RuleChecker & QualityChecker Java Reference Manual

Before using this information, be sure to read the general information under “Notices” section, on page 63.

About This Manual

Audience

This manual is intended for Kalimetrix Logiscope™ *RuleChecker & QualityChecker* users for Java 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 a dedicated document:

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

Overview

Java Project Settings

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

Command Line Mode

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

Standard Metrics

Chapter 3 specifies the source code metrics computed by *Logiscope QualityChecker Java*.

Programming Rules

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

Customizing Standard Rules and Rule Sets

Chapter 5 describes the way to customise standard predefined rule checking with *Logiscope RuleChecker Java*.

Conventions

The following typographical conventions are used:

bold	literals such as tool names (**Studio**) and file extension (***.java**),
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 **Java Project Settings**

1.1 Input Data 1
1.2 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 *Examples of Use* *10*

Chapter 3 ***Standard Metrics***

3.1 Function Scope 12
3.1.1 Line Counting 12
3.1.2 Lexical and Syntactic Items 13
3.1.3 Halstead Metrics 14
3.1.4 Control Graph 16
3.1.5 Relative Call Graph 18
3.2 Class Scope 19
3.2.1 Line Counting 19
3.2.2 Lexical and Syntactic Items 19
3.2.3 Halstead Metrics 19
3.2.4 Data Flow 20
3.2.5 Statistical Aggregates of Function Metrics 21
3.2.6 Inheritance Tree 24
3.2.7 Use Graph 24
3.3 Module Scope 26
3.3.1 Line Counting 26
3.3.2 Lexical and syntactic items 27
3.4 Package Scope 28
3.4.1 Basic Metrics 28
3.4.2 Halstead Metrics 28
3.4.3 Statistical Aggregates of Class Metrics 29
3.4.4 Statistical Aggregates of Function Metrics 30
3.4.5 Inheritance 32

3.5 Application Scope 33
3.5.1 Line Counting 33
3.5.2 Application Aggregates 34
3.5.3 Application Call Graph 35
3.5.4 Inheritance Tree 36

Chapter 4 ***Programming Rules***

Chapter 5 **Customizing Standard Rules and Rule Sets**
5.1 Modifying the Rule Set 53
5.2 Customizing Standard Rule Checking 54
5.3 Renaming Rules 60
5.4 Creating New Rule Scripts 61

Chapter 6 **Notices**

Chapter 1

Java Project Settings

This chapter details specifics of the Logiscope Java projects.

Logiscope Java projects (".ttp") can be created using:

• **Logiscope Studio Wizard:** a graphical interface requiring a user interaction, please refer to *IBM Rational Logiscope - RuleChecker & QualityChecker - Getting Started* documentation 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, please refer to Chapter *Command Line Mode* 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 Input Data

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, rules set.

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 will be created.

Source Files

Logiscope Java RuleChecker & QualityChecker 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. 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 file 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 Java source files needed in the above selected directories. The extensions shall be separated with a semi-colon.

Quality Model File

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, classes, application),

applicable to the application 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.

Rules Set File

Logiscope *RuleChecker* allows to automatically check a set of programming rules / coding standards which are gathered within a Rules Set file. This file is used to indicate which rules must be checked and to give parameters to customizable rules.

About seventy programming rules are supplied with Logiscope Java *RuleChecker* (see Chapter *Programming Rules*). About half of these rules can be customized to match the user applicable requirements (see Chapter *Modifying a Predefined Rule*).

1.2 Output Data

Logiscope Repository

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

The source files for a given Java 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.

