ada.

Programming Rules

Chapter 4 specifies the programming rules checked by Logiscope *RuleChecker Ada*.

Customizing Standard Rules and Rule Sets

Chapter 5 describes the way to modify standard predefined rules and to create new ones with Logiscope *RuleChecker Ada*.

Conventions

The following typographical conventions are used:

bold	literals such as tool names (Studio) and file extension (*.ada),
<i>bold italics</i>	literals such as type names (<i>integer</i>),
<i>italics</i>	names that are user-defined such as directory names (<i>log_installation_dir</i>), 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.

Prerequisites

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 <i>critical</i> 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 <i>significant</i> business impact. The program is usable, but it is severely limited
Major	The problem has a <i>some</i> business impact. The program is usable, but less significant features (not critical to operation) are unavailable.
Minor	The problem has a <i>minimal</i> 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	Project Settings	
1.1	Project Specification.....	1
1.2	Source file Specification	2
1.3	QualityChecker Settings.....	2
1.4	RuleChecker Settings	3
1.5	Output Data	3
Chapter 2	Command Line Mode	
2.1	Logiscope create.....	5
2.1.1	Command Line Mode	5
2.1.2	Makefile mode	6
2.1.3	Options.....	7
2.2	Logiscope batch.....	9
2.2.1	Options.....	9
2.2.2	<i>Examples of Use</i>	10
Chapter 3	Standard Metrics	
3.1	.Function Scope	12
3.1.1	Line Counting	12
3.1.2	Data Flow	13
3.1.3	Halstead Metrics	14
3.1.4	Structured Programming	19
3.1.5	Control Flow	20
3.1.6	Relative Call Graph	21
3.2	Module Scope	23
3.2.1	Line Counting	23
3.2.2	Lexical and Syntactic Items	23
3.2.3	Halstead Metrics	24
3.2.4	Interface	25
3.3	Application Scope	27
3.3.1	Line Counting	27
3.3.2	Application Aggregates	27
3.3.3	Application Call Graph	28

Chapter 4 **Programming Rules**

Chapter 5 **Customizing Standard Rules and Rule Sets**

5.1	Modifying the Rule Set	41
5.2	Customizing Standard Rule Checking.....	42
5.3	Renaming Rules.....	48
5.4	Creating New Rule Scripts	49

Chapter 6 **Notices**

viii Kalimetrix Logiscope RuleChecker & QualityChecker Ada Reference Manual November
2014

Chapter 1

Project Settings

This chapter details specifics of the Logiscope Ada projects.

Logiscope Ada projects (“**.ttp**”) can be created using:

- **Logiscope Studio:** a graphical interface requiring a user interaction; refer to *IBM Rational Logiscope - RuleChecker & QualityChecker - Getting Started* to learn how to create a Logiscope project using Logiscope **Studio**,
- **Logiscope create:** a tool to be used from a standalone command line or within makefiles; refer to Chapter 2 to learn how to create a Logiscope project using Logiscope **create**.

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

1.1 Project Specification

Project Name

The project name is used to create the Logiscope project file containing the specification of a Logiscope project: e.g. list of source code files, parsing options, quality model, rule sets.

The “**.ttp**” extension will be added to the user-specified project name to name the Logiscope project file.

Location

The user shall specify the directory where the Logiscope project file and the Logiscope Repository will be created.

1.2 Source file Specification

Logiscope *RuleChecker* & *QualityChecker* projects must be given all the source files to analyze when creating a project.

Please note that the Logiscope application to be analyzed should be all or part of a complete project, able to be compiled and linked. The source code should be compliant with one of the Logiscope supported Ada dialects. Respecting this prerequisite will avoid problems like for instance multiply defined functions, which are poorly handled by Logiscope.

Source files to be analysed are specified using:

Source files root directory: the single directory gathering all the source files of the application.

Directories: to select the list of directories covering the application sources:

Include all subdirectories means that selected files will be searched for in every sub-directory of the application 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 directories list that includes application files through a new page.

Extensions: to specify the extensions of the Ada source files needed in the above selected directories. The extensions shall be separated with a semi-colon.

1.3 QualityChecker Settings

Logiscope *QualityChecker* allows evaluation of a software quality according to factors and criteria. The Quality Model file specifies:

- the metrics (i.e. static measurements, i.e. obtained without executing a program) to be used for assessing source code characteristics (e.g; maintainability, portability),
- the thresholds associated to each metric,
- the association between metrics and software characteristics to be assessed,
- the rating principles of the components defined in the source code files (e.g. functions, modules, application),

applicable to the project under analysis.

It is highly recommended to adapt the default / example Quality Model files provided in the standard Logiscope installation.

For more information, see *IBM Rational Logiscope - Basic Concepts Manual*.

1.4 RuleChecker Settings

Logiscope *RuleChecker* allows to automatically check a set of programming rules / coding standards which are gathered within a Rule Set file. This file is used to indicate which rules will be checked and to give parameters to customizable rules (see Chapter Customizing Standard Metrics and Rules).

