Rational. Logiscope
Release 6.6

IBM

RuleChecker & QualityChecker
C Reference Manual

IBM Rational Logiscope

RuleChecker & QualityChecker C Reference Manual

Before using this information, be sure to read the general information under “Notices” section, on page **151**.

This edition applies to **VERSION** ***6.6*****,** ***IBM Rational LOGISCOPE*** **(product number** ***5724V81)*** and to all subsequent releases and modifications until otherwise indicated in new editions.

About This Manual

Audience

This manual is intended for IBM® Rational® Logiscope™ *RuleChecker & QualityChecker* users for C source code verification.

Related Documents

Reading first the following manuals is highly recommended:

- *IBM Rational Logiscope - Basic Concepts*.
- *IBM Rational Logiscope - RuleChecker & QualityChecker - Getting Started*.

Additional information on how to write new C rule verification scripts can be found in:

- *IBM Rational Logiscope - Writing C rule using RuleChecker Tcl Verifier.*

Overview

C Project Settings

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

C Parsing Options

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

Command Line Mode

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

Standard Metrics

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

Standard Programming Rules

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

Customizing Standard Rules

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

Developing New Rule Scripts

Chapter 7 provides some basics to write new rule verification scripts to be run by Logiscope *RuleChecker* C.

Logiscope C Data Model

Chapter 8 specifies the C Data Model used by Logiscope Logiscope *RuleChecker* C to locate and report programming rules violations in the source code under analysis.

Conventions

The following typographical conventions are used:

bold	literals such as tool names (**studio**) and file extensions (***.c**),
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 IBM Rational Software Support

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

Note If you are a heritage Telelogic customer, you can go to http://support.telelogic.com/toolbar and download the IBM Rational Telelogic Software Support browser toolbar. This toolbar helps simplify the transition to the IBM Rational Telelogic product online resources. Also, a single reference site for all IBM Rational Telelogic support resources is located at http://www.ibm.com/software/rational/support/telelogic/

Prerequisites

To submit your problem to IBM Rational Software Support, you must have an active Passport Advantage® software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online in Passport Advantage from http://www.ibm.com/software/lotus/passportadvantage/howtoenroll.html

- To learn more about Passport Advantage, visit the Passport Advantage FAQs at http://www.ibm.com/software/lotus/passportadvantage/ brochures_faqs _quickguides. html.
- For further assisance, contact your IBM representative

To submit your problem online (from the IBM Web site) to IBM Rational Software Support, you must additionally:

- Be a registered user on the IBM Rational Software Support Web site. For details about registering, go to http://www-01.ibm.com/software/support/..
- Be listed as an authorized caller in the service request tool

Submitting problems

To submit your problem to IBM Rational Software Support:

1. Determine the business impact of your problem. When you report a problem to IBM, 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
1	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.
2	The problem has a *significantl* business impact. The program is usable, but it is severely limited.
3	The problem has a *some* business impact.The program is usable, but less significant features (not critical to operation) are unavailable.
4	The problem has a *minimal* business impact.The problem causes little impact on operations or a reasonnable circumvention to the problem was implemented.

2. Describe your problem and gather background information, When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Rational 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, use the option applicable to you:

  - Start the IBM Installation Manager and select **File** > **View Installed Packages**. Expand a package group and select a package to see the package name and version number.
  - 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 IBM Rational Software Support. You can submit your problem to IBM Rational Software Support in the following ways:
   - **Online**: Go to the IBM Rational Software Support Web site at https://www.ibm.com/software/rational/support/ and in the Rational support task navigator, click **Open Service Request.** Select the electronic problem reporting tool, and open a Problem Management Record (PMR), describing the problem accurately in your own words.

     For more information about opening a service request, go to http://www.ibm.com/software/support/help.html

     You can also open an online service request using the IBM Support Assistant. For more information, go to http://www-01.ibm.com/software/support/isa/faq.html.
   - **By phone**: For the phone number to call in your country or region, go to the IBM directory of worldwide contacts at http://www.ibm.com/planetwide/ and click the name of your country or geographic region.
   - **Through your IBM Representative**: If you cannot access IBM Rational Software Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. You can find complete contact information for each country at http://www.ibm.com/planetwide/.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Rational Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Rational Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the IBM Rational Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.

Table of Contents

Chapter 1

C Project Settings

A Logiscope project mainly consists in:

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

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

Logiscope C projects can be created using:

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

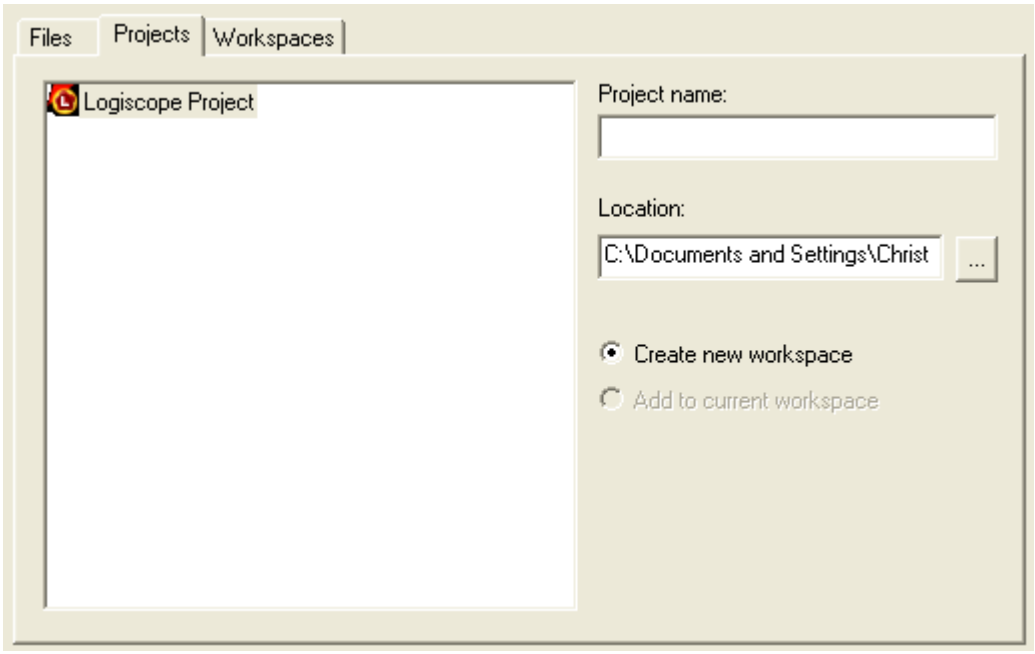
1.1 Starting a Logiscope Studio Session

To begin a Logiscope **Studio** session:

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

1.2 Creating a Logiscope Project

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



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

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

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

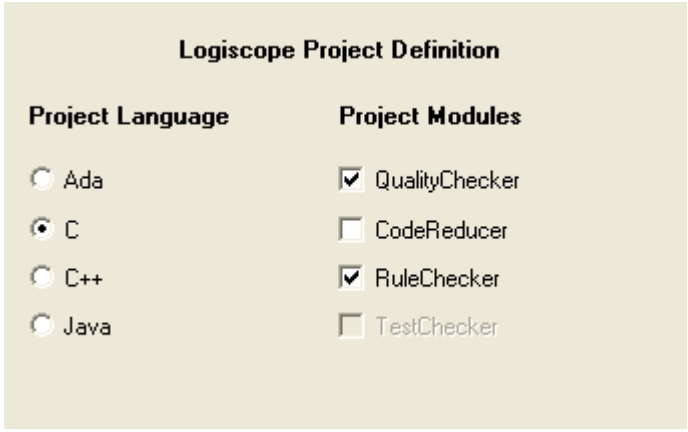
Defining the type of the Logiscope project

The **Logiscope Project Definition** dialog box allows to specify the type of Logiscope projects to be created.

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

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

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



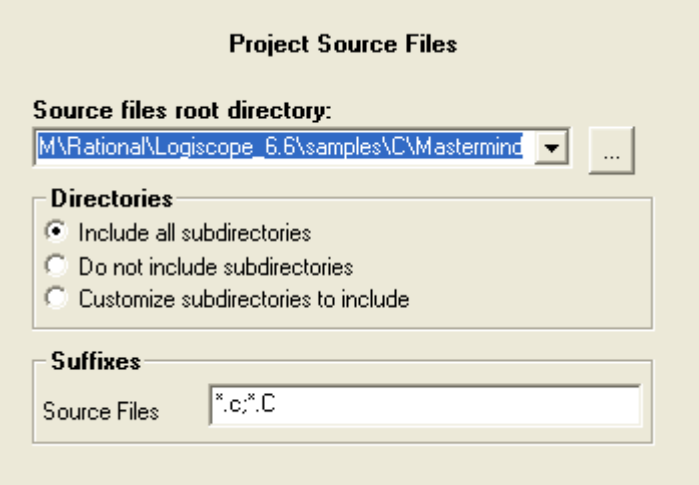
Notes: At least one module should be selected. The *TestChecker* module cannot be selected with an other module.

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

Specifying the source files to be analysed

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

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



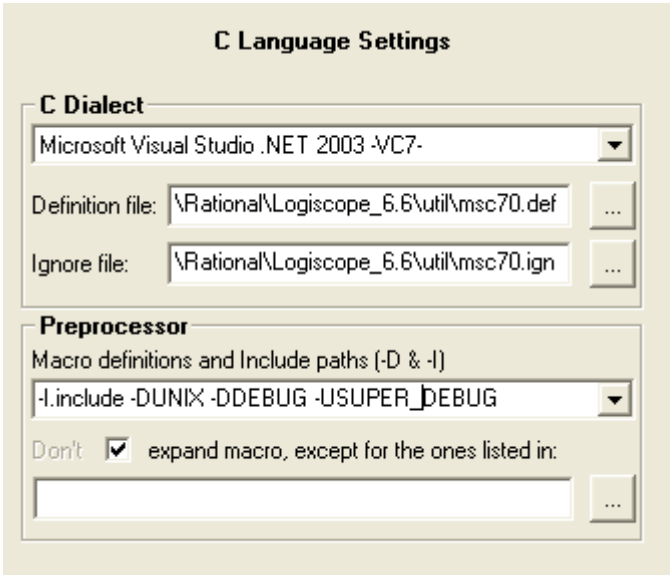
If necessary, use the **Directories** choices to select the list of subdirectories covering the application source files.

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

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

Setting Parsing Options

The **C Language Settings** dialog box allows to set up C source code parsing options:



C Dialect:

A dialect is used to specify some default specifics of the C development environment (e.g. compilers, IDE) in use for the project under analysis:

- access paths to standard inclusion directories,
- predefined macro definitions.
- inclusion directories where rule violations shall not be reported.

In case the proposed C dialects do not match the specifics of the project C development environment, the user can provide a dedicated **Definition file** specifying preprocessor macro definitions and include files paths applicable to the project.

The source code files composing a Logiscope project may contain portions of code that are not written in C (SQL commands, assembler language etc.). To avoid parsing errors or inappropriate counting, the user can provide a dedicated **Ignore file** specifying the syntax of the portions of code to be ignored when parsing the source files.

Please refer to the next chapter *C Parsing Options* for more details on the supported C dialects and the associated Definition file and Ignore file.

Preprocessor

In addition to the predefined preprocessing information associated to the selected C dialect, the user can use the **Preprocessor** pane to provide complementary preprocessing and compilation options:

- access paths to project specific inclusion directories,
- project macro definitions.

The syntax is as for a C compiler:

[-I*directory***]***

[-D*name_of_macro1_with_no_argument* **[**=*definition***]]***

[-U*name_of_macro2_with_no_argument* **[**=*definition***]]***

The number of occurrences of options **-I, -D**, **-U** is unlimited.

A "**-I**" option defines *directory* as access paths to inclusion directories.

A "**-D**" option defines *name_of_macro1_with_no_argument* as if it were in a `#define` directive.

A "**-U**" option considers *name_of_macro2_with_no_argument* as undefined as if it were part of an `#undef` directive.

In the example below:

```
-I./include -DUNIX -DDEBUG -USUPER_DEBUG
```

- Logiscope C parser will search for include files in the sub directory `./include`;
- the `UNIX` and `DEBUG` option are defined, so the corresponding conditional code will be parsed;
- the `SUPER_DEBUG` option is considered as undefined so the corresponding conditional options will not be parsed.

Note: The option `-nowarning` allows to turn off Logiscope warning messages when parsing C files.

Expanding or not expanding macros

By default, macros are expanded by the Logiscope C parser unless other macro processing modes are specified (non expansion, expansion of a subset of macros).

Macro expansion makes it possible to take into account the control structure and the textual elements of a macro. In this way, the constitutive elements of the macro will appear on the control graphs displayed by Logiscope *Viewer*.

Once the macros are expanded, the code is syntactically correct and thus analyzable. This is not guaranteed with no expansion or partial expansion.

If the expansion is partial or absent, the Logiscope C parser will consider:

- non-expanded macros with arguments as functions,
- those with no arguments as identifiers.

Those which are considered as functions will appear on the control graph displayed by Logiscope *Viewer*.

The reason for not expanding macros is to avoid result overload.