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/Java/Jonas/src/Account.java
/users/logiscope/samples/Java/Jonas/src/AccountExplBean.java
/users/logiscope/samples/Java/Jonas/src/AccountHome.java
```

Using the command line:

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

will create a new Logiscope Java project file aProject.ttp containing 3 files: Account.java, AccountExplBean.java and AccountHome.java on which the *QualityChecker* and *RuleChecker* 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:

`create -lang java`

`<ttp_file>`	name of a Logiscope project to be created (with the .ttp extension). Path has to be absolute if the option -- is used.
`[-source <suffixes>]`	where <suffixes> is the list of accepted suffixes for the source files. Default is "*.java".
`[-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, 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/Java/ 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.

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 to *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:

`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.

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 <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 3

Standard Metrics

Logiscope QualityChecker 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 Java functions, classes, modules, packages, 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.

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 functions defined in the source files specified in the Logiscope Project under analysis.
- The *Class scope*: the metrics are available for each Java classes defined in the 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 Java source files specified in the Logiscope Project under analysis.
- The *Package scope*: the metrics are available for each Java package defined in the source files specified in the Logiscope Project under analysis.
- The *Application scope*: the metrics are available for the set of Java source files specified in the Logiscope Project .

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 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 comments in the function.
Alias LCOM

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_parse Number of lines not parsed

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

3.1.2 Lexical and Syntactic Items

lc_dclstat Number of declarative statements

Definition Number of declarations in a method body.

ic_except Number of raised exceptions

Definition Number of exceptions declared by the keyword `throws` in a method.

ic_param Number of parameters

Definition Number of a formal parameters in the function.

Alias PARA

lc_stat Number of statements

Definition Number of executable statements in a function's body.

Executable statements are:

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

lc_stat that can be parametrized to count the statements a familiar way:

- 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 (default),
- if the parameter **"no_decl_stat"** is provided, declarative statements are omitted, as well as statements omitted with the parameter **"no_null_stat"**.

Alias STMT

3.1.3 Halstead Metrics

The four following metrics allows to compute all metrics defined by Halstead [Hal, 77] at function level in the Logiscope Quality Model file. See the Quality Model file Halstead.ref.

For more details on Halstead metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

n1 Number of distinct operators

Definition Number of different operators used in a function.

The following are operators:

- Statements:

IF	**ELSE**	**WHILE( )**	**DO WHILE( )**
RETURN	**FOR(;;)**	**SWITCH**	**BREAK**
CONTINUE	**CASE**	**DEFAULT**	**THROW**
TRY	SYNCHRONIZED	CATCH	
;	(empty statement)		

- Expressions:
  - Unary operators:

+ -	unary plus or minus
++ --	pre-/post- increment or decrement
!	negation
~	complement of 1 or destructor
new	new
delete	delete
isinstance	instance of

  - 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)
...()	function call	(ex: func**(**1**)**)

- Specifiers: class, package, private, public, protected, static, volatile, native, abstract, synchronized, trancient, final, extend, implement.

N1 Total number of operators

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

Definition Number of different operands used in a function.

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, function names, class names, package names,
- this,
- super,
- predefined types : boolean, long, int, byte, short, float, char, double, void

N2 Total number of operands

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).

3.1.4 Control Graph

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

- *Kalimetrix Logiscope - Basic Concepts*.

ct_bran Number of destructuring statements

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

ct_decis Number of decisions

Definition Number of selective structures in a function : `if`, `switch`.

Alias N_STRUCT

ct_degree Maximum degree

Definition Maximum number of edges departing from a node of the function control graph.

ct_edge Number of edges

Definition Number of edges of a function control graph.

Alias N_EDGES

ct_exit Number of exits

Definition Number of exit nodes in the control graph of the function : *`return`*, *`exit`* statements.

Alias N_OUT

ct_loop Number of loops

Definition Number of loop 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 of a function control graph.

Alias N_NODES

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.

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.

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 a selective or iterative structure that does not contains branching or auxiliary exit statements: `goto`, `break`, `continue` or `return`.

3.1.5 Relative Call Graph

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

- *Kalimetrix Logiscope - Basic Concepts*.

cg_levels Number of relative call graph levels

Definition Number of levels of the relative call graph of the function.

Alias LEVELS

cg_entropy Relative call graph entropy

Definition This metric proposed by SCHUTT [SHT, 77] applies to the system call graph. It is an indicator of call graph analysability, characterizing both width and depth of the call graph:

$$H(G_A) = \frac{1}{|x|}\sum_{i=1}^{Np} |x_i| \log_2 \frac{|x|}{|x_i|}$$

where |xi| is the number of components in the ith path.

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.

Alias HIER_CPX

cg_strucpx Relative call graph structural complexity

Definition Average number of calls per component: i.e. number of calls between components divided by the number of components.

Alias STRU_CPX

cg_testab Relative call graph system testability

Definition

$$ST = \frac{1}{Np}\left(\sum_{i=1}^{Np} \frac{1}{TP_i}\right)^{-1}$$

Np is the number of paths through the system.

TP_i is the testability of the i^{th} call path.

The definition involves the number of paths and the test difficulty level for each path. The result obtained can help to evaluate the software reliability.

Alias TESTBTY

3.2 Class Scope

3.2.1 Line Counting

cl_line Number of lines

Definition Total number of lines in the class or interface.

cl_comm Number of lines of comments

Definition Number of comment lines of comment in the class or interface. Comments located outside the class are not counted.

3.2.2 Lexical and Syntactic Items

cl_dclstat Number of declarative statements

Definition Number of declarations of fields and methods in a class or an interface.

cl_stat Number of statements

Definition Number of statements in all methods and initialization code of a class. This counting of staf tements and optional parameters ***"no_null_stat"*** and ***"no_decl_stat"*** are explained in **lc_stat** in the Function Scope part.

Note Because the value of the metric **cl_stat** for the class scope depends on the value of **lc_stat** for the method scope, it is strongly recommended to use the same parametrization for the two scopes.

3.2.3 Halstead Metrics

The four following metrics allows to compute all metrics defined by M.H. Halstead [Hal, 77] at class level in the Logiscope Quality Model file.