1.5 Output Data

Logiscope Repository

Logiscope Ada *RuleChecker* & *QualityChecker* stores all data generated during source code parsing in a specific directory. This user-specified directory is called the Logiscope Repository.

The source files for a given Ada project are parsed one at a time. For each source file, the Logiscope parser produces Logiscope internal ASCII format files containing all necessary information extracted from the source code files among which:

- a file named **standards.chk** containing all the violations found in the source code files 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.

Chapter 2

Command Line Mode

2.1 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) .

2.1.1 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/Ada/OneArmedBandit/onearmedbandit.adb  
/users/logiscope/samples/Ada/OneArmedBandit/onearmedbandit.ads  
/users/logiscope/samples/Ada/OneArmedBandit/slotmachine.adb
```

Using the command line:

```
create aProject.ttp -audit -rule -lang ada -list filelist.lst
```

will create a new Logiscope Ada project file `aProject.ttp` containing 3 files: `onearmedbandit.adb`, `onearmedbandit.ads` and `slotmachine.adb` on which the *RuleChecker* and *QualityChecker* verification modules will be activated.

2.1.2 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.

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.*

2.1.3 Options

The **create** options are the following:

<code>create -lang ada</code>	
<code><ttp_file></code>	name of a Logiscope project to be created (with the “.ttp” extension). Path has to be absolute if the option -- is used.
<code>[-source <suffixes>]</code>	where <suffixes> is the list of accepted suffixes for the source files. Default is “*.ada;*.adb;*.ads”.
<code>[-root <directory>]</code>	where <directory> is the starting point of the source search. Default is the current directory. This option is exclusive with -list option.
<code>[-recurse]</code>	if present the source file search is done recursively in subfolders.
<code>[-list <list_file>]</code>	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.
<code>[-repository <directory>]</code>	where <directory> is the name of the directory where Logiscope internal files will be stored.
<code>[-no_compilation]</code>	avoid compiling the files if the -- option is used
<code>[-]</code>	when used in a makefile, introduces the compilation command with its arguments.
<code>[-audit]</code>	to select the <i>QualityChecker</i> verification module
<code>[-ref <Quality_model>]</code>	where <Quality_model> is the name of the Quality Model file (“.ref”) to add to the project. Default is <install_dir>/Ref/Logiscope.ref
<code>[-rule]</code>	to select the RuleChecker verification module
<code>[-rules <rules_file>]</code>	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/Ada/ will be used.
<code>[-relax]</code>	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.

2.2 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 **batch**, a Logiscope project shall have been created:

- using Logiscope **Studio**, refer refer to Section 1 or *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.

2.2.1 Options

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

<code>batch</code>	
<code><ttp_file></code>	name of a Logiscope project.
<code>[-tcl <tcl_file>]</code>	name of a Tcl script to be used to generate the reports instead of the default Tcl scripts.
<code>[-o <output_directory>]</code>	directory where the all reports are generated.
<code>[-external <violation_file>]*</code>	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 <i>RuleChecker</i> module for which the external violation importation mechanism is activated
<code>[-nobuild]</code>	generates reports without rebuilding the project. The project must have been built at least once previously.
<code>[-clean]</code>	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.
<code>[-addin export -format csv]</code>	generates the reports in csv format available using the file/export command.
<code>[-addin <addin> options]</code>	where <code><addin></code> is the name of the user <code>addin</code> to be activated and <code>options</code> the associated options generating specific reports.

<code>[-table]</code>	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.
<code>[-noframe]</code>	generates HTML reports with no left frame.
<code>[-v]</code>	displays the version of the batch tool.
<code>[-h]</code>	displays help and options for batch .
<code>[-err <log_err_folder>]</code>	where <code><log_err_folder></code> is a directory where troubleshooting files batch.err and batch.out should be put. By default, messages are directed to standard output and error.

2.2.2 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 `<log_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 3

Standard Metrics

Logiscope *QualityChecker* Ada 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 attributes (e.g. complexity, self-descriptiveness) or characteristics (e.g. Maintainability, Reliability) of the Ada functions, 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-descriptiveness 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* manual.

Source code metrics apply to different domains (control flow, data flow, calling relations, etc.) 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* Ada .

- The *Function scope*: the metrics are available for each function defined in the source files specified in the Logiscope project under analysis.
- The *Module scope*: the metrics are available for each Ada source file specified in the Logiscope project under analysis.
- The *Application scope*: the metrics are available for the set of Ada source files specified in the Logiscope project under analysis.

3.1 Function Scope

3.1.1 Line Counting

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

- *Kalimetrix Logiscope - Basic Concepts.*

lc_cline **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 comment lines in the function, including comments just before the function's header.

lc_csbra **Number of “brace” lines**

Definition Number of lines containing only a block tag (e.g. begin, end) in the module.

lc_stat **Number of statements**

Definition Number of executable statements in a function body.

Following statements are counted:

- Control statements: *abort*, block statement, *loop*, *goto*, *if*, labeled statement, named statement, *return*, *raise*, *case*, *exit*,
- Statements followed by “;”,

- Null statements,
- Pragmas.

3.1.2 Data Flow

dc_consts **Numbers of declared constants**