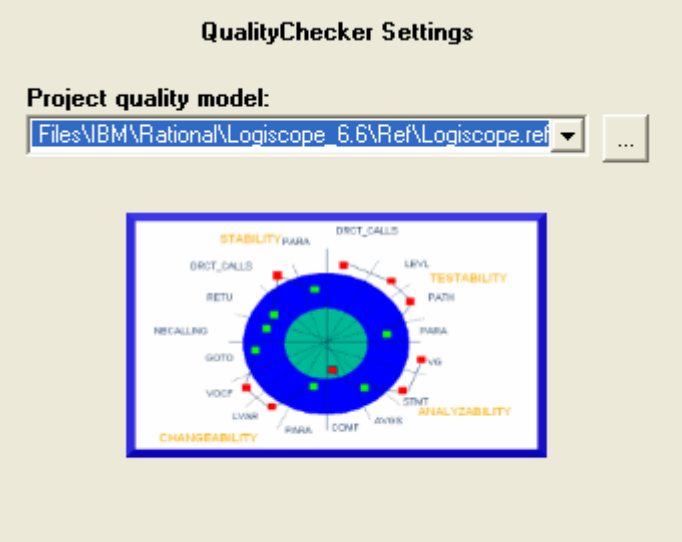
It is possible to invert the macro processing mode for the macros listed in the file specified in the last pane of the **C Language Settings** dialog box. For example, if the macro expansion is requested, the macros in the specified file will not be expanded and others will be. The file should contain a list of macro names (one per line).

Setting QualityChecker Parameters

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

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

For more information, see the *IBM Rational Logiscope Basic Concepts* manual.



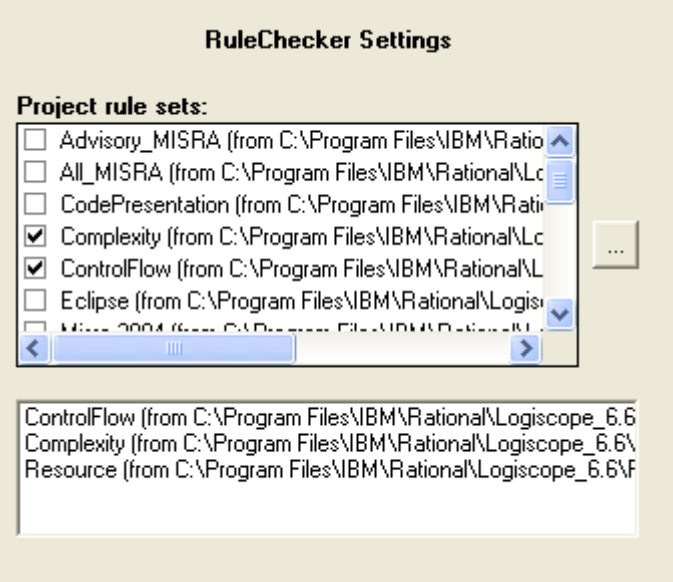
For your project verification, you should define and select your own applicable quality model.

Setting RuleChecker Parameters

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

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

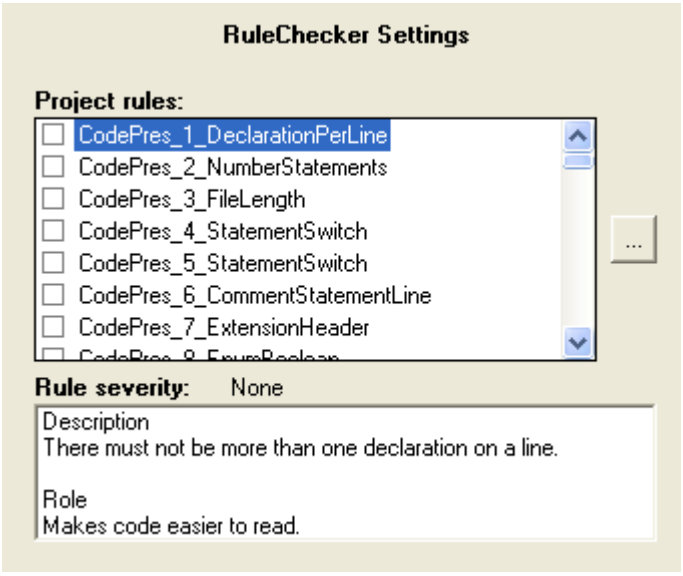
Several rule sets can be selected. If so, Logiscope *RuleChecker* will check the union of the rules specified in all selected rule sets.



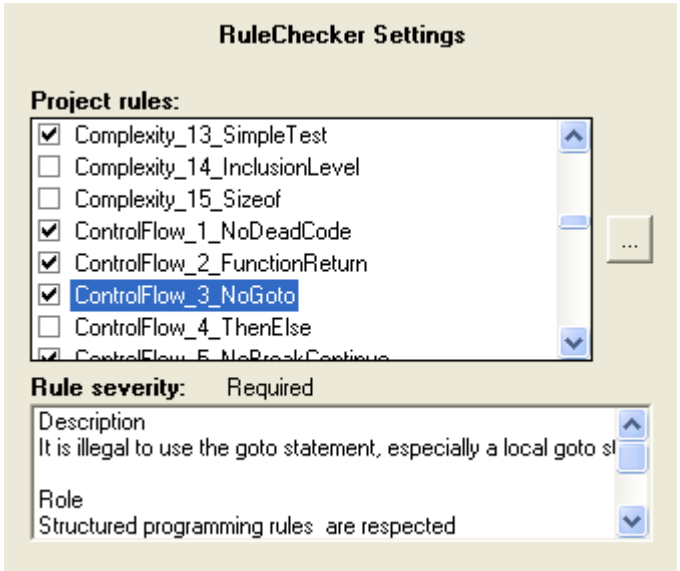
For more details on available rules and rule sets, please refer to the chapter *Standard Programming Rules.*

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

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

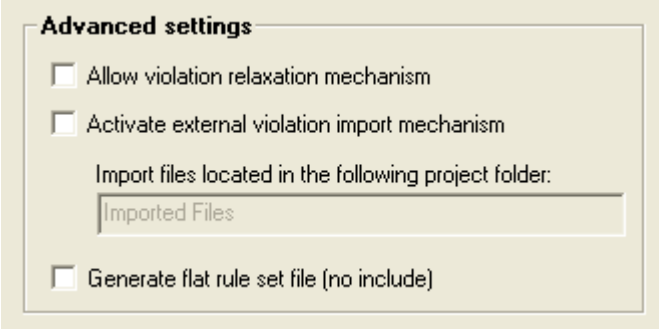


You can check / uncheck the rules.The description of the selected rule and the rule severity are displayed in the bottom pane



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

Advanced Settings:

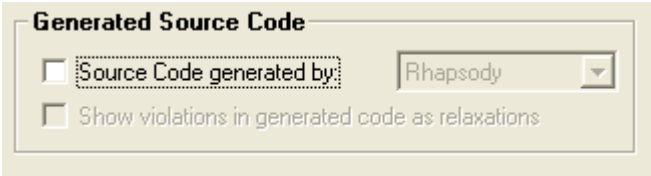


Allow violation relaxation mechanism: when the box is checked, rule violations can be relaxed using special comments in the code. For more details, please refer to *IBM Rational Logiscope - Basic Concepts* document..

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

Generate flat rule set file (no include): when the box is checked, the project rule set file (i.e. with a ".rst") extension) that is generated for the project doesn't contain any includes of other rule set files. It will contain an expanded copy of the contents of any rule sets that were used for the project.
For more details, please refer to the Chapter *Customizing Rules and Rule Sets.*

Generated Source Code:



Source Code generated by: when the box is checked, allows to specify the tool (e.g. Rhapsody) used to generate the code under analysis. Thus, *Logiscope RuleChecker* will not considered the violations found in the generated code.

Show violations in generated code as relaxations: when the box is checked, the violations found in generated code are reported as "relaxations". For more details, please refer to the next section.

1.3 Relaxation Mechanism

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

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

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

Relaxing a single rule violation

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

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

where:

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

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

Relaxing several violations and/or adding a longer justification

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

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

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

Relaxing all violations in pieces of code

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

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

where:

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

Chapter 2

C Parsing Options

2.1 Dialects

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

In order to extract accurate information from the source code under analysis, the Logiscope C parser behaves as a C compiler. Therefore, all information requested for correct preprocessor operation shall be provided to the Logiscope C parser to correctly translate all C units available in the code.

For instance, expanding a macro definition involves during the code analysis, substitution of each macro occurrence by its definition.

The C unit translation is impacted by:

- some default specifics of the C development environments (e.g. compilers, IDE) in use for the project under analysis :
  - access paths to standard inclusion directories,
  - predefined macro definitions,
- project specific preprocessor macro definitions and include file paths.

Once the macro definitions are expanded, the code is syntactically correct and thus analyzable. This is not guaranteed with no expansion or partial expansion.

To consider those specifics when parsing the source code and thus avoid parsing errors and warnings, the user shall select the appropriate C dialect when setting up the Logiscope project (see previous chapter).

The C dialects supported by Logiscope C are listed in section 2.4.

In fact, each C dialect is associated to predefined configuration files for parsing:

- the Definition file : that specifies access paths to standard inclusion directories and predefined macro definitions,
- the Ignore File that allows to ignore non C code ((e.g; SQL commands, assembler language) during parsing.

These two types of configuration file are respectively detailed in section 2.2 and 2.3.

These files can be modified to match the specifics of the C development environments (e.g. compilers, IDE) in use for the project under analysis.

In case of a C dialect not supported by Logiscope, the user can define dedicated Definition files and, if applicable, Ignore files. The syntax of these user specified parsing configuration files shall follow the same syntax of the dialect file specified in the next sections.

2.2 Definition File

For correct and accurate preprocessing operation, the Definition file shall contain:

- the access paths to inclusion directories,
- the list of the predefined macro definitions.

The list of predefined macro definitions for a given compiler is usually provided in the reference manual of the compiler. Compiling code using the "-v" option may also be used to know it.

Since these items are machine/environment configuration dependent (e.g. access path to the system include files), it may be necessary to adapt the Definition file associated to a given dialect or to create a new Definition file.

In case of a user specified Definition file, it shall be provided to Logiscope C:

- using the **Project - Settings ...** command of **Logiscope Studio** once the Logiscope project has been created,
- using the "-ddef" option of the **Logiscope Create** tool.

Syntax: The Definition file syntax is as follows:

[I<*directory***>]***

[D<*macro_with_no_argument***> [=***definition***]]***

[U<*macro_with_no_argument***> [=***definition***]]***

[E<*directory***>]***

A "**I**" option defines *directory* as access paths to inclusion directories.

A "**D**" option defines *macro1_with_no_argument* as if it were in a `#define` directive.

A "**U**" option considers *macro2_with_no_argument* as undefined as if it were part of an `#undef` directive.

A "**E**" option allows to hide the rules violations in source files located in *directory*.

Example:

On Windows, to analyze **Microsoft Visual Studio .NET 2003** C code, Logiscope will read the information predefined in the *msc70.def* Definition file.

The content of this file located by default in the <*log_install_dir*>/**util** directory is listed below:

```
I.
IC:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\INCLUDE
IC:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\atlmfc\INCLUDE
D_M_IX86=600
D_MSC_VER=1310
D_WIN32
D__STDC__
D_INTEGRAL_MAX_BITS=64
```

In this example, "`C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\INCLUDE`" corresponds to the name of the standard include directory and `_M_IX86` is the name of a compiler predefined macro.

Note:

If Microsoft Visual Studio is installed on another drive than C:, change access paths in the Definition file**.**

The Definition file will be sought in the following sequence:

1 from the access file indicated in the `LOG_CC_DEF` environment variable,

2 from the Logiscope startup directory,

3 from the directory indicated in the `LOG_UTIL` environment variable.

2.3 Ignore File

The source code files composing a Logiscope project may contain portions of code that are not written in C (e.g. SQL commands, assembler language).

To ignore these portions of code during C source code parsing, just define the sequences of code that delimit the portions of code to be ignored and place them in a text file (suggested extension **.ign**).

Examples of such a file are provided in the <*log_install_dir*>/**util** directory.

The syntax of the Ignore file defining the code to be ignored is as follows:

- To ignore a portion of code between two keyword sequences:
  word1 word2 ... wordn --> word1' word2' ... wordm'
  Example:
  `SQL BEGIN --> SQL END`
  Code between `SQL BEGIN` and `SQL END` is ignored.
- To ignore a portion of code between a keyword sequence and the end of the line:
  word1 word2 ... wordn --> $
  Examples:
  `_asm --> $`
  The portion of code between _asm and the end of the line is ignored.
  `# pragma --> # pragma end`
  (Please note the spaces between # and pragma)
  The portion of code between #pragma and #pragma end is ignored.
- To ignore a keyword sequence:
  word1 word2 ... wordn - ->
  Example:
  `user input -->`
  The keyword sequence `user input` is ignored.

Note:

A portion of code starting with the same keyword as another portion of code and whose left sequence is a subsequence of the portion is prohibited.