See the Quality Model file: Halstead.ref.

For more details on Halstead metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

cl_n1 Number of distinct operators

Definition Number of different operators used in the class.

cl_N1 Total number of operators

Definition Total number of operators used in the class.

cl_n2 Number of distinct operands

Definition Number of different operands used in the class.

cl_N2 Total number of operands

Definition Total number of operands used in the class.

3.2.4 Data Flow

cl_interf Number of implemented interfaces

Definition Number of declared interfaces implemented by a class or extended by an interface.

cl_extend Number of extended classes

Definition Equals 1 if the class extends another class, 0 otherwise.

cl_subclass Number of included classes

Definition Number of classes or interfaces declared inside a class or an interface.
Note Anonymous classes are not taken into account.

cl_data Total number of attributes

Definition Total number of data members declared inside a class declaration.
Alias cl_field

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_final Number of final attributes

Definition Number of data members declared in a class declaration with the attribute `final`.
Note For interfaces, **cl_data_final** is equal to **cl_data**.
Alias cl_field_final

cl_data_const Number of constants

Definition Number of data members declared in a class declaration with the attributes `final` and `static`.
Note For interfaces, **cl_data_const** is equal to **cl_data**.
Alias cl_field_const

cl_data_static Number of class attributes

Definition Number of data members declared in a class declaration with the attribute `static` and without the `final` attribute.
Note For interfaces, **cl_data_static** is equal to 0.
Alias cl_field_static

cl_data_pack Number of attributes in package scope

Definition Number of data members declared in the class declaration without any of the attributes `private`, `protected` or `public`.
Note For public classes or interfaces, **cl_data_pack** is equal to 0.
Alias cl_field_pack

cl_data_nostat Number of instance attributes

Definition Number of fields declared in a class declaration without attribute `static`.
Note For interfaces, **cl_data_nostat** is equal to 0.
Alias cl_field_nostat

3.2.5 Statistical Aggregates of Function Metrics

cl_func Total number of methods

Definition Total number of methods declared inside a class.

Alias cl_meth

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_abstract Number of abstract methods

Definition Number of methods declared in a class declaration with the attribute `abstract`.

Note For interfaces, **cl_func_abstract** is equal to **cl_func**.

Alias cl_meth_abstract

cl_func_native Number of methods implemented in another language

Definition Number of methods declared in a class declaration with the attribute `native`.

Note For interfaces, **cl_func_native** should be 0.

Alias cl_meth_native

cl_func_pack Number of methods in package scope

Definition Number of methods declared in a class declaration without any of the attributes `private`, `protected` or `public`.

Note For public interfaces, **cl_func_pack** is equal to 0.

Alias cl_meth_pack

cl_func_static Number of class methods

Definition Number of methods declared in a class declaration with the attribute `static`.

Note For interfaces, **cl_func_static** is equal to 0.

The sum of **cl_func_static** and **cl_func_nostat** gives the total number of methods **cl_func**.

Alias cl_meth_static

cl_func_nostat Number of instance methods

Definition Number of methods declared in a class declaration without the attribute `static`.

Note For interfaces, **cl_func_nostat** is equal to **cl_func**.

The sum of **cl_func_static** and **cl_func_nostat** gives the total number of methods **cl_func**.

Alias cl_meth_nostat

cl_fpriv_path Sum of paths for private class methods

Definition Sum of non-cyclic execution paths for each class's private methods. This metric is an indicator of the static complexity of the private part of the class.

Alias LMPIPATH

cl_fprot_path Sum of paths for protected class methodss

Definition Sum of non-cyclic execution paths for each class's protected methods. This metric is an indicator of the static complexity of the class protected part.

Alias LMPOPATH

cl_fpubl_path Sum of paths for public class methods

Definition Sum of non-cyclic execution paths for each class's public methods. This metric is an indicator of the static complexity of the public part of the class.

Alias LMPUPATH

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

3.2.6 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.

Justification The children number of a class is an indicator of the class criticalness within a given system. In fact, more children a class has, more the modifications made to the class will induce changes in the global system.

Alias NOC, in_dderived

3.2.7 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 users classes

Definition Number of classes which use directly a class.

cu_cusers Number of users classes

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

3.3 Module Scope

3.3.1 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_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.

3.3.2 Lexical and syntactic items

md_class Number of classes

Definition Number of classes declared at the first level of the file.

md_interf Number of interfaces

Definition Number of interfaces declared at the first level of the file.

md_import_pack Number of imported packages

Definition Number of packages appearing in the import statement of a module.
The parameter of the import stement is supposed to be a package name if it is a simple name (without a dot in it) or if it is not used as a type in the module.

md_import_demd Number of importations on demand

Definition Number of import statements in a module whose parameter is a generic name (ended by .*).

md_import_type Number of imported types

Definition Number of types appearing in the import statements of a module.
The parameter of the import stement is supposed to be a type name if it is not a simple name (with at least a dot in it) or if it is used as a type in the module.

md_dclstat Number of declarative statements

Definition Total number of declarations in the method bodies in the file.

md_stat Number of statements

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

3.4 Package Scope

3.4.1 Basic Metrics

pk_line Number of lines

Definition Total number of lines in the files containing the package.

pk_com: Number of lines of comments

Definition Total number of comment lines in the package.
Comments located outside the package are not counted.

pk_file Number of files

Definition Total number of files within the package.

pk_pkused Number of used packages

Definition Number of imported packages of the package.

3.4.2 Halstead Metrics

The four following metrics allows to compute all metrics defined by Halstead [Hal, 77] at package level in the Logiscope Quality Model file.
See the Quality Model file: Halstead.ref.

For more details on Halstead metrics, please refer to:

- *Kalimetrix Logiscope - Basic Concepts*.

pk_n1 Number of distinct operators

Definition Number of distinct operators referenced in the package.

pk_n2 Number of distinct operands

Definition Number of distinct operands referenced in the package.

pk_N1 Total number of operators

Definition Total number of operators referenced in the package.

pk_N2 Total number of operands

Definition Total number of operands referenced in the package.

3.4.3 Statistical Aggregates of Class Metrics

pk_class Number of classes

Definition Total number of classes declared in the package.
Nested classes are counted.

pk_interf Number of interfaces

Definition Total number of interfaces declared in the package.

pk_const Number of constants

Definition Total number of constants declared in the classes of the package.

pk_data Number of attributes

Definition Total number of data declared in the classes of the package.

pk_data_priv Number of private attributes

Definition Total number of data explicitely declared with the “private” keyword in the classes of the package.