Definition Number of constants declared in constant and number declarations in a function.

dc_types **Number of declared types**

Definition Number of types and sub-types declarations in a function.

dc_vars **Number of declared variables**

Definition Number of variables declared in the variable declarations in a function.

dc_exc **Number of declared exceptions**

Definition Number of exceptions declared in the exception declarations in a function.

ic_param **Number of parameters**

Definition Number of formal parameters of a function.

3.1.3 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 parameterized 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 the function's "end",
- if the parameter "**in_body**" is provided, operators are only counted in the function's body (that is between the function's "is" and "end").

The following are operators:

- Declarations and types:

type declaration	(ex: type Int is range 1 .. 10;)
private type	(ex: type X is private ;))
private extension	(ex: type X is new Y with private ;))
subtype declaration	(ex: subtype Int is Integer;)
object declaration	(ex: I : Integer;)
aliased object	(ex: I : aliased Integer := 1;)
constant object	(ex: I : constant Integer := 1;)
aliased constant object	(ex: I : aliased constant Integer := 1;)
number declaration	(ex: PI : constant := 3.14;)
scalar type	(ex: type X is Integer range 1 .. 10;)
floating point definition	(ex: type X is digits 8;)
fixed point definition	(ex: type X is delta 0.125 range 0.0 .. 255.0;)
array type	(ex: type X is array (1 .. 10) of Y;)
index subtype definition	(ex: type X is array (Integer range <>) of Y)
unknown discriminant part	(ex: type X (<>);)
record type	(ex: type X is record ... end record ;))
null component list	(ex: type X is record null ; end record ;))
null record type	(ex: type X is null record ;))
tagged type	(ex: type X is abstract tagged null record;)
abstract type	(ex: type X is abstract new T;)
access to object type	(ex: type X is access A;)
access to subprogram type	(ex: type X is access procedure Proc;)

protected calling convention (ex: type X is access **protected** procedure Proc;)

access definition (ex: type T1 is record V : **access** T2; end record;

- Expressions:
 - Unary operators:

+ - unary plus or minus
not negation
() expression in parenthesis

- Binary Operators:

+ - * / mod rem ** arithmetic operators
& catenation operator
> < <= >= = /= comparison operators
and and then or or else xor logical operators

- Assignment operator: :=
- Other operators:

...(..)	type conversion	(ex: Integer(1.6))
()	subscripting	(ex: a[i])
...()	subprogram call	(ex: proc(1))
(..., ..., ...)	x parameter part	(ex: func(1,2,3))
	enumeration type definition	(ex: type X is (Y, Z);)
	index constraint	(ex: X : Y(1 .. 8, 1 .. 10);
	aggregate	(ex: (2 4 => 1, others => 0))
	index definition list	(ex: X : array (1 .. 3, 1 .. 5) of Y;
..., ..., ...	defining identifier list	(ex: A, B : Identifier;)
(...; ...; ...)	formal part	(ex: procedure P(A : in X; B : out Y))
	discriminant part	(ex: type T(A : X; B : Y) is ...;)
(..)	slice	(ex: Page(1 .. 4))
...	selected component	(ex: Car.Owner)
	compound name	(ex: Pack.Proc)
...'	attribute	(ex: Color'First)
(... with ...)	extension aggregate	(ex: Painted_Point'(Point with Red))
null	null access value	
others	others	
null record	null record aggregate	
... in ...	membership test	(ex: Today in Weekday)
...'(..)	qualified expression	(ex: A'(B))

- Statements:

IF	ELSE	ELSIF	LOOP
WHILE	FOR	CASE	WHEN
RETURN	GOTO label	<<LABEL>>	EXIT
DELAY	DELAY UNTIL	ABORT	RAISE
ACCEPT	TERMINATE	REQUEUE ... [WITH ABORT]	
... : ...	(named statement)		
[DECLARE ...] BEGIN ... END			(block statement)
... ' (...);	(code statement)		
NULL	(null statement)		

- Subprograms:

subprogram declaration	(ex: procedure Proc;)
abstract subprogram	(ex: procedure Proc is abstract ;))
subprogram body	(ex: procedure Proc is begin ... end Proc;)
stub	(ex: procedure Proc is separate ;))
subunit	(ex: separate (X) procedure Y is begin ... end Y;)
parameter mode	(ex: procedure Proc(P1 : in X; P2 : access Y);)

- Visibility rules:

use package clause	(ex: use Pack;)
use type clause	(ex: use type T;)
renaming declaration	(ex: package X renames Y;)

- Packages:

package declaration	(ex: package P is ... end P;)
package body	(ex: package body P is begin ... end P;)

- Tasks:

task declaration	(ex: task T is ... end T;)
task body	(ex: task body T is begin ... end T;)
protected type declaration	(ex: protected type T is ... end T;)
protected declaration	(ex: protected P is ... end P;)
protected body	(ex: protected body P is ... end P;)
entry declaration	(ex: entry E;)
entry body	(ex: entry E when ... is begin ... end E;)
entry index	(ex: for I in T range <>
select clause list	(ex: select ... or ... ab ... end select ;))
timed entry call	(ex: select ... or delay ... end select ;))

conditional entry call (ex: **select ... else ... end select;**)
asynchronous select (ex: **select ... then abort ... end select;**)

- Program structure:

private unit (ex: **private** procedure Proc;)
with clause (ex: **with** Pack;)
pragma (ex: **pragma** Page;)

- Exceptions:

exception declaration (ex: X : **exception**;)
exception handler (ex: **when** X => ...;)

- Generic units:

generic declaration (ex: **generic** procedure Proc;)
generic instantiation (ex: procedure X **is new** Proc;)
formal type (ex: **type** T **is digits** <>;)
formal subprogram (ex: **with procedure** Proc **is** <>;)
formal package (ex: **with package** Pack **is new** GP <>;)

- Representation issues:

at clause (ex: **for** X **use at** Y;)
attribute definition clause (ex: **for** X'Address **use** Y;)
record representation clause (ex: **for** T **use record** ... **end record**;)
component clause (ex: X **at** 1*Word range 0 .. 1;)

N1 Total number of operators

Also called `ha_topt`.

Definition Total number of operators used in a function.

n2 Number of distinct operands

Also called `ha_dopd`.

Definition Number of different operands used in a function.

This metric can be parametrized 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 the function's "end",
- if the parameter "**in_body**" is provided, operands are only counted in the function's body (that is between the function's "is" and "end").

The following are operands:

- Literals:
 - Integer literals (ex: 12, 0, 1E6, 123_456)
 - Real literals (ex: 12.0, 0.0, 0.456, 3.14159_26)
 - Based literals (ex: 2#1111_1111#, 16#F.FF#E+2)
 - Character literals (ex: 'A', '*', '"', ' ')
 - String literals (ex: "", "hello", "this is a ""string""")
- Identifiers (variable names, type names, function names, etc.)
- Operator names:

```
"and"  "or"   "xor"  "="   "/="   "<"   "<="  ">"   ">="
"+"    "-"   "&"   "*"   "/"    "*"   "mod" "rem" "***"
"abs"  "not"
```

N2 **Total number of operands**

Definition Total number of operands used in a function.
Alias ha_topd

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

3.1.4 Structured Programming

In structured programming:

- a function shall have a single entry point and a single exit point,
- each iterative or selective structures shall have a single exit point.

ct_bran Number of destructuring statements

Definition Number of destructuring statements in a function (*goto*, *exit* and *raise*).

ct_goto Number of gotos

Definition Number of *goto* statements in a function.

Alias GOTO

ct_exit **Number of exits**

Definition Number of explicit exit from a function (*return, terminate, raise* non récupéré dans la fonction).

Alias N_OUT

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.

3.1.5 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 statement in a function :
if, case, select

Alias N_STRUCTURE

ct_degree **Maximum degree**

Definition Maximum number of edges departing from a node.

ct_edge **Number of edges**

Definition Number of edges *e* 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_nest **Maximum nesting level**

Definition Maximum nesting level of control structures in a function.

ct_node **Number of nodes**

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

ct_vg **Cyclomatic number (VG)**

Definition Cyclomatic Number of the control graph of the function.

Alias VG, ct_cyclo

ct_npath **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.

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.

3.1.6 Relative Call Graph

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

- *Kalimetrix Logiscope - Basic Concepts.*

dc_calling **Number of callers**

Definition Number of functions calling the designated function.

Alias NBCALLING

IND_CALLS **Relative call graph call-paths**

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

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

3.2 Module Scope

3.2.1 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_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_sbrc Number of “brace” lines

Definition Number of lines containing only a block tag (e.g. begin, end) in the module.

3.2.2 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 a module.

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 file.

md_stat **Number of statements**

Definition Total number of executable statements in the method bodies defined in the file.

md_consts **Number of declared constants**

Definition Number of constants declared in the file.

md_exc **Number of declared exceptions**

Definition Total number of exceptions declared in the exception declaration in the 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.

3.2.3 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: $n = n_1 + n_2$
md_N	Halstead length
Definition	Halstead observed length of the module: $N = N_1 + N_2$
md_CN	Halstead estimated length
Definition	Halstead estimated length of the module. $\hat{N} = n_1 * \log_2(n_1) + n_2 * \log_2(n_2)$.
md_V	Halstead volume
Definition	Halstead Program Volume: $V = N * \log_2(n)$
md_L	Halstead level
Definition	Halstead Program Level: $L = (2 * n_2) / (n_1 * N_2)$
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: $E = V / L$

3.2.4 Interface

md_expc	Number of exported constants
Definition	Numbers of constants exported by the different compilation units of the module.

md_expex **Number of exported exceptions**

Definition Numbers of exceptions exported by the differents compilation units of the module.

md_expfn **Number of exported functions**

Definition Numbers of functions (packages, subprograms, tasks) exported by the differents compilation units of the module.

md_expty **Number of exported types**

Definition Numbers of types exported by the differents compilation units of the module.

md_expva **Number of exported variables**

Definition Numbers of variables exported by the differents compilation units of the module.

md_with **Number of WITH clauses**

Definition Numbers of WITH clauses in the module.

3.3 Application Scope

Metrics presented in this section are based on the set of Ada 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.

3.3.1 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 source lines

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

ap_ssbra Number of “brace” lines

Definition Number of lines containing only a block tag (e.g. begin, end) in the application source files.

ap_sblank Total number of empty lines

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

ap_scomm Total number of source lines of comments

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

3.3.2 Application Aggregates

ap_func Number of functions

Definition Number of functions defined in the application.

Alias LMA

ap_stat **Number of statements**

Definition Sum of the numbers of executable statement (i.e. lc_stat) for all the functions defined in the application.

ap_vg **Sum of cyclomatic numbers**

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

Alias VGA, ap_cyclo

3.3.3 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_level **Call graph depth**

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

Alias GA_LEVEL

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

Chapter 4

Programming Rules

This section describes the default set of rules provided by Logiscope Ada *RuleChecker*. About half of these rules can be customized by modifying parameters in the selected Rules Set file (see next chapter).

address **"use at" Clause**

Description	The <code>use at</code> clause is forbidden for local variables and parameters.
Justification	Improves code portability.

aggregate **Choices in Aggregates**

Description	In aggregates component associations shall be all named or all positional.
Justification	Makes the code easier to read.

Example:

```
-- do not write
(14, MONTH=>JULY, YEAR=>1789)

-- write
(14, JULY, 1789)
-- or
(DAY=>14, MONTH=>JULY, YEAR=>1789)
```

arraytyp **Array Types**

Description	An array has to be declared as a type and not used directly inside a declaration.
Justification	Makes maintenance easier by avoiding the scattering of array types among the code, often with the same specification.

Example:

```
-- do not write
Var_Array : array (1 .. 4) of Var;

-- write
type My_Array is array (1 .. 4) of Var;
Var_Array : My_Array;
```

const **Literal Constants**

Description Numbers, characters and strings have to be declared as constants instead of being used as literals inside a program. Characters are allowed inside enumerative types.

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

Parameters A list of character strings representing the allowed literal constants.

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

count **"count" Attribute**

Description The `count` attribute is forbidden.

Justification Improves code portability.

dblneg **Double Negation**

Description Only one `not` operator shall be used in each expression.

Justification Makes the code easier to understand.

enum **Enumerations**

Description The number of literal values in an enumerated type is limited.

Parameters A number representing the maximum authorized number of values.

Justification Makes the code easier to understand and maintain.

excephand **Handled Exceptions**

Description Each function or procedure shall handle a predefined list of exceptions.

Parameters A list of character strings representing names of exceptions that shall be handled.

Justification Makes the code more robust.

exprlevel **Level of Complexity of Expression**

Description The complexity of an expression is limited by the depth of the syntactic tree used to represent it. The parenthesis, the association of a name and the `"."` (dot) operator do not increase the level of complexity.

By default, the maximum authorized complexity level is 3.

Parameters A number representing the maximum authorized complexity level.

Justification Makes the code easier to read.

Example:

```
The level of a + b is 2.
The level of a + b + c or (a + b) + c is 3.
```

genpack **Generic Packages**

Description Each instantiation of a generic package belonging to a pre-defined list of packages shall be included in another package.

Parameters A list of character strings representing the names of the packages from which the generic packages shall not be instantiated outside an including package.

Justification Reinforces code structuration.

Example:

```
-- if the Text_Io generic package is listed in the
-- parameter list,
-- do not write
package My_Integer_Io is new
Ada.Text_Io.Integer_Io (My_Integer);