Example:

```
m1 m2 m3 m4 --> x y z
m1 m2 --> $
```

2.4 Supported C Dialects Specification

The current list of available C dialects is the following:

- **ANSI 89 / ISO 90**
- **ANSI / ISO 99**
- **DIAB C**
- **GNU C**
- **GNU C D950**
- **GNU C Red Hat Linux 3**
- **GNU C Red Hat Linux 4**
- **GNU C Red Hat Linux 5**
- **HP C**
- **IAR C**
- **Kernighan and Ritchie 78**
- **Microsoft C 1.5**
- **Microsoft Developer Studio 4**
- **Microsoft Developer Studio 5**
- **Microsoft Visual Studio 6 -VC98-**
- **Microsoft Visual Studio .NET 2003 -VC7-**
- **Microtec Reseach C**
- **Microtec Reseach C ANSI**
- **SUN C**

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

2.4.1 ANSI 89 / ISO 90

Definition Files

ansi.def file on Windows

.**log_cc_***sun4os5*_**ansi.def** on UNIX

.**log_cc_***linux*_**ansi.def** on Linux

Reference Documentation

ISO / IEC 9899
Programming languages - C
ISO / IEC 9899 : 1990 (E)

2.4.2 ANSI / ISO 99

Definition Files

iso99.def file on Windows

.log_cc_*sun4os5*_iso99.def on UNIX

.log_cc_*linux*_iso99.def on Linux

Reference Documentation

ISO / IEC 9899
Programming languages - C
ISO / IEC 9899 : 1999 (E)

2.4.3 DIAB C

Definition Files

diab.def file on Windows

.log_cc_*sun4os5*_diab.def on UNIX

.log_cc_*linux*_diab.def on Linux

Ignore File

- **diab.ign**

The __asm { *text* } and __asm *text_until_end_of_line* instructions are ignored.

Reference Documentation

D-CC™ & D-C++™ Compiler Suites
NEC V800 Series Family User's Guide and Getting Started Version 4.4

Language Specifics

The macros PPC and __DIAB are recognized.

2.4.4 GNU C

Definition Files

gnu.def file on Windows

.log_cc_*sun4os5*_gnu.def on UNIX

.log_cc_*linux*_gnu.def on Linux

Reference Documentation

GNU C Compiler - ST9 Family - User Manual
SGS-THOMSON Microelectronics
Release 3.0
May 1993

Preprocessor Specifics

The #pragma directives are not interpreted by the analyzer.

Language Specifics

The following keywords are recognized:

- asm, __asm__
- typeof, __typeof__
- inline, __inline__
- __alignof__
- __signed__
- __const__
- __volatile__

2.4.5 GNU C D950

Definition Files

gnu_d950.def file on Windows

.log_cc_*sun4os5*_gnu_d950.def on UNIX

.log_cc_*linux*_gnu_d950.def on Linux

Ignore File

- **gnu_D950.ign**

Reference Documentation

GNU C Compiler - D950 Family of DSP Processors
SGS-THOMSON Microelectronics
Release 1.1
January 1995

Preprocessor Specifics

The #pragma directives are not interpreted by the analyzer.

Language Specifics

The following keywords are recognized:

- asm, __asm__
- typeof, __typeof__
- inline, __inline__
- __alignof__
- __signed__
- __const__
- __volatile__
- __space__

2.4.6 GNU C Red Hat Linux 3

Definition Files

gnu_rhel_3.def file on Windows

.log_cc_*sun4os5*_gnu_rhel_3.def on UNIX

.log_cc_*linux*_gnu_rhel_3.def on Linux

Ignore Files

Gnu_Rhel_3.ign file on Windows

.log_cc_*sun4os5*_gnu_rhel_3.ign on UNIX

.log_cc_*linux*_gnu_rhel_3.ign on Linux

Reference Documentation

GNU C 3.2.3 Manual

2.4.7 GNU C Red Hat Linux 4

Definition Files

gnu_rhel_4.def file on Windows

.log_cc_*sun4os5*_gnu_rhel_4.def on UNIX

.log_cc_*linux*_gnu_rhel_4.def on Linux

Ignore Files

Gnu_Rhel_4.ign file on Windows

.log_cc_*sun4os5*_gnu_rhel_4.ign on UNIX

.log_cc_*linux*_gnu_rhel_4.ign on Linux

Reference Documentation

GNU C 3.4.4 Manual

2.4.8 GNU C Red Hat Linux 5

Definition Files

gnu_rhel_5.def file on Windows

.log_cc_*sun4os5*_gnu_rhel_5.def on UNIX

.log_cc_*linux*_gnu_rhel_5.def on Linux

Ignore Files

Gnu_Rhel_5.ign file on Windows

.log_cc_*sun4os5*_gnu_rhel_5.ign on UNIX

.log_cc_*linux*_gnu_rhel_5.ign on Linux

Reference Documentation

GNU C 4.1 Manual

2.4.9 HP C

Definition Files

hp.def file on Windows

.log_cc_*sun4os5*_hp.def on UNIX

.log_cc_*linux*_hp.def on Linux

Reference Documentation

HP C / HP-UX Reference Manual (Hp 9000 Series 800 Computers)
Hewlett Packard
First Edition
August 1989

The list of predefined macro definitions can be obtained by compiling a file with the **-v** option of the HP C compiler.

2.4.10 IAR C

Definition Files

iar.def file on Windows

.log_cc_*sun4os5*_iar.def on UNIX

.log_cc_*linux*_iar.def on Linux

Reference Documentation

IAR C COMPILER FOR THE H8/300 SERIES

Fourth Edition: January 1995

Part Number: ICCH83-4

Language Specifics

The following keywords are recognized:

- ANSI_main,
- banked_func, non_banked, banked
- C_task
- far, far_func
- huge
- near, near_func
- no_init
- tiny, tiny_func
- version_2
- zpage
- monitor
- interrupt
- ccr_mask
- bit
- sfr, sfrp

The following macros are recognized:

- __STDC__ 0
- __IAR_SYSTEMS_ICC__
- __ON_SIZEOF_NOT_SUPPORTED__ 4
- _argt$(a) 1
- _arg$ "1"
- __TID__ 1

2.4.11 Kernighan and Ritchie 78

Definition Files

kr78.def file on Windows

.log_cc_*sun4os5*_kr78.def on UNIX

.log_cc_*linux*_kr78.def on Linux

Reference Documentation

The C Programming Language
Kernighan and Ritchie
Prentice Hall Software Series 78

2.4.12Microsoft C 1.5

Definition Files

msc15.def on Windows

.log_cc_*sun4os5*_microsoft_15.def on UNIX.

.log_cc_*linux*_microsoft_15.def on UNIX.

Ignore File

- **msc15.ign**

Reference Documentation

Extract related to C MICROSOFT 1.5 language of the CD-ROM
Microsoft Visual C++
Development System and Tools for Windows

Language Specifics

The following keywords are recognized, ignored and copied in the instrumented source code:

- __based, _based
- __cdecl, _cdecl, cdecl
- __export, _export
- __far, _far, far
- __fastcall, _fastcall
- __fortran, _fortran
- __huge, _huge, huge
- __inline, _inline
- __interrupt, _interrupt
- __loadds, _loadds
- __near, _near, near
- __pascal, _pascal
- __saveregs, _saveregs
- __segment, _segment
- __segname, _segname

The __asm (or _asm) instruction is recognized in different forms but not in cases listed with the following limitations header.

Limitations

- The __asm { text } instruction is recognized if character ”}” does not appear in text (nor in comments).
- The #@ (Charizing Operator) preprocessor operator is not accepted.
- The (:>) base operator is not recognized.

2.4.13Microsoft Developer / Visual Studio

Definition Files

On Windows:

- **msc40.def** for **Microsoft Developer Studio 4.X,**
- **msc50.def** for **Microsoft Developer Studio 5.0,**
- **msc60.def** for **Microsoft Visual Studio 6.0 -VC98,**
- **msc70.def** for **Microsoft Visual Studio .NET 2003 -VC7-,**

On UNIX:

- **.log_cc_*sun4os5*_microsoft_20.def** for **Microsoft Developer Studio 4.X,**
- **.log_cc_*sun4os5*_microsoft_50.def** for **Microsoft Developer Studio 5.0,**
- **.log_cc_*sun4os5*_microsoft_60.def** for **Microsoft Visual Studio 6.0 -VC98**,
- **.log_cc_*sun4os5*_microsoft_70.def** for **Microsoft Visual Studio .NET 2003 -VC7-**,

On Linux:

- **.log_cc_*linux*_microsoft_20.def** for **Microsoft Developer Studio 4.X,**
- **.log_cc_*linux*_microsoft_50.def** for **Microsoft Developer Studio 5.0,**
- **.log_cc_*linux*_microsoft_60.def** for **Microsoft Visual Studio 6.0 -VC98**,
- **.log_cc_*linux*_microsoft_70.def** for **Microsoft Visual Studio .NET 2003 -VC7-**,

Ignore Files

- **msc40.ign** for **Microsoft Developer Studio 4.X,**
- **msc50.ign** for **Microsoft Developer Studio 5.0,**
- **msc60.ign** for **Microsoft Visual Studio 6.0 -VC98,**
- **msc70.ign** for **Microsoft Visual Studio .NET 2003 -VC7-,**

Reference Documentation

Extract on the CD-ROM C MICROSOFT 2.0 language
Microsoft Visual C++
Development System and Tools for Windows

Language Specifics

The following keywords are recognized but ignored:

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

The __asm (or _asm) instruction is recognized in different forms but not in cases listed with the following limitations header.

Limitations

- The __asm { text } instruction is recognized if character ”}” does not appear in text (nor in comments).
- The #@ (Charizing Operator) preprocessor operator is not accepted.

2.4.14Microtec Research C

Definition Files for Standard Mode

mcc_std.def file on Windows

.log_cc_*sun4os5*_mcc_std.def on UNIX

.log_cc_*linux*_mcc_std.def on Linux

Definition Files for ANSI Mode

mcc.def file on Windows

.log_cc_*sun4os5*_mcc.def on UNIX

.log_cc_*linux*_mcc.def on Linux

Reference Documentation

MCC68K C Compiler
Microtec Research Inc.
Version 4.4 - December 1993

The list of compiler specifics can be obtained by compiling a file containing the #pragma macro directive.

Language Specifics (Standard and ANSI Modes)

The following keywords are recognized but ignored:

- interrupt
- packed
- unpacked
- typeof

The asm pseudo function is recognized.

Preprocessor Specifics (Standard and ANSI Modes)

The following directives are recognized but ignored:

- #info, #inform, #informing
- #pragma eject, #pragma error, #pragma info, #pragma list, #pragma macro, #pragma option, #pragma warn
- #warn, #warning

The following directives are recognized and the portions of code found between the two directives are ignored:

#pragma asm, #pragma endasm

2.4.15SUN C

Definition Files

sun.def file on Windows

.log_cc_*sun4os5*_**sun.def** on UNIX

.log_cc_*linux*_**sun.def** on Linux

Reference Documentation

The C Programming Language - Kernighan and Ritchie
Prentice Hall Software Series 78

Language Specifics

The $ character is authorized in identifiers.

Chapter 3

Command Line Mode

3.1 Logiscope create

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

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

3.1.1 Command Line Mode

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

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

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

Automatic search

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

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

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

File list

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

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

```
/users/logiscope/samples/C/mstrmind/master.c
/users/logiscope/samples/C/mstrmind/player.c
/users/logiscope/samples/C/mstrmind/machine.c
```

Using the command line:

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

will create a new Logiscope C project file named aProject.ttp containing 3 files: master.c, player.c and machine.c on which *RuleChecker* and *QualityChecker* verification modules will be activated.

3.1.2 Makefile mode

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

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

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

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

-- : which introduces the compiler command.

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

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

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

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

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

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

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

3.1.3 Options

The **create** options are the following:

`create -lang c`

`<ttp_file>`	name of a Logiscope project to be created (with the .ttp extension). Path has to be absolute if the option -- is used.
`[-root <directory>]`	where <directory> is the starting point of the source search. Default is the current directory. This option is exclusive with -list option.
`[-recurse]`	if present the source file search is done recursively in subfolders.
`[-list <list_file>]`	where <list_file> is the name of a file containing the list of filenames to add to the project (one file per line). This option is exclusive with -root option.
`[-repository <directory>]`	where <directory> is the name of the directory where Logiscope internal files will be stored.
`[-no_compilation]`	avoid compiling the files if the -- option is used
`[--]`	when used in a makefile, 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/C/ 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.
`[-source <suffixes>]`	where <suffixes> is the list of accepted suffixes for the source files. Default is "*.c;*.C".
`[-dial <dialect_name>]`	where <dialect_name> is one of the available C dialects.
`[-def <definition_file>]`	where <definition_file> is a definition file (.def) containing include paths and macro definitions.
`[-ign <ignore_file>]`	where <ignore_file> is an ignore file (.ign) specifying code to be ignored during parsing.
`[-I<include_path>]*`	same syntax as a preprocessor. Only if option -- is not used.
`[-D<macro_name>]*`	same syntax as a preprocessor. Only if option -- is not used.
`[-U<macro_name>]*`	same syntax as a preprocessor. Only if option -- is not used.
`[-mode=exp|noexp]`	to specify the mode of preprocessing of the macros statements. Default is `exp`: macros are expanded.
`[-mac <macro_file>]`	where <macro_file> is a text file specifying a list of macros statements to be or not to be expanded according to the value of the -mode option.