pk_data_prot Number of protected attributes

Definition Total number of data explicitely declared with the “protected” keyword in the classes of the package.

pk_data_publ Number of public attributes

Definition Total number of data explicitely declared with the “public” keyword in the classes of the package.

pk_data_stat Number of static attributes

Definition Total number of data explicitely declared with the “static” keyword in the classes of the package.

pk_except Number of raised exceptions

Definition Total number of exceptions declared by the keyword `throw` in the method declaration of the package.

pk_raise Number of raising an exceptions raises

Definition Total number of occurrences of `throw` keyword in the classes of the package.

pk_try Number of exception handlers

Definition Total number of occurrences of `try` blocks in the classes of the package.

pk_type Number of public classes

Definition Total number of public classes of the package.

3.4.4 Statistical Aggregates of Function Metrics

pk_cpx Sum of size of statements

Definition Sum of the size (number of operands and operators) of the statements in the package.

pk_cpx_max Maximum size of statements

Definition Maximum number of operands and operators in a statement of the package.

pk_func Number of functions

Definition Total number of functions declared in the classes of the package

pk_func_priv Number of private functions

Definition Total number of functions explicitely declared with the “private” keyword in the classes of the package.

pk_func_prot Number of protected functions

Definition Total number of functions explicitely declared with the “protected” keyword in the classes of the package.

pk_func_publ Number of public functions

Definition Total number of functions explicitely declared with the “public” keyword in the classes of the package.

pk_func_stat Number of static functions

Definition Total number of functions explicitely declared with the “static” keyword in the classes of the package.

pk_func_abstract Number of abstract functions

Definition Total number of abstract functions in the classes of the package.

pk_func_used Sum of called functions

Definition Number of calls of functions by a function declared in the classes of the package.

pk_func_used_max Maximum number of called functions

Definition Maximum number of calls of functions by a function declared in the classes of the package.

pk_levl Sum of maximum nested levels

Definition Sum of nested levels (ct_nest) in the functions declared in the classes of the package.

pk_levl_max Maximum nested levels

Definition Maximum number of nested levels (ct_nest) in a function declared in the classes of the package.

pk_path Sum of non-cyclic paths

Definition Sum of non-cyclic paths (ct_path) in the functions declared in the classes of the package.

pk_path_max Maximum number of non-cyclic paths

Definition Maximum number of non-cyclic paths (ct_path) in a function declared in the classes of the package.

pk_param Sum of function parameters

Definition Sum of the number of formal parameters (ic_param) in the functions declared in the classes of the package.

pk_param_max Maximum number of parameters

Definition Maximum number of formal parameters (ic_param) in a function declared in the classes of the package.

pk_stmt Sum of statements

Definition Sum of executable statements (lc_stat) in the functions declared in the classes of the package.

pk_stmt_max Maximum number of statements

Definition Maximum number of executable statements (lc_stat) in a function declared in the classes of the package.

pk_vg Sum of cyclomatic numbers

Definition Sum of cyclomatic numbers (ct_vg) of the functions declared in the classes of the package.

pk_vg_max Maximum cyclomatic number

Definition Maximum cyclomatic numbers (ct_vg) in a function declared in the classes of the package.

3.4.5 Inheritance

pk_extend Total number of extends

Definition Number of classes referenced in the “extend” directives of the classes in the package.If a class is referenced several times, it is counted several times.

pk_implement Total number of implement

Definition Number of classes referenced in the “implement” directives of the classes in the package. A class referenced several times is counted several times.

pk_inh_levl Sum of depth of the inheritance tree

Definition Sum of the depth of the inheritance tree of each class declared in the package.

pk_inh_levl_max Depth of the inheritance tree

Definition Maximum depth of an inheritance tree of a class declared in the package.

3.5 Application Scope

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

3.5.1 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 Total number of lines of comments

Definition Number of lines of comments in the application source files.

ap_ssbra Number of lines with lone braces

Definition Number of lines containing only a single brace character : i.e. “{“ or “}” application source files.

3.5.2 Application Aggregates

ap_clas Number of application classes

Definition Number of classes in the application.

Alias LCA

ap_func Number of application functions

Definition Number of functions in the application. The application is defined by the list of analyzed files.

Alias LMA

ap_interf_func Number of application interface functions

Definition Number of interface functions in the application.

ap_npm Number of public methods

Definition Number of public method in the application.

Alias LCA

ap_line Number of function lines

Definition Sum of numbers of lines (i.e. lc_line) of all the functions defined in the application.

See also ap_sloc

ap_stat Number of statements

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

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

3.5.3 Application Call Graph

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.

Justification Excessive use of recursiveness increases the global complexity of the application and may diminish system performances.

Alias GA_CYCLE

ap_cg_edge Number of Edges in the Call graph

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

Alias GA_EDGE

ap_cg_levl Number of Levels in the Call graph

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

Justification Too many call graph levels indicates a strong hierarchy of calls among system functions. This may be due to incorrectly implemented object-coupling relationships.

Alias GA_LEVL

ap_cg_maxdeg Maximum of Calling/Called

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

Languages C, ADA

Alias GA_MAXDEG

ap_cg_maxin Maximum of Calling

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

Alias GA_MAX_IN

ap_cg_maxout Maximum of Called

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

Alias GA_MAX_OUT

ap_cg_node Number of Nodes in the Call graph

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 Number of Roots

Definition Number of roots functions in the call graph of Application functions.

Alias GA_NSP

ap_cg_leaf Number of Leaves

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

Alias GA_NSS

3.5.4 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

$$ap\_inhg\_cpx = \frac{SUM(N * i)}{SUM(N)}$$

where N is the number of nodes for level i.

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

Chapter 4

Programming Rules

This section describes the default set of rules provided by Logiscope Java *RuleChecker*. About half of these rules can be customized by modifying parameters in the Rule Set file (see Chapter Customizing Standard Metrics & 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 (=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --) must 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:
	•a variable has to be assigned only once,
	•with multiple assignments, an assigned variable can appear only where it has been assigned.
Justification	Removes ambiguity about the evaluation order.

Example:

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

// but you can write
(new B).i = i = 7;

// the following expressions are detected as a violation,
// but they are not really multiple assignments
// nevertheless, this should be avoided
(new B).i = (new B).i = 7;
nextval().i = nextval().i = 7;
```