-- encapsulate the instantiation in a package
-- write
package Pack is
  package My_Integer_Io is new
Ada.Text_Io.Integer_Io (My_Integer);
  ...
end Pack;
```

goto **Goto Statement**

- Description** The `goto` statement must not be used.
It is possible to specify certain labels which are authorized.
By default, all `goto` statements are forbidden.
- Parameters** A list of strings representing the labels that can be used with the `goto` statement.
- Justification** 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 Comments**

- Description** Modules must be preceded by a header comment.
It is possible to define a format for this comment.
By default, a header comment with the name of the file, its author, its date and possible remarks is required for each module (see below example).
- Parameters** A list of character 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 Header Comments

- Description** Packages and subprograms must be preceded by a comment.
- It is possible to define a format for this comment depending on the type of the package or subprogram (**pack_decl**, **pack_body**, **proc_decl**, **proc_body**, **func_decl**, **func_body**).
- By default, only a comment beginning with "" is required for packages or subprograms.
- Parameters** Six lists of character strings concerning six cases listed above. Each list begins with one of the six strings (**proc_decl** for instance), followed by a string representing the regular expression.
- Justification** Makes the code easier to read.

identfmt Identifier Format

- Description** The identifier of a package, subprogram, task, task type, entry, type, constant, variable or exception exported by a compilation unit must have a format corresponding to the category of the declaration.
- By default, no restrictions are imposed.
- 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 an identifier (of a package, subprogram, task, task type, entry, type, constant, variable or exception) exported by a compilation unit must be between a minimum and a maximum value.
- By default, the above identifiers must have between 5 and 25 characters.
- 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.

loopexit Exits in Loops

Description The `exit` statement shall be unique inside a `loop` statement, it shall be associated to the `when` statement and shall not be at the beginning of the loop.

Justification Having only one exit point in a loop makes it easier to understand. The `when` statement in an `exit` is easier to read than the `exit` statement within an `if` statement.

loopname Named Loops

Description Every loop shall have a name. Each loop shall have a different name.

By default, the name of a loop shall be unique in each compilation unit.

Parameters A character string with two possible values, "**unit**" which means that the name of the loop shall be unique in each compilation unit, or "**subprogram**" which means that the name of the loop shall be unique in the body of each function or procedure.

Justification Makes the code easier to understand.

Ivarinit Local Variable Initialization

Description Local variables shall be initialized in the first branch of the declaration block. That means before any conditional statement. Out parameters shall be initialized in the first branch of the body of the function or procedure

Note Potential initializations by calling a procedure are not taken into account.

Limitation Violations are detected for records even if they have default values for their fields.

Justification Reliability.

mainpar Parameters of Main

Description A main program shall not have parameters.

Justification Portability.

others "when others" Forbidden

Description The use of the `when others` clause is forbidden in `case` statements, exception handlers and record variant parts

Justification It is better to anticipate all the possible cases than resorting to a choice with no precise value(s).

noabort	"abort" Statement
Description	The <code>abort</code> statement is forbidden.
Justification	Portability.
parinit	Parameter Default Value
Description	No default value shall be provided for function or procedure parameters.
Justification	Maintainability.
parname	Named Parameters
Description	When calling a function or a procedure, parameters shall be all named or all positional (no named parameters).
Justification	Maintainability.
parord	Parameter Order
Description	Inside a subprogram declaration, parameters must be ordered according to their nature (<code>in</code> , <code>in out</code> or <code>out</code>). Parameters of mode <code>in</code> with a default value are allowed at the end of the list .
Parameters	A list of character strings (from zero to three, with the following possible values: "in" , "out" or "in out") giving the imposed order. No parameter means that the order is indifferent.
Justification	Maintainability.
pragma	Pragma Statement
Description	Using some pragmas is forbidden or authorized. By default all pragmas are authorized.
Parameters	A list of character strings representing names of pragmas. The list shall begin either by "authorized" which indicates the following strings are names of pragmas that can be used, or by "forbidden" which indicates the following strings are names of pragmas that are forbidden.
Justification	Portability.

raisedef	Raise defined exceptions
Description	A subprogram declared in a specification package may only raise in its body exceptions that are defined in that specification package.
Justification	Maintainability.
recnest	Structured Types
Description	The number of levels of structured record types is limited. The level of a record type not containing any record type is 1.
Parameters	A number representing the maximum authorized level.
Justification	Makes the code easier to understand.
resize	Length Clause
Description	The use of the length clause (<code>... 'size use ...;</code>) is forbidden.
Justification	Portability.
retinit	Return Value Initialization
Description	In a function body, each local variable used in the returned expression shall be initialized outside a conditional statement.
Note	Potential initializations by calling a procedure are not taken into account.
Justification	Reliability.
return	"return" Statement
Description	The <code>return</code> statement has to be the last statement of a statements sequence.
Justification	Prevents inaccessible parts of code.
slret	Single "return"
Description	Each procedure or function shall only have one return statement.
Justification	Maintainability: Structured Programming
specbod	Specification and Body
Description	The specification and the body part shall be in different files.
Justification	Maintainability:.

typacs	Access Types
Description	The access types are forbidden. The use of the new clause is forbidden.
Justification	Prevents memory leaks.
typeres	Reserved types
Description	The use of some types in variable or subprogram declarations and return types is forbidden.
Parameters	A list of character strings representing the names of the forbidden types.
Justification	Portability: not relying on predefined types.
use	"use" Clauses
Description	No <code>use</code> clause must be used inside context clauses of a unit. It is possible to specify certain units which are authorized. By default, all <code>use</code> clauses are forbidden.
Parameters	A list of strings representing names that can be used in <code>use</code> clauses.
Justification	Makes the code easier to understand.
varinit	Variable Initialization
Description	Variables must be initialized in their declarations.
Limitation	Violations are detected for records even if they have default values for their fields.
Justification	Ensures correct variable initialization prior to use.
with	"with" Clauses
Description	Using some <code>with</code> clauses is forbidden. By default, all <code>with</code> clauses are authorized.
Parameters	A list of strings representing names of units that can not be used in a <code>with</code> clause.
Justification	Prevents from using non portable or dangerous packages.

nameres **Reserved Names**

Description Using some functions, procedures, tasks or exceptions is forbidden. Only the first use of each item in a function, procedure or task is taken into account.

Parameters A list of character strings representing names of forbidden functions, procedures, tasks or exceptions.

Justification Improves code portability.

specvar **Variables Inside a Specification**

Description No variable must be declared inside the visible part of a package specification.

Justification The good way to have access to the services of a package is via its subprograms, not its variables. The variables of a package specification should only be private.

Chapter 5

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.

5.1 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\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 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.

5.2 Customizing Standard Rule Checking

The precise definition of these rules has been given in the previous chapter.

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
```