3.2 Logiscope batch

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

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

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

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

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

3.2.1 Options

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

`batch`

`<ttp_file>`	name of a Logiscope project.
`[-tcl <tcl_file>]`	name of a **Tcl** script to be used to generate the reports instead of the default **Tcl** scripts.
`[-o <output_directory>]`	directory where the all reports are generated.
`[-external <violation_file>]*`	name of the file to be added into the import project folder. This option can be repeated as many times as needed. This option is only significant for *RuleChecker* module for which the external violation importation mechanism is activated
`[-nobuild]`	generate 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 <addin> options]`	where `addin` nis the name of the addin to be activated and `options` the associated options generating the reports.

`[-table]`	generate 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]`	generate reports with no left frame.
`[-v]`	display the version of the **batch** tool.
`[-h]`	display help and options for **batch**.
`[-err <log_err_folder>]`	directory where troubleshooting files **batch.err** and **batch.out** should be put. By default, messages are directed to standard output and error.

3.2.2 Examples of Use

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

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

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

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

```
batch MyProject.ttp
```

will:

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

Chapter 4

Standard Metrics

Logiscope QualityChecker C proposes a set of standard source code metrics. Source code metrics are static measurements (i.e. obtained without executing the program) to be used to assess attributes (e.g. complexity, self-descriptiveness) or characteristics (e.g. Maintainability, Reliability) of the C source code 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 standard 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 C functions defined in the source files specified in the Logiscope Project under analysis.
- The *Module scope*: the metrics are available for each C source files specified in the Logiscope Project under analysis; header files (i.e. suffixed by “.h” and referenced in #include preprocessor directives) are not considered.
- The *Application scope*: the metrics are available for the set of C source files specified in the Logiscope Project .

4.1 Function Scope

4.1.1 Line Counting

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

- *IBM Rational Logiscope - Basic Concepts*.

lc_cline Total number of lines

Definition Total number of lines within the function.

lc_cloc Number of lines of code

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

lc_cblank Number of empty lines

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

lc_comm Number of lines of comments and header

Definition Number of lines of comment s
- between the function header and the closing curly bracket of the previous function) and,
- within the function.

Alias LCOM

lc_ccomm Number of lines of comments

Definition Number of lines of comments within the function.

lc_csbra Number of lines with lone braces

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

lc_ccpp Number of preprocessor statements

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

lc_stat Number of statements

Definition Number of executable statements in the function.

The following are statements:

IF
[ELSE]
SWITCH
WHILE
DO
FOR
GOTO
BREAK
CONTINUE
RETURN
THROW
TRY
ASM
; (empty statement)
expression; (simple statement)

Statements located in external declarations are not taken into account.

Alias STMT

lc_bcob Number of comments blocks before

Definition 1 if at least a comment is located between the function header and the closing curly bracket of the previous function or between the function header and the beginning of the file.
0 if not.

Example

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

lc_bcob = 1

Alias BCOB

lc_bcom Number of comments blocks

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

Example

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

lc_bcom value = 3

Alias BCOM

CCOM Number of characters in the comments

Definition Number of alphanumeric characters in comments located between the function header and the closing curly bracket.

CCOB Number of characters in the comments before

Definition Number of alphanumeric characters in comments located between the function's header and the closing curly bracket of the previous function or between the function's header and the beginning of the file

LCOB Number of lines of comments before

Definition Number of comments lines located between the function header and the closing curly bracket of the previous function or between the function header and the beginning of the file.

4.1.2 Data Flow

dc_lvars Number of local variables

Definition Number of local variables declared in the function.
Alias LVAR

ic_param Number of parameters

Definition Number of formal parameters of the function.
Alias PARA

UPRO Number of functions used but not yet defined

Definition Number of functions with an unknown prototype used in the function.

MACC Number of macros used as constants

Definition Number of macro-instructions used as constants in the function.

MACP Number of macros with parameters

Definition Number of macro-instructions with parameters used in the function.

4.1.3 Halstead Metrics

For more details on Halstead Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts*.

n1 Number of distinct operators

Definition Number of different operators between the function's header and its closing curly bracket.

Alias ha_dopt

The following are C operators:

- Expressions:
  - n Unary operators:

+ -	unary plus or minus
++ --	pre-/post- increment or decrement
!	negation
~	complement of 1
*	indirection
&	address
sizeof	sizeof
.	dot
->	arrow
()	expression in parenthesis

  - n Binary Operators:

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

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

(...)	cast	(ex: **(**float**)**1)
dynamic_cast	cast	(ex: **dynamic_cast**<T>(v))
static_cast	cast	(ex: **static_cast**<T>(v))
reinterpret_cast	cast	(ex: **reinterpret_cast**<T>(v))

const_cast	cast	(ex: **const_cast**<T>(v))
[]	subscripting	(ex: a**[**i**]**)
...()	function call	(ex: func**(**1**)**)
(.., .., ..)	expressions list	(ex: func**(**1**,**2**,**3**)**)

- Statements:

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

- Declarations:

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

- Declarators:

	function declarator	(ex: **int func();**)
[ ]	array declarator	(ex: int tab**[**5**]**;**)**
*****	pointer declarator	(ex: int ***i;**)
&	reference declarator	(ex: int**&** i;)
(.., .., ..)	parameter-declaration-list	(ex: int func**(**int i**,** char *j**)**;)
{.., .., ..}	initializer-list	(ex: int tab[] = **{**1**,** 3**,** 5**}**;)
	type qualifier	(ex: **const**, **volatile**)
	type identifier	(ex: sizeof(**int**), new (**int**))

N1 Total number of operators

Definition Total number of operators between the function's header and its closing curly bracket.

Alias ha_topt

n2 Number of distinct operands

Definition Number of different operands between the function's header and its closing curly bracket.

Alias ha_dopd

The following are operands:

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

new	**delete**	**new[]**	**delete[]**	******				
+	**-**	*****	**/**	**%**	**^**	**&**	**|**	**~**
!	**=**	**<**	**>**	**+=**	**-=**	***=**	**/=**	**%=**
^=	**&=**	**|=**	**<<**	**>>**	**>>=**	**<<=**	**==**	**!=**
<=	**>=**	**&&**	**||**	**++**	**--**	**,**	**->***	**->**
()	**[]**	**and**	**or**	**xor**	**mod**	**rem**	**abs**	**not**

N2 Total number of operands

Definition Total number of operands between the function's header and its closing curly bracket.

Alias ha_topd

4.1.4 Keywords

ct_andthen Number of “and_then” operators

Definition Number of occurrences of the logical operator “`&&`” in the function.

ct_break_inloop Number of break in loop

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

ct_break_inswitch Number of break in switch

Definition Number of `break` statements used to exit from embedding `switch` statements in the function.

ct_case Number of case labels

Definition Total number of `case` and `default` labels in the function.

Example

```
switch(var) ;
{
  case A:
  case B: ;
  case C:
     /* A first block of statements */
     i = j + 1;
     break;
  case D:
  case E:
     /* A second block of statements */
     i = k + 1;
     break;
  default:
     /* A third block of statements */
     break;
}
```

ct_case = 6

ct_casepath Number of case block statements

Definition Total number of blocks of statements in `switch` statements in the function.
Sequential case labels are counted for one block of statements.

Example

```
switch(var) ;
{
  case A:
  case B: ;
  case C:
     /* A first block of statements */
     i = j + 1;
     break;
  case D:
  case E:
     /* A second block of statements */
     i = k + 1;
     break;
  default:
     /* A third block of statements */
     break;
}
```

ct_casepath = 3

ct_continue Number of continue statements

Definition Number of `continue` statements in the function.

ct_dowhile Number of do while statements

Definition Number of `do ... while` statements in the function.

ct_for Number of for statements

Definition Number of `for` statements in the function.

ct_if Number of if statements

Definition Number of `if` statements in the function.

ct_orelse Number of “or_else” operators

Definition Number of occurrences of the logical operator “`||`” in the function.

ct_ternary Number of ternary operators

Definition Number of occurrences of the ternary operator “`?:`” in the function.

ct_return Number of return statements

Definition Number of `return` statements in the function plus one if the last statement of the function is not a `return`.

Alias RETU

ct_switch Number of switch statements

Definition Number of `switch` statements in the function.

ct_while Number of while statements

Definition Number of `while` statements in the function.

4.1.5 Structured Programming

In structured programming:

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

Structured programming improves source code maintainability.

ct_bran Number of destructuring statements

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

ct_bran = ct_break_inloop + ct_continue + ct_goto

For structured programming, ct_bran shall be equal to 0.

ct_break Number of break and continue branchings

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

`break` statements in `switch` structures are not counted (cf. ct_breakinswitch).

ct_break = ct_break_inloop + ct_continue

For structured programming, ct_break shall be equal to 0.

Alias COND_STRUCT

ct_exit Number of out statements

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

For structured programming, ct_exit shall be equal to 1.

Alias N_OUT

ct_goto Number of gotos

Definition Number of *`goto`* statements in the function.

For structured programming, ct_goto shall be equal to 0.

Alias GOTO

ESS_CPX Essentiel complexity

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

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

4.1.6 Control Graph

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

- *IBM Rational Logiscope - Basic Concepts*.

ct_decis Number of decisions

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

Alias N_STRUCT

ct_loop Number of loops

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

ct_nest Maximum nesting level

Definition Maximum nesting level of control structures in a function.
Also available: LEVL = ct_nest + 1

ct_path Number of paths

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

Alias PATH

ct_vg Cyclomatic number (VG)

Definition Cyclomatic number of the control graph of the function.

Alias VG, ct_cyclo

DES_CPX Design complexity

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

4.1.7 Relative Call Graph

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

- *IBM Rational Logiscope - Basic Concepts*.

CALL Number of calls

Definition Number of calls in the function.
Each call to the same function counts for one.

cg_entropy Relative call graph entropy

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

cg_ hiercpx Relative call graph hierarchical complexity

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

cg_levels Relative call graph levels

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

cg_strucpx Relative call graph structural complexity

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

cg_testab Relative call graph testability

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

dc_calls Number of direct calls

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

Alias DRCT_CALLS

dc_calling Number of callers

Definition Number of functions calling the designated function.

Alias NBCALLING

IND_CALLS Relative call graph call-paths

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

4.2 Module Scope

4.2.1 Line Counting

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

- *IBM Rational Logiscope - Basic Concepts*.

md_blank Number of empty lines

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

md_comm Number of lines of comments

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

md_cpp Number of preprocessor statements

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

md_line Total number of lines

Definition Total number of lines in the module.

md_loc Number of lines of code

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

md_sbra Number of lines with lone braces

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

md_stat Number of statements

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

4.3 Application Scope

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

4.3.1 Line Counting

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

- *IBM Rational Logiscope - Basic Concepts*.

Note that the line counting only considers the C source files specified in the Logiscope project: i.e. usually files suffixed by ".c". Header files are not taken into account in line counting for the application.

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_scpp Number of preprocessor statements

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

4.3.2 Application Aggregates

ap_func Number of application functions

Definition Number of functions defined in the application.

Alias LMA

ap_stat Number of statements

Definition Sum of numbers of statements (i.e. lc_stat) of all the functions defined in the application source files.

ap_vg Sum of cyclomatic numbers

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

Alias VGA, ap_cyclo

4.3.3 Application Call Graph

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

- *IBM Rational Logiscope - Basic Concepts*.

ap_cg_cycle Call graph recursions

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

Alias GA_CYCLE

ap_cg_edge Call graph edges

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

Alias GA_EDGE

ap_cg_leaf Call graph leaves

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

Alias GA_NSS

ap_cg_levl Call graph depth

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

Alias GA_LEVL

ap_cg_maxdeg Maximum callers/called

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

Alias GA_MAXDEG

ap_cg_maxin Maximum callers

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

Alias GA_MAX_IN

ap_cg_maxout Maximum called

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

Alias GA_MAX_OUT

ap_cg_node Call graph nodes

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

Alias GA_NODE

ap_cg_root Call graph roots

Definition Number of roots functions in the application call graph.

Alias GA_NSP

Chapter 5

Standard Programming Rules

5.1 Standard Programming Rules

Logiscope RuleChecker C comes with programming rules based on:

- Industrial C language programming standards,
- IBM Rational experience in Software Product Evaluation.

Different industrial programming standards sometimes contain contradictory rules. For example, the character '_' is sometimes authorized under certain conditions (not at the beginning or at the end of a key, or no consecutive '_' characters), and sometimes prohibited altogether.

Therefore some of the rules resulting from these standards may be contradictory. However, they are made available to the user for selecting the appropriate sub-set of applicable rules in his/her context.

Rules are organized in Rule Sets according to their type. *Logiscope RuleChecker C* comes with several default Rule Sets:

- Code Presentation,

- Complexity,

- Control Flow,

- Naming,

- Portability,

- Resource.

5.1.1 Presentation of rules

Each rule is described as follows:.

Key: Summary	the **Key** of the rule file as specified in the **.KEY** field; the **Key** is made of : - a prefix related to the rule set the rule belongs to: e.g. **CodePres_**, **ControlFlow_**, **Complexity_**, **Naming_**, **Portability_** or **Resource_**; - an ordering number; - a nmenomic; a summary of the rule as specified in the **.NAME** field of the rule file.
Description	the description of the programming rule as provided in the **description** and/or **role** options of the **.TITLE** field of the corresponding rule file.
Role	the software characteristic(s) enforced by the rule.

The complete name of the rule file is <*log_install_dir*>/**Ref/Rules/C/builtin/***Key***.rl** where <*log_install_dir*> is the Logiscope installation directory.