blockdecl Declarations in blocks

Description	Declarations must appear at the beginning of blocks.
Justification	Makes the code easier to read.

brkcont break and continue forbidden

Description	`Break` and `continue` statements are forbidden inside conditional expressions in control statements (`for`, `do`, `while`, labeled statements).
	Nevertheless, the `break` statement is allowed in the block statement of the `switch` statement.
	It is possible to choose between three options:
	• **in_switch** (or no parameter) means that the `break` are allowed in `switch` statements, `break` and `continue` are forbidden everywhere else,
	• **without_label** means that any `break` or `continue` without a label is allowed,
	• **with_label** means that any `break` and `continue` with a label is allowed, `break` and `continue` without a label is forbidden everywhere.
Parameters	One of the three character strings explained above.
Justification	Like a `goto`, these instructions break down code structure. Prohibiting them in loops makes the code easier to understand.

condop No ternary operator

Description	The ternary conditional operator `? ... : ...` must not be used.
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, the allowed literal constants are **0**, **1**, **'\0'** and the character string literals.
Parameters	A list of character strings representing the allowed literal constants. The **"*"** character can be used in constants definition to indicate that only the beginning of the literal shall match the definition in parameter
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
String tab = new String(100);
int i;
...
if (i == 7) {
    p = "Hello World.\n";
}

// write
static final int TAB_SIZE =100
static final int ok =7, ko =11;
static final String HelloWorld = "Hello World.\n";
String tab = new String(TAB_SIZE);
i_val i;
...
if (i == ok) {
    p = HelloWorld;
}
```

constrdef Default constructor

Description	Each class must contain the explicit declaration of at least a constructor.
	By default, a default constructor (without parameters) is required for each class.
Parameters	The character string **"default"**, which, if used, requires a default constructor (without parameters) to be provided.
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--;
    }
}
```

declinitsep Declaration and initialisation separate

Description	Declaration and initialisation of a variable shall be done separately.
Justification	Maintainability.

declord Declaration order

Description	In a class, declarations must follow a particular order, given in the parameters of the rule. The order depends on the types of the declarations.
	The type is defined by:
	•the access type (**public**, **protected**, **private** or **package** if no access type is specified),
	•the scope (**class** or **instance**),
	•the variable type (**constant**, **variable**, **method**, **constructor**, **type**).
	The order is defined by an ordered list of strings defining a set of declaration types.
	A declaration of one type can not follow a declaration of another type if its type matches a set of type that is before the set of types of the first one in the ordered list.
	A declaration matches a set of types if the set of types is the first of the list of the highest number of criteria which includes the type of the declaration.

Parameters	A list of character strings representing the declaration types in the wanted order. Each string contains a set of the above keywords. Several keywords of the category are alternatives. Several categories increase the number of criteria of the set. In addition to the keywords described above, **allaccess** means private, protected, public or package; **allscope** means class or instance; **alldecl** means constant, variable, method, constructor or type; **others** means any types not listed above.
Notes	Class definitions have not to contain all the types defined in the standard.
	If the constructor type does not appear in the list, constructors will be considered as ordinary methods.
	It is advisable to use **allaccess**, **allscope** and **alldecl** to increase the number of criteria of a set. For instance, use "allaccess allscope constructor" to match any constructor, but use "method" to match the methods not matched by other sets.
Justification	Makes the code easier to read.