enum Enumerations

By default, the maximum number of literal values in an enumerated type is 25:

```
STANDARD enum ON MINMAX 0 25 END STANDARD
```

To change this value to 16, for example:

```
STANDARD enum ON MINMAX 0 16 END STANDARD
```

exceptand Handled Exceptions

By default, no exception handling is imposed on subprograms:

```
STANDARD exceptand ON LIST END LIST END STANDARD
```

To impose the handling of **Storage_Error** and **Constraint_Error**:

```
STANDARD exceptand ON LIST "Storage_Error" "Constraint_Error" END LIST  
END STANDARD
```

exprlevel Level of Complexity of Expression

By default, the maximum authorized level of complexity is 3.

```
STANDARD exprlevel ON MINMAX 0 3 END STANDARD
```

To change this value to 7, for example:

```
STANDARD exprlevel ON MINMAX 0 7 END STANDARD
```

genpack Generic Packages

By default, no instantiation of a generic package is forbidden outside an including package:

```
STANDARD genpack ON LIST END LIST END STANDARD
```

Not to instantiate the generic packages inside `Text_Io` outside an including package :

```
STANDARD genpack ON LIST "Text_Io" 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

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",
"[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 each file:

```

STANDARD Headercom ON
LIST "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: John Doe
-- Date: 26/06/2002
-- Remarks: example of comments
-----

```

headercom Function Header Comments

It is possible to define a format for the comment preceding a package or a subprogram, depending on the type of the package or subprogram (**pack_decl**, **pack_body**, **proc_decl**, **proc_body**, **func_decl**, **func_body**).

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 "pack_decl"      ".*" END LIST
LIST "pack_body"     ".*" END LIST
LIST "proc_decl"     ".*" END LIST
LIST "proc_body"     ".*" END LIST
LIST "func_decl"     ".*" END LIST

```

```
LIST "func_body"          ".*" END LIST
END STANDARD
```

Here is another example, with different required comments depending on the item type:

```
STANDARD headercom ON
LIST "pack_decl"         "Definition of the package declaration:"
                        "Author: [A-Z][a-z]*"
END LIST
LIST "pack_body"        "Definition of the package body:"
                        "Author: [A-Z][a-z]*"
END LIST
LIST "proc_decl"        "Declaration of the procedure:"
                        "Date: [0-9][0-9]/[0-9][0-9]/[0-9][0-9]"
END LIST
LIST "proc_body"       "Body of the procedure:"
                        "Remarks:"
END LIST
LIST "func_decl"        "Declaration of the function:"
                        "Date: [0-9][0-9]/[0-9][0-9]/[0-9][0-9]"
END LIST
LIST "func_body"       "Body of the function:"
                        "Remarks:"
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
variable	variable name	any
parameter	parameter name	variable
constant	constant name	any
exception	exception name	any
procedure	procedure name	subprogram, any
function	function name	subprogram, any
subprogram	subprogram name	any
package	package name	any
task	task name	any
task_type	task type name	type, task, any
entry	entry name	any

The third column represents inherited types: for instance, for no distinction between the **procedure** and the **function** categories, simply define a particular format for the **subprogram** category, 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, no restrictions are imposed:

```
STANDARD identfmt ON
LIST "any"           ".*"
     "type"          ".*"
     "variable"      ".*"
     "constant"      ".*"
     "exception"     ".*"
     "procedure"     ".*"
     "function"      ".*"
     "subprogram"    ".*"
     "package"       ".*"
     "task"          ".*"
     "entry"         ".*"
     "task_type"     ".*"
END LIST END STANDARD
```

For the subprograms to begin with "S_", the constants to have no lower case letter and no underscore at the beginning and the end, the variables to begin with "V_" and all other identifiers not to begin or end with an underscore:

```
STANDARD identfmt ON
LIST "any"           "[^_](.*[^_])?$"
     "subprogram"    "S_.*[^_]"
     "const"         "[A-Z0-9]([A-Z0-9_]*[A-Z0-9])?$"
     "variable"      "V_.*[^_]"
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, all the identifiers must have between 5 and 25 characters:

```
STANDARD identl ON
LIST "any"           MINMAX 1 25
     "type"          MINMAX 5 25
     "variable"      MINMAX 5 25
     "constant"      MINMAX 5 25
     "exception"     MINMAX 5 25
     "procedure"     MINMAX 5 25
     "function"      MINMAX 5 25
     "subprogram"    MINMAX 5 25
     "package"       MINMAX 5 25
     "task"          MINMAX 5 25
     "entry"         MINMAX 5 25
     "task_type"     MINMAX 5 25
END LIST END STANDARD
```

loopname Named Loops

By default, every loop shall have a name and the name of a loop shall be unique in each compilation unit:

```
STANDARD loopname ON "unit" END STANDARD
```

To have the loop name to be unique in the body of each function or procedure:

```
STANDARD loopname ON "subprogram" END STANDARD
```


nameres Reserved Names

By default, there are no reserved names:

```
STANDARD nameres ON LIST END LIST END STANDARD
```

To forbid the use of the subprograms `Unchecked_Deallocation` and `Unchecked_Conversion`:

```
STANDARD nameres ON LIST "Unchecked_Deallocation"
"Unchecked_Conversion"
END LIST END STANDARD
```

parord Parameter OrderOrder

By default the authorized order of parameters in a subprogram is in parameters first, then in out parameters and then out parameters:

```
STANDARD parord ON LIST "in" "in out" "out" END LIST END STANDARD
```

To authorize only in parameters and then out parameters:

```
STANDARD parord ON LIST "in" "out" END LIST END STANDARD
```

pragma Pragma Statement

By default, all pragmas are authorized:

```
STANDARD pragma ON LIST "forbidden" END LIST END STANDARD
```

To forbid all pragmas :

```
STANDARD pragma ON LIST "authorized" END LIST END STANDARD
```

To forbid the pragmas `SYSTEM_NAME`, `MEMORY_SIZE`, `STORAGE_UNIT` and `SHARED`:

```
STANDARD pragma ON LIST "forbidden" "SYSTEM_NAME" "MEMORY_SIZE"
"STORAGE_UNIT" "SHARED" END LIST END STANDARD
```

To authorize only the Ada95 pragmas:

```
STANDARD pragma ON LIST "authorized" "All_Calls_Remote" "Asynchronous"
"Atomic" "Atomic_Components" "Attach_Handler" "Controlled" "Convention"
"Discard_Names" "Elaborate" "Elaborate_All" "Elaborate_Body" "Export"
"Import" "Inline" "Inspection_Point" "Interrupt_Handler"
"Interrupt_Priority" "Linker_Options" "List" "Locking_Policy"
"Normalize_Scalars" "Optimize" "Pack" "Page" "Preelaborate" "Priority"
"Pure" "Queuing_Policy" "Remote_Call_Interface" "Remote_Types" "Restrictions"
"Reviewable" "Shared_Passive" "Storage_Size" "Suppress"
"Task_Dispatching_Policy" "Volatile" "Volatile_Components" END LIST END
STANDARD
```

To authorize only the Ada83 pragmas:

```
STANDARD pragma ON LIST "authorized" "CONTROLLED" "ELABORATE" "INLINE"
"INTERFACE" "LIST" "MEMORY_SIZE" "OPTIMIZE" "PACK" "PAGE" "PRIORITY"
"SHARED" "STORAGE_UNIT" "SUPPRESS" "SYSTEM_NAME" END LIST END STANDARD
```

recnest Structured Types

By default, the maximum authorized level of structured record types is 5:

```
STANDARD recnest ON MINMAX 0 5 END STANDARD
```

To change this value to 3, for example:

```
STANDARD recnest ON MINMAX 0 3 END STANDARD
```

typeres Reserved Types

By default, there are no reserved types:

```
STANDARD typeres ON LIST END LIST END STANDARD
```

To forbid the types Integer and Float:

```
STANDARD typeres ON LIST "Integer" "Float" END LIST END STANDARD
```

use "use" Clauses

By default, all `use` clauses are forbidden:

```
STANDARD use ON LIST END LIST END STANDARD
```

To authorize the use of `Text_Io` and `System`:

```
STANDARD use ON LIST "Text_Io" "System" END LIST END STANDARD
```

with "with" Clauses

By default, all `with` clauses are authorized:

```
STANDARD with ON LIST END LIST END STANDARD
```

To forbid the `with Standard; clause`:

```
STANDARD with ON LIST "Standard" END LIST END STANDARD
```

5.3 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
```

5.4 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

© Copyright 2014

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.

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.