5.1.2 Rule Sets

Code Presentation

Code Presentation rules are rules restricting how code is presented, in order to improve code analysability and prevent maintenance problems, etc.

CodePres_1_DeclarationPerLine: One declaration per line

Definition	Each line must contain no more than one declaration.
Role	Maintainability.

CodePres_2_NumberStatements: limited number of statements

Definition	The number of statements shall not exceed 100 in a function and 1000 in a module.
Role	Maintainability, Reliability

CodePres_3_FileLength: Length of files

Definition	A file shall not exceed 2000 lines.
Role	Maintainability.

CodePres_4_StatementSwitch: Number of first level statements per switch branch

Definition	The number of first level statements in each clause of a switch statement shall not exceed 10.

Role	Maintainability.

CodePres_5_StatementSwitch: Limited total number of statements per switch branch

Definition	The total number of statements in each clause of a switch statement shall not exceed 25 (all levels included).
Role	Maintainability.

CodePres_6_CommentStatementLine: No comment and statement on the same line

Definition	A comment must be on a line without any statement. The exception concerns a comment written on a single line after a statement.
	Example: *while ((a>0) || (b>0) || (c>0)) { /* Comment*
	** on several lines*
	** and barely readable*
	**/*
	}
	while (a>0) { /* Accepted comment */

Role	Maintainability.

CodePres_7_ExtensionHeader: Included files have the extension .h

Definition	Included files have the extension .h. If those files contain data definition or code, the user can define another extension (.db for example for tables of a database.)
Role	Maintainability..

CodePres_8_EnumBoolean: Enum boolean type

Definition	Systematically define a *Boolean* enumerated type containing two values : true and false.
Role	Maintainability.

CodePres_9_ParamFunction: Maximum number of parameters

Definition	The number of parameters of a function is limited to 7. This number may be customized.
Role	Maintainability.

CodePres_10_StatementPerLine: One statement per line

Definition	No more than one basic statement per line.
Role	Maintainability.

CodePres_11_ControlStructure: Control structure on a new line

Definition	A control structure (*do, while, for, if, else, switch, return, break, continue*) shall start on a new line.
Role	Maintainability.

CodePres_12_BlankLine: Blank line after definitions

Definition	Function definition/declaration and function body must be separated by a blank line.
Role	Maintainability.

CodePres_13_Brace: Braces alone on a line

Definition	Each brace (opening and closing) must be placed alone on a line.
Role	Maintainability.
Parameter	If the value of the variable "exceptionAllowed" is set to 1, then some exceptions are allowed: - the block only includes one instruction: - the braces and the instruction are placed on a single line. - Inside a block, the instructions are indented by 2 spaces with respect to the braces. Note: avoid using tabulations for indentations, the way they are interpreted depends on the editor used (portability). No automatic alignment check

CodePres_14_CommentDeclaration: Comment for declaration

Definition	Declarations must be commented: Each declaration (type, variable, enumeration item, structure field) is commented. The directives to the pre-processor are commented with the name of the associated variable.
Role	Maintainability.

CodePres_15_PointerDeclaration: Pointer declaration

Definition	In the declaration of a pointer to a data type, the * character shall be stuck to the pointer's identifier.
Role	Maintainability.

CodePres_16_SpacingRef: No space before and after '.' and '-> '

Definition	There shall be no blank before or after the . and -> operators.
Role	Maintainability

CodePres_17_SpacingOperator: No space between operators and operands

Definition	Operators ++, -, & (functionAddress), * (functionRef) shall be stuck to their operand.
Role	Maintainability.

CodePres_18_SpacingParameter: Function parameters spacing

Definition	Do not insert a blank after the opening parenthesis or before the closing one. Insert a blank before the opening parenthesis of a function or macro call.
Role	Maintainability.

CodePres_19_LineLength: Length of lines

Definition	A line in a source file shall not exceed 80 characters.
Role	Maintainability, Portability.

CodePres_21U_InclusionLevel: Number of inclusion levels

Definition	The inclusion relation graph of a file shall not have more than 2 levels.
Role	Portability.
Note	Only available on Unix platforms.

CodePres_22U_CommentPrepro: Comment directivess

Definition	The directives #else and #elif shall have a comment.
Role	Portability.
Note	Only available on Unix platforms.

CodePres_23U_Antislash: Use of \ s

Definition	Declarations using ”\” shall not be used.
Role	Portability.
Note	Only available on Unix platforms.

CodePres_24U_Indent: Indentations

Definition	Statements, comments, *{* and *}* shall be indented.
Role	Maintainability.
Note	Only available on Unix platforms.

CodePres_25_SingleLineComment: Use of comments

Definition	Comments shall be one line long.
Role	Maintainability.

CodePres_26_CommentDefinition: Definition comments

Definition	All the definitions got a comment.
Role	Maintainability.

CodePres_28_Definitions: Definitions

Definition	A module's "*.c"* body file must contain the "in public" definitions of the exported functions, and the "in public" definitions of the exported variables.
Role	Maintainability.

CodePres_29_SpacingUnaryOperator: No space after unary operators

Definition	Unary operators *!* and ~ must be stuck to their operand to avoid confusion with binary operators.
Role	Maintainability.

CodePres_30_Define: Define altogether after include

Definition	The *#define* preprocessing directives shall be grouped altogether. This group shall follow the *#include* directives.
Role	Maintainability.

Complexity

Complexity rules concern operators, statements and language traps in order to improve code reliability and maintainability.

Complexity_1_MultipleAssignment: No multiple assignments

Definition	Multiple assignments shall not be used.
Example	*x = y = z ;*
Role	Maintainability.

Complexity_2_NoTernaryOp: No ternary operator

Definition	The ternary operator (*?:*) shall not be used.
Example	*z = (a>b) ? a : b*
Role	Maintainability.

Complexity_3_NoUnary+: No unary + operator

Definition	The unary + operator shall not be used
Example	*x = +10;*
Role	Maintainability.

Complexity_4_NoAssignmentOp: Assignment operators not recommended

Definition	Assignment operators other than = (e.g. **=, /=, %=, &=)* shall not be used.
Role	Maintainability.

Complexity_5_CallResult: Use of the result of the function calls

Definition	A function call must never appear as an independent statement. A function shall never be used for its side-effects
Role	Reliability.

Complexity_6_++--Operators: Use of ++ and --

Definition	The use of ++ and -- shall be limited to simple cases. They shall not be used in statements where other operators occur. The prefix use is always forbidden.
Role	Maintainability.

Complexity_7_NoCast: No explicit casting

Definition	Cast functions shall not be used..
Role	Maintainability, Portability.

Complexity_8_NoMultipleInit: Initialisations in multiple declarations

Definition	Initialisations in multiple declarations are forbidden Initialisations only occur on single expressions and are done, when possible, through symbolic constants.
Role	Maintainability.

Complexity_9_Macro: One statement by macro

Definition	A macro shall not contain several statements. Multi-line macros shall not be used.
Role	Maintainability.

Complexity_10_FieldAddressing: No (*ptr). field

Definition	To address a structure field via a pointer to the structure, the notation *ptr>Field* shall be used. Example: *struct foo {* int a; int b; } ; struct foo *p_foo ; p_foo->a ; /* Correct */ (*p_foo).a ; /* Rejected */
Role	Maintainability.

Complexity_11_NoCommaAndTernary: ?: and , operators

Definition	*?:* and , shall not be used
Role	Maintainability.

Complexity_12_OperatorInCondition: Operator in conditions

Definition	A condition with more than 4 operators shall not contain several distinct operators.
Role	Maintainability.

Complexity_13_SimpleTest: No simple statements

Definition	Statements like *x == y ;* or *x != y ;* shall not be used..
Role	Reliability.

Complexity_14_InclusionLevel: Only one inclusion level

Definition	File inclusion shall not exceed one level. *Include* are therefore forbidden in header files.
Role	Maintainability.

Complexity_15_Sizeof: Parentheses for sizeofl

Definition	Always uses parentheses to isolate the sizeof operand.
Role	Maintainability.

Control Flow

These rules deal with the control flow of the program in order to improve its maintainability and reliability.

ControlFlow_1_NoDeadCode: No inaccessible code

Description	There shall be no dead code, especially after *goto* and *return* statements.
Role	Maintainability.

ControlFlow_2_FunctionReturn: Use of return

Description	One *return* statement per function. It shall be the last statement of the function.
Role	Maintainability.

ControlFlow_3_NoGoto: No goto

Description	Goto statement, especially local goto statement, shall not be used.
Role	Maintainability.

ControlFlow_4_ThenElse: Then and else parts of if instructions

Description	The *then* and *else* parts of *if* statements shall not be void.
Role	Maintainability.

ControlFlow_5_NoBreakContinue: Use of break and continue

Description	*Break* and *continue* shall not be used in loops (*for, do, while*)
Role	Maintainability.

ControlFlow_6_DefaultInSwitch: Default in switch

Description	The *default* clause is mandatory in a *switch* statement.
Role	Reliability.

ControlFlow_7_BreakInSwitch: Break in case clauses

Description	Break is mandatory for case clauses containing statements and shall be the last statement of the clause.
Role	Reliability.

ControlFlow_8_BreakPathInSwitch: Break in paths of switch branch

Description	Break is mandatory for case clauses containing statements. If break is not the last instruction of a switch branch, one break shall be added for each path.
Role	Reliability.

ControlFlow_9_ControlStructureNesting: Control structure nesting limited

Description	Control structure nesting is limited to 6 levels
Role	Understandability, Maintainability.

ControlFlow_10_SwitchBetterThanIf: Switch and several if

Description	It is better to use a *switch* than several *if* statements.

Example	if ()
	else if ()
	[else if ()]*
	else
	will provoke violations (only 3 nested *if* statements).

Role	Maintainability.

ControlFlow_11_OneBreakContinue: One break or continue

Description	Only one *continue* or *break* statement is authorized in the body of *for*, *do* or *while* loops.
Role	Maintainability.

Naming

Naming rules define the way the different entities of the application can be named. They improve maintainability of the code.

Naming_1_MinLength: Minimum length of identifiers

Description	Identifiers shall be at least X+1 characters long. X may be customized.
Role	Maintainability.

Naming_2_Underscore: '_' at the beginning or at the end of an identifier

Description	Identifiers shall not start or finish with the character underscore '_'
Example	It is difficult to distinguish *_name, name and name_*.
Role	Maintainability.

Naming_3_DoubleUnderscore: No double underscore

Description	Identifiers shall not contain two underscore '_' characters consecutively.
Example	It is difficult to distinguish *_name* and *__name*.
Role	Maintainability.

Naming_4_NoUnderscore: Underscore in identifiers

Description	The underscore character '_' shall not be used.
Role	Maintainability.

Naming_5_GlobalVariable: Global variable naming

Description	The first character of a global variable identifier is upper-case. The others are lower-case letters, numbers or the underscore character.
Role	Maintainability.

Naming_6_LocalVariable: Local variable naming

Description	The first character of a local variable identifier is lower-case. The others are lower-case letters, numbers or the underscore character.
Role	Maintainability.

Naming_7_Function: Function naming

Description	The first character of a function identifier is lower-case. The others are lower-case letters, numbers or the underscore character.
Role	Maintainability.

Naming_8_Constant: Constant naming

Description	The first character of a constant identifier is upper-case. The others are upper-case letters, numbers or the underscore character.
Role	Maintainability.

Naming_9_Macro: Macro naming

Description	The first character of a macro identifier is upper-case. The others are upper-case letters, numbers or the underscore character.

Role	Maintainability.

Naming_10_Type: Type naming

Description	The first character of a type identifier is upper-case. The others are upper-case letters, numbers or the underscore character.
Role	Maintainability.

Naming_11_StructField: Structure type fields naming

Description	The first character of a structured type component identifier is upper-case. The others are lower-case letters, numbers or the underscore character.
Role	Maintainability.

Naming_12_MainParam: Parameters of main:

Description	Parameters of *main* shall be named: - *argc*: integer representing the command parameter number - *argv*: array of strings of length of *argc*
Role	Maintainability.

Naming_13_EnumConstant: Enum constant naming

Description	Enum constants shall be written with upper-case letters.
Role	Maintainability.

Naming_14U_Module: Module naming

Description	All C modules consist of a body file and an interface file. These two files have the same root which is the module name.
Role	Maintainability.
Note	Not available on Windows platforms.

Naming_15_Prefix: Name prefix

Description	This concerns module level entities (internal and external). Choosing a module name as prefix guarantees that all prefixes are distinct.
Role	Maintainability.

Naming_16_SymbolNaming: Symbol naming

Description	This rule concerns all symbols of an application: - Language keyword: Lower-case letters, - [macro-]function: First letter upper-case and the others lower-case, - [macro-]constant: Upper-case letters, - Type: First letter upper-case, the others lower-case, - Structure Field: Lower-case letters, - Enumeration items: Lower-case letters, - Variable: Lower-case letters, - Parameters: Lower-case letters.

Role	Maintainability.

Portability

This set of rules concern characters, keywords and C Standard. They improve portability of the program.

Portability_1_C++Keywords: C++ keywords use

Description	Keywords from C++ language (*class, new, friend*...) shall not be used.
Role	Portability.