Example:

```
// if the standard has the following strings
// in this order:
// "allaccess allscope constructor" "public class method"
// "public method" "method" "constant" "others",
// following declarations are allowed:
class aClass {
  public aClass(){}
  public int f() {}
  int i;
}
class aClass {
  static public void p() {}
  public int f(int j) {}
  int f() {}
  static final int ID = 123;
  class subClass { }
}
// and not the following one:
class aClass  {
int f() {}
public F() {}
}
```

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 private part of a class are authorized.
Parameters	A list of character strings corresponding to the forbidden access specifiers for the data members. The keyword **package** indicates that no access specifier is provided.

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.

emptythen No empty then

Description	The then part of an if structure shall not be empty.

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 parenthesis for it. In the same way, omit parenthesis in the case of the right operand of an assignment operator. Moreover, omit parenthesis 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 parenthesis, according to the rule above.
Justification	Reliability, Maintainability: Removes ambiguity about the evaluation priorities.

Example:

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

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 10.
Parameters	A number representing the maximum authorized complexity level.
Justification	Maintainability.

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.

filelength File length

Description	A source file shall not contain more than a maximum number of lines.
	By default, the maximum length is limited to 1000 lines. `STANDARD filelength ON 1000 END STANDARD`
Parameter	A number representing the maximum number of lines authorised.
Justification	Analysability

headercom Header comments

Description	Modules, interfaces, classes, methods and attributes must be preceded by a comment.
	It is possible to define a format for this comment depending on the type of the item (**module**, **interface**, **class**, **method**, **attribute**).
	By default, a header comment with the author and the version is required for each class and interface.
Parameters	Five lists of character strings concerning the five cases listed above. Each list begins with one of the five strings (**method** for instance), followed by strings representing the regular expressions.
Justification	Makes the code easier to read.

Example of the default required header comment for classes and interfaces:

```
/**
 *
 * @author  Andrieu
 * @version 1.3, 08/07/96
 */
```

identfmt Identifier format

Description	The identifier of a class, method, type or variable declared in a module must have a format corresponding to the category of the declaration.
	By default, the names of classes, interfaces and constants must begin with an uppercase letter and the names of packages and variables must begin with a lowercase 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 class, method, type or variable identifier has to be between a minimum and a maximum value.
	By default, the packages, classes, interfaces, methods and global variables must have between 5 and 25 characters, the constants between 2 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 package names or in libraries.
	By default, the reserved identifiers are "byvalue", "cast", "const", "future", "generic", "goto", "inner", "operator", "outer", "rest" and "var".
Parameters	A list of character strings representing reserved identifiers.
Justification	Portability.

import Explicit import

Description	Always use explicit import such as: import.io.basic. Never use generic import such as java.io.*.
Justification	Maintainability.

linelength Line length

Description	A line in a source shall not exceed a maximum number of characters.
	By default, the maximum number of characters is limited to 80 lines. STANDARD linelength ON 80 END STANDARD
Parameter	A number representing the maximum number of characters authorised.

Justification	Analysability, Portability

mclass A single class definition per file

Description	A file must not contain more than one class definition.
	Nested classes are tolerated.
Justification	Analysability.

mname File names

Description	A class name and the name of the file in which it is declared or defined must be closely related. The name of the public class declared in the file is taken into account. If no public class is declared the name of the first declared class is taken into account.
	Two modes of comparison are available:
	•If a parameter is provided, the comparison is made only on alphanumeric characters and is not case sensitive. The part of the file name taken into account 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.
	•If no parameter is provided, the name of the class shall be exactly the name of the file.
	The extension of the file name is never taken into account.
	By default, the name of the class shall be exactly the name of the file.
Parameters	An optional MINMAX couple of values giving the part of the file name to take into account.
Justification	Analysability.

Example:

```
if the MINMAX parameters are 4 and 10, and the file is
     My_graph_node.java
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 have 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 { ...}
```

nodeadcode No inaccesible code

Description	There shall be no dead code: i.e. statement located after *break, continue, return* and *exit* statements.
Justification	Maintainability.

packres Reserved Packages

Description	Some packages cannot be used in import statements or in the scope of identifiers.
	By default, the reserved packages are "java.awt" and "java.util.zip".
Parameters	A list of character strings representing reserved packages. These names may include dots.
Justification	Prevents from the import or the use of packages that are non portable or dangerous.

Example:

```
// if the java.rmi package is forbidden, do not write
import java.rmi.*;
import java.rmi.server.RemoteServer;

java.rmi.server.RemoteRef ref;
host = java.rmi.server.RemoteServer.getClientHost();
throw new java.rmi.ServerNotActiveException;
```

parse Parse Error

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

proxdecl Variable Declarations Close to the Use

Description	Variables must be declared as close as possible to their uses. Each local variable shall be declared in the block where it is used or in the smallest block containing the blocks where it is used. If a variable is used in a loop (`do`, `while`, `for`) or a multiple alternatives statement (`switch`) it can be declared in the enclosing block.
Note	Local variables that are declared but not used is a violation of the rule.
Justification	Maintainability.

Example:

```
// do not write
  int temp;
  String str;
...
  if (a > b) {
    temp = a;
    a = b;
    b = temp
  }

// write
  if (a > b) {
    int temp;
    temp = a;
    a = b;
    b = temp
  }
```

simplestmt Effective statement

Description	There shall not be a statement containing only the following operators or a cast cannot be a statement. >,<,>=, <=, ==, !=, &&, ||, true, false, not, <<, >>, &, |, +, -, /, *.
Justification	Reliability: such a statement is useless and may be a typing error.

Example:

```
// statements with no effect
x + 5;    // violation: may be a misspelling with x = 5;
x == y;   // violation: may be a misspelling with x = y;
```

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.
Parameters	The character string **"forinit"**, which, if used, specifies that the multiple declarations are allowed in `for` statements.

Justification	Makes the code easier to read.

Example:

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

// do not write
int width, length;

// with forinit option you can write
for (int i=0, j=0; i<len; i++, j++) { ...}
```

sglreturn A single return per function

Description	Only one return instruction is allowed in a function.
Justification	Maintainability : a basic rule for structured programming.

slstat A single statement per line

Description	There shall 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`). An empty block ({}) is not allowed on the same line as another statement.
	A line containing a label cannot contain another label or a statement.
	If an anonymous class appears inside a statement, its declarations shall be on different lines and shall not be on the same lines as the beginning or the end of the including statement.
Justification	Makes the code easier to read.

Example:

```
// write
x = x0;
y = y0;
while (IsOk(x)) {
   x++;
}
new_id = (new B{
  int f() {
    return value;
  }
}).id;

// do not write
x = x0; y = y0;
while (IsOk(x)) {x++;}
new_id = (new B {int f() {return value;}}).id;
```

swdef Default in switch

Description	A `switch` statement shall contain a `default` case. The `default` label shall be the last label.
	By default, the `default` case shall be the last label.
Parameters	The character string **"last"**, which, if used, specifies that the `default` case has to be the last one.
Justification	Fault Tolerance: All cases must be provided for in a `switch`.

swend End of cases in switch

Description	Each case in a `switch` shall end with `break`, `continue`, `return`, `System.exit()`, `Runtime.getRuntime().exit()` or `Thread.currentThread().stop()`. 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.
	•A character string beginning with **comment** and containing a regular expression, which, if used, allows to use a comment containing a string matching the regular expression to end a case.
Justification	Makes the code easier to understand and reduces the risk of errors.

Example:

```
//with the following parameters
//STANDARD swend "nolast" "comment CONTINUE WITH NEXT CASE"
//END STANDARD
//you can write
switch (x) {
case 1:
case 2:
    i++;
    break;
case 3:
    System.exit();
case 4:
    i += 2;
// CONTINUE WITH NEXT CASE because ...
case 5:
    return f(i);
default:
    i = 0;
}
```

unaryplus No unary plus operator

Description	The unary plus operator shall not be used.

Example:

```
x = +10; // violation
```

varinit Variable initialization

Description	Variables shall be initialized in their declarations.
Justification	Ensures correct variable initialization prior to use.

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\Java** 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.

brkcont Break and Continue Forbidden

By default, `break` statements are allowed in `switch` statements, `break` and `continue` are forbidden everywhere else:

```
STANDARD brkcont ON "in_switch" END STANDARD
```

To allow any `break` or `continue` statement without a label:

```
STANDARD brkcont ON "without_label" END STANDARD
```

To allow any `break` or `continue` statement with a label:

```
STANDARD brkcont ON "with_label" END STANDARD
```

const Literal Constants

By default, the allowed literal constants are **0**, **1**, **'\0'** and the character string literals:

```
STANDARD const ON LIST "0" "1" "'\0'" """*" END LIST END STANDARD
```

To allow the use of hexadecimal literals and character string literals:

```
STANDARD const ON LIST "0x*" """*" END LIST END STANDARD
```

constrdef Default Constructor

By default, a default constructor (without parameters) is required for each class:

```
STANDARD constrdef ON "default" END STANDARD
```

For each class to contain the explicit declaration of at least a constructor:

```
STANDARD constrdef ON END STANDARD
```

declord Declarations Order

By default, in a class, declarations must be in the following order: constructors, public class methods, public methods, public declarations and other declarations, and must end with private declarations:

```
STANDARD declord ON
LIST "constructor" "public method class" "public method" "public"
"others" "private" END LIST END STANDARD
```

dmaccess Access to Data Members

By default, only the data members definition in the private part of a class are authorized:

```
STANDARD dmaccess ON LIST "public" "protected" "package" END LIST END
STANDARD
```

To forbid the data members only in the public part of a class:

```
STANDARD dmaccess ON LIST "public" END LIST END STANDARD
```

exprcplx Expressions Complexity

By default, the maximum authorized complexity level is 10:

```
STANDARD exprcplx ON MINMAX 0 10 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 the rule to be stricter, remove this parameter:

```
STANDARD exprparenth ON END STANDARD
```

headercom Header Comments

It is possible to define a format for comments depending on the type of the item (**module**, **interface**, **class**, **method**, **attribute**).