Portability_2_NoDollar: No '$' in identifier

Description	The '$' character shall not be used in an identifier. Restriction imposed by the C ANSI standard.
Role	Portability.

Portability_4_CharIdentifier: Authorized characters

Description	The only authorized characters in identifiers shall be: - letters (upper- and lower-case), - numbers, - underscore character '_' ;
Role	Portability.

Portability_5_NoSignedRightShift: Use of >>

Description	The right shift operator >> shall not be used on signed integer.
Role	Portability.

Portability_6_MainNaming: Exit from main

Description	Only the *exit* function shall be used to go out from *main*.
Role	Portability.

Portability_7_NoRecursiveHeader: No recursive inclusion

Description	Header files shall not include themselves recursively.
Role	Portability.

Portability_8U_ConditionalCompilation: Conditional compilation

Description	Header files shall have the following structure : #ifndef ModuleName_h_ #define ModuleName_h_ .... #endif
Role	Portability.

Note	Not available on Windows platforms.

Portability_9U_AbsolutePathInclude: #include

Description	File names in *#include* directives must be in the same case than the file name and shall not contain any absolute path.
Role	Portability.
Note	Not available on Windows platforms.

Portability_10U_DirectiveFirstColumn: Compilation directive

Description	The character # of compilation directives shall be on the first column.
Role	Portability.
Note	Not available on Windows platforms.

Portability_11U_NoAsmDirective: #asm

Description	#asm directive shall not be used.
Role	Portability.
Note	Not available on Windows platforms.

Portability_12U_FilenameLength: File naming

Description	File names shall be lower-case and shall not exceed 8 characters for the name and 3 characters for the extension.
Role	Portability.
Note	Not available on Windows platforms.

Portability_13_NoTab: Use of tabulations

Description	Tabulations shall not be used in source files.
Role	Portability.

Resource

Resource rules are rules restricting how resources in the application are used, in order to improve code maintainability, efficiency and reliability.

Resource_1_AccessArray: Access to an array

Description	A pointer shall be used to run through successive elements of an array rather than an index.
Role	Efficiency.

Resource_2_ForCounter: Counter in for statements

Description	The counter in a *for* statement shall not be modified inside the loop and shall be a local variable.
Role	Reliability.

Resource_3_DeclarationInitSeparate: Declaration and initialisation separate

Description	Declaration and initialisation of a variable shall be separate.
Role	Maintainability.

Resource_4_DeclarationInitCombine: Declaration and initialisation combined

Description	Declaration and initialisation of a variable shall be done at the same time, if possible.
Role	Reliability.

Resource_5_LocalDeclaration: Local variable declaration

Description	Declaration of local variables in an instruction block shall not be used.
Role	Maintainability.

Resource_6_GlobalDeclaration: Global variable declaration

Description	Global objects shall be declared in an inclusion file.
Role	Maintainability

Resource_7_VariableUse: Use of variables

Description	Declared variables shall be used.
Role	Maintainability.

Resource_8_FunctionUse: Use of functions

Description	Declared functions shall be used.
Role	Maintainability.

Resource_9_ParameterUse: Use of parameters

Description	Function parameters shall be used.
Role	Maintainability.

Resource_10_NoGlobalParameter: Global variable as a parameter

Description	A global variable shall not be used as a parameter.
Role	Maintainability.

Resource_11_InputParameter: Entry parameter

Description	A function’s input parameter shall be either a pointer to *const*, or passed by value.
Role	Reliability.

Resource_12_NoExternBody: No extern in body file

Description	The keyword *extern* shall not be used in a.*c* file.
Role	Maintainability.

Resource_13_NoStaticInFunc: Static in functions

Description	The keyword *static* shall not be used in the body of a function.
Role	Reliability.

Resource_14_ExternHeader: Variable in header files

Description	Declarations of variables in an header file shall be preceded by *extern*.
Role	Reliability.

Resource_15_NoFunctionHeader: Definition of functions

Description	Functions (other than macros) shall not be defined in an header file.
Role	Maintainability.

Resource_16_FileExtension: File extension

Description	The header file shall have the extension *.h* and the body file the extension *.c*.
Role	Maintainability.

Resource_18_NoBodyInclusion: Body inclusion

Description	A *.c* file shall not be included in another file, it shall be compiled to give an object module.
Role	Maintainability.

Resource_19_NoBitfield: No bitfields

Description	Bitfields shall not be used.
Role	Reliability.

Resource_20_NoAuto: Auto attribute

Description	Declaration of variables local to a function shall never be made with .
Role	Reliability.

Resource_21_ArrayInit: Array initialization

Description	Initialization of an array shall conform to its structure.
Role	Readability.

Resource_22_PointerInit: Pointer initialization

Description	A pointer shall always be initialized. If it points to no known variable, it shall be initialized to *NULL*.
Role	Reliability.

Resource_23_WhileInit: Initialization of while statement variables

Description	The initial value of a parameter of a *while* loop shall be known before entering the loop. If not, there shall be a comment explaining the initial state of the parameter, the comment shall be situated at *MaxLine* of the *while* statement. *MaxLine* may be customized.
Role	Reliability.

Resource_24_ConstVolatileInit: Initialization of const and volatile variables

Description	Only *const* and *volatile* variables to a function shall be initialized when they are defined.
Role	Reliability.

Resource_26_TypedefUnionStruct: Typedef for unions and structures

Description	A *typedef* shall not be used to mask structures or unions.
Role	Maintainability.

Resource_30_EnumInit: Initialization of enumerations

Description	The initialization of enumeration fields shall not be explicit.
Role	Reliability.

Resource_31_StructUnion: Union and structure

Description	Using the *union* type shall be limited to declaring partially variable types.
Role	Maintainability.

Resource_32_ForSpecification: Specification of for

Description	All parts a *for* statement shall be filled.
Role	Reliability.

5.2 MISRA Programming Rules

The **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation has published guidelines containing list of rules for the use of the C programming language for embedded systems, especially for embedded automotive systems:

- *Guidelines For The Use Of The C Language In Vehicle Based Software* - April 1998 [MISRA-C:1998],
- *MISRA-C:2004 Guidelines for the use of the C language critial systems* - October 2004 [MISRA-C:2004].

Apart from standard programming rules, MISRA programming rules packages are available. These packages are not shipped with *Logiscope RuleChecker C* and have to be purchased in addition to the product. Compressed and encrypted files are available in the <*log_install_dir*> directory.

Rules are organized in rule sets according to their classification i.e. Required or Advisory in the corresponding MISRA Guidelines:

- the MISRA Required rule set,
- the MISRA Advisory rule set,
- the MISRA "All" rule set containing all of the rule sets presented above.

When using the MISRA packages, please rename the rulesets.lst.MISRA file to rulesets.lst in the directory where the packages have been extracted.

5.2.1 Presentation of the rules

Each rule is described as follows:.

Key: Summary	the **Key** of the rule file as specified in the **.KEY** field; the **Key** is made of the **MISRA_** prefix followed by the rule identifier in the corresponding MISRA Guidelines. a summary of the rule as specified in the **.NAME** field of the rule file.
Description	the description of the programming rule as provided in the **description** and/or **role** options of the **.TITLE** field of the corresponding rule file.
Role	the software characteristic(s) enforced by the rule.
Classification	the classification of the rule as specified in the corresponding MISRA Guidelines: i.e. Required or Advisory

The complete name of the rule file is <*log_install_dir*>/**Ref/Rules/C**/*Key*.**rl** where <*log_install_dir*> is the Logiscope installation directory. The syntax of this file is described in the reference part in the "File - programming rules" field.

5.2.2 MISRA-C:1998 Rule Package

83 of the 93 “Required” rules specified in the MISRA-C:1998 document can be checked using the *Logiscope RuleChecker C* MISRA 1998 programming rule package as well as 23 of the 34 “Advisory” rules.

MISRA_Rule5: ISO C standard Characters only

Description	Only those characters and escape sequences which are defined in the ISO C standard shall be used.
Role	Maintainability.
Classification	Required.

MISRA_Rule7: Trigraphs

Description	Trigraphs shall not be used.
Role	Maintainability.
Classification	Required.

MISRA_Rule8: Multibyte characters

Description	Multibyte characters and wide string literals shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule9: Nested comments

Description	Comments shall not be nested.
Role	Portability.
Classification	Required.

MISRA_Rule11: Length of identifiers

Description	Identifiers shall not exceed 31 characters. Restriction imposed by the C ANSI standard.
Role	Portability.
Classification	Required.

MISRA_Rule12: Name of identifiers

Description	No identifier in one name space shall have the same spelling as an identifier in another name space.
Role	Reliability.
Classification	Advisory.

MISRA_Rule13: Basic types

Description	The basic types of *char, int, short, long, float and double* should not be used, but specific-length equivalents should be *typedef’d* for the specific compiler.

Role	Reliability.
Classification	Advisory.

MISRA_Rule14: Type char

Description	The type *char* shall always be declared as *unsigned char* or *signed char*.
Role	Portability.
Classification	Required.

MISRA_Rule16: Underlying representation of floating point numbers

Description	The underlying bit representation of floating point numbers shall not be used in any way by the programmer.
Role	Reliability.
Classification	Required.

MISRA_Rule17: Typedef names

Description	Typedef names shall not be reused.
Role	Reliability.
Classification	Required.

MISRA_Rule18: Numeric constants and suffixes

Description	Numeric constants should be suffixed to indicate type, where an appropriate suffix is available.
Role	Reliability.
Classification	Advisory.

MISRA_Rule19: Octal constants

Description	Octal constants other than zero shall not be used.
Role	Maintainability.
Classification	Required.

MISRA_Rule20: Declaration before use

Description	All objects and functions identifiers shall be declared before use.
Role	Reliability.
Classification	Required.

MISRA_Rule21: Hidden identifiers linkage of identifiers

Description	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier. Identifiers shall not simultaneously have both internal and external linkage in the same translation unit. Rule 24 violations will be caught by this rule and flagged as rule 21 violations.

Role	Reliability.
Classification	Required.

MISRA_Rule22: Object declarations

Description	Declarations of objects should be at function scope unless a wider scope is necessary.
Role	Reliability.
Classification	Advisory.

MISRA_Rule23i: Functions declaration

Description	A declaration of function at file scope should be static where possible.
Role	Maintainability, Reliability
Classification	Advisory.

MISRA_Rule25: External definition

Description	An identifier with external linkage shall have exactly one external definition.
Role	Reliability.
Classification	Required.

MISRA_Rule26: Declarations of functions must be compatible

Description	If objects or functions are declared more than once their types shall be compatible.
Role	Reliability, Portability.
Classification	Required.

MISRA_Rule27: External declarations

Description	External objects should not be declared in more than one file.
Role	Reliability.
Classification	Advisory.

MISRA_Rule28: Use of register

Description	The *register* storage class specifier shall not be used.
Role	Portability.
Classification	Advisory.

MISRA_Rule29:Use of tags

Description	Use of tags shall agree with its declaration.
Role	Reliability.
Classification	Required.

MISRA_Rule30: Assignment

Description	All automatic variables must have been assigned a value before being used.
Role	Reliability.
Classification	Required.

MISRA_Rule31: Structured initialisation

Description	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.
Role	Reliability.
Classification	Required.

MISRA_Rule32: Enumeration initialization

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

MISRA_Rule33: Side effects

Description	The right hand operand of a *&&* or || operator shall not contain side effects.
Role	Reliability, Portability.
Classification	Required.

MISRA_Rule34: Logical operand

Description	Operands of a logical *&&* and || shall be primary expressions.
Role	Reliability.
Classification	Required.

MISRA_Rule35: Test and assignment result

Description	Assignment operators shall not be used in expressions which returns *Boolean* values.
	Example:
	if (x = y) { /* Violation */ }
	
	if ((x = y) != 0) { /* Violation */ }
	
	x = y ;
	if (x != 0) { /* Correct */ }

Role	Reliability.
Classification	Required.

MISRA_Rule37: Bitwise operations

Description	Bitwise operations (~, <<, >>, &, ^ and |) shall not be performed on signed integer types.
Role	Reliability.
Classification	Required.

MISRA_Rule38: Shift operator and right hand operand

Description	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left hand operand (inclusive).
Role	Reliability.
Classification	Required.

MISRA_Rule39: Unary minus operator

Description	The unary minus operator shall not be applied to an unsigned expression.
Role	Reliability.
Classification	Required.

MISRA_Rule40: Operator sizeof

Description	The sizeof operator should not be used on expressions that contain side effects.
Role	Reliability.
Classification	Advisory.

MISRA_Rule42: Comma operator

Description	The comma operator shall not be used, except in the control expression of a *for* loop.
Role	Reliability.
Classification	Required.

MISRA_Rule43: Conversions

Description	Implicit conversions which may result in a loss of information shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule44: Redundant casts

Description	Redundant explicit casts should not be used.
Role	Reliability
Classification	Advisory.

MISRA_Rule45: Cast and pointers

Description	Type casting from any type to or from pointers shall not be used.

Role	Reliability.
Classification	Required.

MISRA_Rule46: Evaluation order

Description	The value of an expression shall be the same under any order of evaluation that standard permits.
Role	Reliability
Classification	Required.

MISRA_Rule48: Mixed precision arithmetic and cast

Description	Mixed precision arithmetic should use explicit casting to generate the desired result.
Role	Reliability
Classification	Advisory.

MISRA_Rule50: Test between floats

Description	Floating point variables shall not be tested for exact equality or inequality.
Role	Reliability.
Classification	Required.

MISRA_Rule52: Unreachable code

Description	There shall be no unreachable code.
Role	Reliability.
Classification	Required.

MISRA_Rule53: Non-null statements

Description	Non-null statements shall have a side-effect.
Role	Reliability.
Classification	Required.

MISRA_Rule54: Location of null statements

Description	A null statement shall occur on a line by itself, and shall not have any other text on the same line.
Role	Reliability.
Classification	Required.

MISRA_Rule55: No labels

Description	Labels should not be used, except in *switch* statements.
Role	Understandability
Classification	Advisory.

MISRA_Rule56: Goto

Description	The *goto* statement shall not be used.
Role	Maintainability.
Classification	Required.

MISRA_Rules5758: Break and continue

Description	The *continue* statement shall not be used. The *break* statement shall not be used (except to terminate the cases of a *switch* statement).
Role	Maintainability.
Classification	Required.

MISRA_Rule59: Use of braces

Description	Statements forming the body of an *if, else if, else, while, do ... while* or *for* statement shall always be in brackets.
Role	Maintainability.
Classification	Required.

MISRA_Rule60: Then and else

Description	All *if, else if* constructs should contain a final *else* clause.
Role	Reliability, Understandability
Classification	Advisory.

MISRA_Rule61: Break in switch

Description	Every non-empty *case* clause in a *switch* statement shall be terminated with a *break* statement.
Role	Reliability.
Classification	Required.

MISRA_Rule62: Default in switch

Description	All *switch* statements should contain a final *default* clause.
Role	Reliability.
Classification	Required.

MISRA_Rule63: Switch and boolean

Description	A *switch* expression should not represent a Boolean value.
Role	Maintainability.
Classification	Advisory.

MISRA_Rule64: Switch without case

Description	Every *switch* statement shall have at least one *case*.
Role	Maintainability.

Classification	Required.

MISRA_Rule65: Loop counter

Description	Floating point variables shall not be used as loop counters.
Role	Reliability.
Classification	Required.

MISRA_Rule66: Loop control

Description	Only expressions concerned with loop control should appear within a for statement.
Role	Reliability.
Classification	Advisory.

MISRA_Rule67: Counter in for statements

Description	Numeric variables being used within a *for* loop for iteration counting should not be modified in the body of the loop.
Role	Reliability.

MISRA_Rule68: Scope of functions

Description	Functions shall always be declared at file scope.
Role	Maintainability.
Classification	Required.

MISRA_Rule69: Variable number of arguments

Description	Functions with variable numbers of arguments shall not be used.
Role	Reliability, Maintainability
Classification	Required.

MISRA_Rule70: Recursion

Description	Functions shall not call themselves, either directly or indirectly.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_Rule71: Prototyping

Description	Functions shall always have prototype declarations and the prototype shall be visible at both the function declaration and call.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_Rule7576: Void type and functions

Description	Every function shall have an explicit return type. Functions with no parameters shall be declared with parameter type *void*.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_Rule78: Parameters

Description	A parameter number passed to a function shall match the function prototype.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_Rule79: Values of void functions

Description	Values returned by *void* functions shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule80: Void expressions and function parameters

Description	Void expressions shall not be passed as function parameters.
Role	Reliability.
Classification	Required.

MISRA_Rule81: Function parameters and const

Description	Const qualification should be used on function parameters which are passed by reference, where it is intended that the function will not modify the parameter.
Role	Reliability.
Classification	Advisory.

MISRA_Rule82: Use of return

Description	A function should have a single point of exit.
Role	Maintainability.
Classification	Advisory.

MISRA_Rule83i: Functions with non-void return types

Description	For functions with non-void return type, there shall be one *return* statement for every exit branch.
Role	Reliability.
Classification	Required.

MISRA_Rule83ii: Functions with non-void return types

Description	For functions with non-void return type, each *return* shall have an expression.
Role	Reliability.
Classification	Required.

MISRA_Rule83iii: Functions with non-void return types

Description	For functions with non-void return type, the *return* expression shall match the declared return type.
Role	Reliability.
Classification	Required.

MISRA_Rule84: Void functions

Description	For functions with void return type, *return* statements shall not have an expression.
Role	Reliability.
Classification	Required.

MISRA_Rule85: Function with no parameters

Description	Functions called with no parameters should have empty parentheses.
Role	Reliability.
Classification	Advisory.

MISRA_Rule87: Code structure

Description	#include statements in a file shall only be preceded by other preprocessor directives or comments.
Role	Reliability.
Classification	Required.

MISRA_Rules8889: #include syntax

Description	Non-standard characters shall not occur in header file names in *#include* directive. The *#include* directive shall be followed by either a <filename> or “filename” sequence.
Role	Reliability.
Classification	Required.

MISRA_Rule91: Define and undefine in a block

Description	Macros shall not be *#define'd* and *#undef'd* within a block.
Role	Reliability.
Classification	Required.

MISRA_Rule92: Use of #undef

Description	#undef should not be used.
Role	Reliability.
Classification	Advisory.

MISRA_Rule93: Functions and macros

Description	A function should be used in preference to a function-like macro.
Role	Reliability.
Classification	Advisory.

MISRA_Rule94: Function-like macro call

Description	A function-like macro shall not be called without all of its arguments.
Role	Reliability.
Classification	Required.

MISRA_Rule95: Arguments to function-like macros

Description	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.
Role	Reliability.
Classification	Required.

MISRA_Rule96i: Parentheses for macro occurences

Description	In a definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses.
Role	Reliability.
Classification	Required.

MISRA_Rule96ii: Parentheses for macro occurences

Description	In a definition of a function-like macro, the whole definition shall be enclosed in parentheses.
Role	Reliability.
Classification	Required.

MISRA_Rule97: Identifiers in pre-processor directives

Description	Identifiers in pre-processor directives should be defined before use.
Role	Reliability.
Classification	Advisory.

M ISRA_Rule98: # and ## in macros

Description	There shall be at most one occurence of the # or ## pre-processor operators in a single macro definition.
Role	Reliability.

Classification	Required.

M ISRA_Rule99: All uses of the #pragma directive shall be documented and explained

Description	The line beofre the #pragma directive shall containa comment.
Role	Maintainability.
Classification	Required.

MISRA_Rule100:Operator defined

Description	The defined pre-processor operator shall only be used in one of the two standard forms.
Role	Reliability.
Classification	Required.

MISRA_Rule101: Pointer arithmetic

Description	Pointer arithmetic should not be used.
Role	Reliability.
Classification	Advisory.

MISRA_Rule102: Reference complexity

Description	No more than 2 levels of pointer indirection should be used.
Role	Maintainability.
Classification	Advisory.

MISRA_Rule103: Pointers and operators

Description	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to the same array, structure or union.
Role	Reliability.
Classification	Required.

MISRA_Rule104: Pointers to functions

Description	Non-constant pointers to functions shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule105: Pointers to functions

Description	All the functions pointed to by a single pointer to function shall be identical in the number and type of parameters and the return type.
Role	Reliability.
Classification	Required.

MISRA_Rule106: Address assignment

Description	The address of an object with automatic storage shall not be assigned to an object which may persist after the object has ceased to exit.
Role	Reliability.
Classification	Required.

MISRA_Rule107: Null pointer

Description	The null pointer shall not be de-referenced.
Role	Reliability.
Classification	Required.

MISRA_Rule108: Members of structures and unions

Description	In the specification of a structure or union type, all members of the structure or union shall be fully specified.
Role	Reliability.
Classification	Required.

MISRA_Rule109: Variable storage

Description	Overlapping variable storage shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule110: Unions access

Description	Unions shall not be used to access sub-parts of larger data types.
Role	Reliability.
Classification	Required.

MISRA_Rule111: Type of bitfields

Description	Bit fields shall only be defined to be of type unsigned int or signed int.
Role	Reliability.
Classification	Required.

MISRA_Rule112: Two bits long bit fields

Description	Bit fields of type *signed inst* shall be at least two bits long.
Role	Reliability.
Classification	Required.

MISRA_Rule113: Structure fields

Description	All members of a structure (or union) shall be named and shall only be accessed via their name.
Role	Reliability, Maintainability.

Classification	Required.

MISRA_Rule114: Define and undef

Description	Reserved words and standard library function names shall be not redefined or undefined.
Role	Reliability, Maintainability.
Classification	Required.
Note	Implemented using 2 complementary rule scripts.

MISRA_Rule115: Redefinition of standard library function names

Description	Standard library function names shall not be reused.
Role	Maintainability.
Classification	Required.

MISRA_Rule118: Dynamic heap memory

Description	Dynamic heap memory allocation shall not be used.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_Rule119: Errno

Description	The error indicator *errno* shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule120: Offsetof

Description	The macro *offsetof*, in library <stddef.h> shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule121Fct: <locale.h>

Description	<locale.h> and the *setlocale* function shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule122: Setjmp and longjmp

Description	The *setjmp* macro and the *longjmp* function shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule123: signal.h

Description	Signal handling facilities of <signal.h> shall not be used.
Role	Reliability.

Classification	Required.

MISRA_Rule124Fct: stdio.h

Description	The input/ouput library <stdio.h> shall not be used in production code.
Role	Reliability.
Classification	Required.

MISRA_Rules121124Include: <locale.h> and <stdio.h>

Description	<locale.h> and <stdio.h> shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule125: atof, atoi and atol

Description	Library functions *atof, atoi* and *atol* from library <stdlib.h> shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule126: abort, exit, getenv and system

Description	Library functions *abort*, *exit*, *getenv* and *system* from library <stdlib.h> shall not be used.
Role	Reliability.
Classification	Required.

MISRA_Rule127: time.h

Description	Time handling functions of library <time.h> shall not be used.
Role	Reliability.
Classification	Required.

5.2.3 MISRA-C:2004 Rule Package

95 of the 121 “Required” rules specified in the MISRA-C:2004 document can be checked using the *Logiscope RuleChecker C* MISRA 2004 programming rule package as well as 14 of the 20 “Advisory” rules.

MISRA_2_2: No // Comment

Description	Source code shall only use / * ... */ style comments.
Role	Portability.
Classification	Required.

MISRA_2_3: No nested comments

Description	The character sequence /* shall not be used within a comment.
Role	Portability.
Classification	Required.

MISRA_3_4: Use of the #pragma directive

Description	All uses of the #pragma directive shall be documented and explained.
Role	Reliability.
Classification	Required.

MISRA_4_1: Escape sequences

Description	Only those escape sequences which are defined in the ISO C standard shall be used.
Role	Maintainability.
Classification	Required.

MISRA_4_2: Trigraphs

Description	Trigraphs shall not be used.
Role	Maintainability.
Classification	Required.

MISRA_5_1: Length of identifiers

Description	Identifiers (internal and external) shall not rely on the significance of more than 31 characters. Restriction imposed by the C ANSI standard.
Role	Portability.
Classification	Required.

MISRA_5_2: Identifiers linkage and scope

Description	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
Role	Reliability.

Classification	Required.

MISRA_5_3: Typedef names

Description	A typedef name shall be a unique identifier.
Role	Reliability.
Classification	Required.

MISRA_5_4:Use of tags

Description	A tag name shall be a unique identifier.
Role	Reliability.
Classification	Required.

MISRA_5_5: Do not reuse name of static objects

Description	No object or function identifier with static storage duration should be reused.
Role	Reliability.
Classification	Advisory.

MISRA_5_6: Name of identifiers

Description	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
Role	Reliability.
Classification	Advisory.

MISRA_5_7: No reused identifiers

Description	No identifier name should be reused.
Role	Reliability.
Classification	Advisory.

MISRA_6_1: Plain char type usage

Description	The plain char type shall be used only for storage and use of character values.
Role	Reliability.
Classification	Required.

MISRA_6_2: signed/unsigned char type usage

Description	signed and unsigned char type shall be used only for the storage and use of numeric values.
Role	Reliability.
Classification	Required.

MISRA_6_3: Basic types

Description	Typedefs that indicate size and signedness should be used in place of the basic types.
Role	Reliability.
Classification	Advisory.

MISRA_6_4: Type of bitfields

Description	Bit fields shall only be defined to be of type unsigned int or signed int.
Role	Reliability.
Classification	Required.

MISRA_6_5: Two bits long bit fields

Description	Bit fields of type *signed inst* shall be at least two bits long.
Role	Reliability.
Classification	Required.

MISRA_7_1: Octal constants

Description	Octal constants other than zero shall not be used.
Role	Maintainability.
Classification	Required.

MISRA_8_1: Prototyping

Description	Functions shall always have prototype declarations and the prototype shall be visible at both the function declaration and call.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_8_2: Use explicit types

Description	Whenever an object or function is declared or defined, its type shall be explicitly stated.
Role	Reliability, Portability.
Classification	Required.

MISRA_8_4: Declarations of functions must be compatible

Description	If objects or functions are declared more than once their types shall be compatible.
Role	Reliability, Portability.
Classification	Required.

MISRA_8_5:No definition in header

Description	There shall be no definitions of objects or functions in a header file.

Role	Reliability, Portability.
Classification	Required.