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 author and the version is required for each class and interface:

```
STANDARD headercom ON
LIST "module"            "/\*"  END LIST
LIST "class"            "/\*\*" "@author" "@version" END LIST
LIST "interface"            "/\*\*" "@author" "@version" END LIST
LIST "attribute"            "/\*" END LIST
LIST "method"            "/\*" END LIST
END STANDARD
```

Example of required header for classes and interfaces:

```
/**
 *
 * @author  Andrieu
 * @version 1.3, 08/07/96
 */
```

identfmt Identifier Format

It is possible to define a format for each of the categories listed below:

NAME	DESCRIPTION	DEFAULT
package	package name	any
interface	interface name	any
interface-public	public interface name	interface, any
class	class name	any
class-public	public class name	class, any
class-abstract	abstract class name	class, any
class-abstract-public	public abstract class name	class-public, class-abstract, class, any
class-local	local class name	class, any
class-local-abstract	local abstract class name	class-local, class-abstract, class, any
method	method name	any
method-public	public method name	method, any
method-private	private method name	method, any
method-protected	protected method name	method, any
method-class	class method name	method, any
method-class-public	public class method name	method-class, method-public, method, any
method-class-private	private class method name	method-class, method-private, method, any
method-class-protected	protected class method name	method-class, method-protected, method, any
method-abstract	abstract method name	method, any
method-abstract-public	public abstract method name	method-abstract, method-public, method, any
method-abstract-private	private abstract method name	method-abstract, method-private, method, any
method-abstract-protected	protected abstract method name	method-abstract, method-protected, method, any
var	variable name	any
var-public	public variable name	var, any
var-private	private variable name	var, any
var-protected	protected variable name	var, any
var-class	class variable name	var, any
var-class-public	public class variable name	var-class, var-public, var, any
var-class-private	private class variable name	var-class, var-private, var, any

var-class-protected	protected class variable name	var-class, var-protected, var, any
var-local	local variable name	var, any
constant	constant name	var, any
constant-local	local constant name	constant, var-local, var, any
parameter	method parameter name	var-local, var, any
parameter-constant	constant method parameter name	parameter, constant-local, constant, var-local, var, any

The third column represents inherited categories: for instance, for no distinction between the **method-public**, the **method-private** and the **method-protected** categories, just define a particular format for the **method** 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 Header Comments*).

By default, the names of classes, interfaces and constants must begin with an uppercase letter and the names of packages and variables must begin with a lowercase letter:

```
STANDARD identfmt ON
LIST "any"                  ".*"
     "package"              "[a-z]*"
     "interface"            "[A-Z][A-Za-z0-9]*"
     "class"                "[A-Z][A-Za-z0-9]*"
     "constant"             "[A-Z][A-Z0-9_]*"
     "var"                  "[a-z][A-Za-z0-9]*"
     "var-local"            "[a-z][a-z0-9]*"
END LIST END STANDARD
```

For the class methods to begin with "m_", the constants to have no lower case letter and no underscore at the beginning and the end, the local variables to begin with "l_" and all other identifiers not to begin or end with an underscore:

```
STANDARD identfmt ON
LIST "any"                 "[^_](.*[^_])?$"
     "method"              "m_.*[^_]$"
     "const"               "[A-Z0-9]([A-Z0-9_]*[A-Z0-9])?$"
     "var-local"           "l_.*[^_]$"
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 packages, classes, interfaces, methods and global variables must have between 5 and 25 characters, the constants between 2 and 25, and the other identifiers between 1 and 25:

```
STANDARD identl ON
LIST "any"          MINMAX 1 25
     "package"      MINMAX 5 15
     "class"        MINMAX 5 25
     "interface"    MINMAX 5 25
```

```
     "method"       MINMAX 5 25
     "constant"     MINMAX 2 25
     "var"          MINMAX 5 25
     "var-local"    MINMAX 1 25
     "parameter"    MINMAX 1 25
END LIST END STANDARD
```

identres Reserved Identifiers

By default, the reserved identifiers are "byvalue", "cast", "const", "future", "generic", "goto", "inner", "operator", "outer", "rest" and "var":

```
STANDARD identres ON LIST "byvalue" "cast" "const" "future" "generic"
"goto" "inner" "operator" "outer" "rest" "var" END LIST END STANDARD
```

mname File Names

By default, the name of the file shall be exactly the name of the class:

```
STANDARD mname ON END STANDARD
```

For the part of the class name to be taken into account to be between the characters 1 and 5:

```
STANDARD mname ON MINMAX 1 5 END STANDARD
```

packres Reserved Packages

By default, the reserved packages are "java.awt" and "java.util.zip":

```
STANDARD packres ON LIST "java.awt" "java.util.zip" END LIST END STAN-
DARD
```

sgdecl A Single Variable per Declaration

By default, multiple declarations are allowed in `for` statements:

```
STANDARD sgdecl ON "forinit" END STANDARD
```

To forbid multiple declarations in all declarations:

```
STANDARD sgdecl ON 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 a `default` case, whatever its position:

```
STANDARD swdef ON END STANDARD
```

swend End of Cases in a "switch"

By default, an instruction `break`, `continue`, `return`, `System.exit()`, `Runtime.getRuntime().exit()` or `Thread.currentThread().stop()` 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` including the last one:

```
STANDARD swend ON 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 public data members
.DESCRIPTION
There shall be no public attributes.
.COMMAND rename dmaccess
```

Rule Script Location

The rule script should be placed in one of the following places in the **Rules\Java** 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
```

wher:e

<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

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.