MISRA_8_6: Scope of functions

Description	Functions shall be declared at file scope.
Role	Maintainability.
Classification	Required.

MISRA_8_7: Object declarations

Description	Objects shall be defined at block scope if they are only accessed from within a single function.
Role	Reliability.
Classification	Advisory.

MISRA_8_8: External declarations

Description	An external object or function shall be declared in one and only one file.
Role	Reliability.
Classification	Required.

MISRA_8_9: External definition of identifiers

Description	An identifier with external linkage shall have exactly one external definition.
Role	Reliability.
Classification	Required.

MISRA_8_10: File scope declarations

Description	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
Role	Maintainability, Reliability
Classification	Required

MISRA_9_1: Assignment

Description	All automatic variables must have been assigned a value before being used.
Role	Reliability.
Classification	Required.

MISRA_9_2: Structured initialisation

Description	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.
Role	Reliability.
Classification	Required.

MISRA_9_3: Enumeration initialization

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

MISRA_10_1: Integer type conversions

Description	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a) it is not a conversion to a wider integer type of the same signedness, or b) the expression is complex, or c) the expression is not constant and is a function argument, or d) the expression is not constant and is a return expression.
Role	Reliability.
Classification	Required.

MISRA_10_2: Floating type conversion

Description	The value of an expression of floating type shall not be implicitly converted to a different type, if : a) it is not a conversion to a wider floating type, or b) the expression is complex, or c) the expression is a function argument, or d) the expression is a return expression.
Role	Reliability.
Classification	Required.

MISRA_10_3: Integer type casting

Description	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression..
Role	Reliability.
Classification	Required.

MISRA_10_4: Floating type casting

Description	The value of a complex expression of floating type may only be cast to a narrower floating type.
Role	Reliability, Portability.
Classification	Required.

MISRA_10_5: Unsigned casting

Description	If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand..
Role	Reliability.
Classification	Required.

MISRA_10_6:U suffixing

Description	A «U» suffix shall be applied to all constants of unsigned type..
Role	Reliability.
Classification	Required.

MISRA_11_3: Pointer / integral type cast

Description	A cast should not be performed between a pointer type and an integral type.
Role	Reliability.
Classification	Advisory.

MISRA_11_4: Cast between pointers to different object type

Description	A cast should not be performed between a pointer to object type and a different pointer to object type.
Role	Reliability.
Classification	Advisory.

MISRA_12_1: Operator precedence

Description	Limited dependence should be placed on C's operator precedence rule in expression .
Role	Reliability
Classification	Advisory.

MISRA_12_2: Evaluation order

Description	The value of an expression shall be the same under any order of evaluation that standard permits.
Role	Reliability
Classification	Required.

MISRA_12_3: Operator sizeof

Description	The sizeof operator should not be used on expressions that contain side effects.
Role	Reliability.
Classification	Required.

MISRA_12_4: Side effects

Description	The right hand operand of a *&&* or || operator shall not contain side effects.
Role	Reliability, Portability.
Classification	Required.

MISRA_12_5: Logical operand

Description	Operands of a logical *&&* and || shall be primary expressions.
Role	Reliability.
Classification	Required.

MISRA_12_7: Bitwise operations

Description	Bitwise operations (~, <<, >>, &, ^ and |) shall not be appliedto operands whose underlying type is signed.
Role	Reliability.
Classification	Required.

MISRA_12_8: Shift operator and right hand operand

Description	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.
Role	Reliability.
Classification	Required.

MISRA_12_9: Unary minus operator

Description	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
Role	Reliability.
Classification	Required.

MISRA_12_10: Comma operator

Description	The comma operator shall not be used.
Role	Reliability.
Classification	Required.

MISRA_12_12: Underlying representation of floating point numbers

Description	The underlying bit representation of floating point numbers shall not be used.
Role	Reliability, Portability.
Classification	Required.

MISRA_12_13: Do not mix increment and decrement with other operators

Description	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.
Role	Reliability.
Classification	Advisory.

MISRA_13_1: Test and assignment result

Description	Assignment operators shall not be used in expressions that yield a Boolean value.
	Example:
	`if (x = y) {  /* Violation */ }`
	
	`if ( (x = y) != 0 ) { /* Violation */ }`
	
	`x = y ;`
	`if (x != 0) { /* Correct */ }`

Role	Reliability.
Classification	Required.

MISRA_13_3: Test between floats

Description	Floating point variables shall not be tested for exact equality or inequality.
Role	Reliability.
Classification	Required.

MISRA_13_4: Loop counter

Description	The controlling expression of a for statement shall not contain any objects of floating type.
Role	Reliability.
Classification	Required.

MISRA_13_5: Loop control

Description	The three expressions of a for statement shall be concerned only with loop control.
Role	Reliability.
Classification	Required.

MISRA_13_6: Counter in for statements

Description	Numeric variables being used within a *for* loop for iteration counting should not be modified in the body of the loop.
Role	Reliability.

MISRA_14_1: Unreachable code

Description	There shall be no unreachable code.
Role	Reliability.
Classification	Required.

MISRA_14_2: Non-null statements

Description	Non-null statements shall have a side-effect.
Role	Reliability.
Classification	Required.

MISRA_14_3: Location of null statements

Description	Before preprocessing, a null statement shall only occur on a line by itself.
Role	Reliability.
Classification	Required.

MISRA_14_4: No goto statement

Description	The *goto* statement shall not be used.
Role	Maintainability.
Classification	Required.

MISRA_14_5: No continue statement

Description	The *continue* statement shall not be used.
Role	Maintainability.
Classification	Required.

MISRA_14_6: Break in loop

Description	For any iteration statement there shall be at most one *break* statement used for loop termination.
Role	Maintainability.
Classification	Required.

MISRA_14_7: Use of return

Description	A function shall have a single point of exit at the end of the function
Role	Maintainability.
Classification	Required.

MISRA_14_8: Use of braces

Description	The statement forming the body of a *switch, while, do ... while* or *fo*r statement shall be a compound statement
Role	Maintainability.
Classification	Required.

MISRA_14_9: If statement

Description	An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement
Role	Maintainability.
Classification	Required.

MISRA_14_10: Then and else

Description	All *if, else if* constructs shall be terminated with an *else* clause.
Role	Reliability.
Classification	Required.

MISRA_15_1: Use of switch labels

Description	A *switch* label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement.
Role	Maintainability.
Classification	Required.

MISRA_15_2: Break in switch

Description	An unconditional *break* statement shall terminate every non-empty switch clause.
Role	Reliability.
Classification	Required.

MISRA_15_3: Default in switch

Description	The final clause of a *switch* statement shall be the *default* clause.
Role	Reliability.
Classification	Required.

MISRA_15_4: Switch and boolean

Description	A switch expression shall not represent a value that is effectively Boolean.
Role	Maintainability.
Classification	Required.

MISRA_15_5: Switch without case

Description	Every *switch* statement shall have at least one *case* clause.
Role	Maintainability.
Classification	Required.

MISRA_16_1: No function with variable number of arguments

Description	Functions shall not be defined with variable numbers of arguments.

Role	Reliability, Maintainability
Classification	Required.
Note	Implemented using 2 complementary rule scripts.

MISRA_16_2: Recursion

Description	Functions shall not call themselves, either directly or indirectly.
Role	Reliability, Maintainability
Classification	Required.

MISRA_16_5: Functions with no parameters use explicit void

Description	Functions with no parameters shall be declared with parameter type void.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_16_6: Parameters

Description	The number of arguments passed to a function shall match the number of parameters.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_16_7: Function parameters and const

Description	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
Role	Reliability.
Classification	Advisory.

MISRA_16_8: Functions with non-void return types

Description	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
Role	Reliability.
Classification	Required.
Note	Implemented using 3 complementary rule scripts.

MISRA_16_9: Use of function identiers

Description	Afunction identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty.
Role	Reliability.
Classification	Required.

MISRA_17_3: Relational operators

Description	Relational operators shall not be applied to pointer types except where they point to the same array.
Role	Reliability.
Classification	Required.

MISRA_17_4: Pointer arithmetic only with array indexing

Description	Array indexing shall be the only allowed form of pointer arithmetic.
Role	Reliability.
Classification	Required.

MISRA_17_5: Reference complexity

Description	The declaration of objects should contain nom ore than 2 levels of pointer indirection.
Role	Maintainability.
Classification	Advisory.

MISRA_17_6: Address assignment

Description	The address of an object with automatic storage shall not be assigned to an object which may persist after the object has ceased to exit.
Role	Reliability.
Classification	Required.

MISRA_18_1: Members of structures and unions

Description	All structure or union types shall be complete at the end of a translation unit.
Role	Reliability.
Classification	Required.

MISRA_18_2: Variable storage

Description	An object shall not be assigned to an overlapping object.
Role	Reliability.
Classification	Required.

MISRA_18_4: Unions access

Description	Unions shall not be used.
Role	Reliability.
Classification	Required.

MISRA_19_1: Code structure

Description	#include statements in a file should only be preceded by other preprocessor directives or comments.

Role	Reliability.
Classification	Advisory.

MISRA_19_2: Non-standard characters

Description	Non-standard characters shall not occur in header file names in *#include* directive.
Role	Reliability.
Classification	Advisory.

MISRA_19_3: #include syntax

Description	The *#include* directive shall be followed by either a <filename> or “filename” sequence.
Role	Reliability.
Classification	Required.

MISRA_19_5: Define and undefine in a block

Description	Macros shall not be *#define'd* and *#undef'd* within a block.
Role	Reliability.
Classification	Required.

MISRA_19_6: Use of #undef

Description	#undef should not be used.
Role	Reliability.
Classification	Required.

MISRA_19_7: Functions and macros

Description	A function should be used in preference to a function-like macro.
Role	Reliability.
Classification	Advisory.

MISRA_19_8: Function-like macro call

Description	A function-like macro shall not be invoked without all of its arguments.
Role	Reliability.
Classification	Required.

MISRA_19_9: Arguments to function-like macros

Description	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.
Role	Reliability.
Classification	Required.

MISRA_19_10: Parentheses for macro occurences

Description	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.
Role	Reliability.
Classification	Required.
Note	Implemented using 2 complementary rule scripts.

MISRA_19_11: Identifiers in pre-processor directives

Description	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.
Role	Reliability.
Classification	Required.

M ISRA_19_12: Occurences of # and ## in macros

Description	There shall be at most one occurence of the # or ## pre-processor operators in a single macro definition.
Role	Reliability.
Classification	Required.

MISRA_19_13: # and ## preprocessor operators

Description	The # and ## preprocessor operators should not be used.
Role	Reliability.
Classification	Advisory.

MISRA_19_14: Two forms for defined pre-processor operator

Description	The defined preprocessor operator shall only be used in one of the two standard forms.
Role	Reliability.
Classification	Required.

MISRA_19_15: Header inclusion

Description	Precautions shall be taken in order to prevent the contents of a header file being included twice.
Role	Reliability, Portability.
Classification	Required.

MISRA_19_17: Pre-processor directives

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

MISRA_20_1: Define and undef standard names

Description	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
Role	Reliability, Maintainability.
Classification	Required.
Note	Implemented using 2 complementary rule scripts.

MISRA_20_2: Redefinition of standard library function names

Description	The names of standard library macros, objects and functions shall not be reused.
Role	Maintainability.
Classification	Required.

MISRA_20_4: Dynamic heap memory

Description	Dynamic heap memory allocation shall not be used.
Role	Reliability, Maintainability.
Classification	Required.

MISRA_20_5: Errno

Description	The error indicator *errno* shall not be used.
Role	Reliability.
Classification	Required.

MISRA_20_6: Offsetof

Description	The macro *offsetof*, in library <stddef.h> shall not be used.
Role	Reliability.
Classification	Required.

MISRA_20_7: Setjmp and longjmp

Description	The *setjmp* macro and the *longjmp* function shall not be used.
Role	Reliability.
Classification	Required.

MISRA_20_8: signal.h

Description	Signal handling facilities of <signal.h> shall not be used.
Role	Reliability.
Classification	Required.

MISRA_20_9: No <stdio.h> functions

Description	The input/ouput library <stdio.h> shall not be used in production code.
Role	Reliability.

Classification	Required.

MISRA_20_10: atof, atoi and atol

Description	Library functions *atof, atoi* and *atol* from library <stdlib.h> shall not be used.
Role	Reliability.
Classification	Required.

MISRA_20_11: abort, exit, getenv and system

Description	Library functions *abort*, *exit*, *getenv* and *system* from library <stdlib.h> shall not be used.
Role	Reliability.
Classification	Required.

MISRA_20_12: time.h

Description	Time handling functions of library <time.h> shall not be used.
Classification	Required.

Chapter 6

Customizing Standard Rules

Logiscope RuleChecker C 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 the next chapter.

6.1 Modifying the Rule Set File

A Rule Set file, with extension **".rst",** specifies the set of programming rules to be checked.

More information on how rule sets are used in Logiscope projects can be found in the *Logiscope RuleChecker & QualityChecker - Getting Started* manual.

The detailed syntax of the rule set file can be found in the Logiscope *RuleChecker & QualityChecker Basic Concepts* manual.

The Rule Set files should be in the following directories:

1. in <*log_installation_dir*>**/Ref/RuleSets/C/** where <*log_installation_dir*> is the Logiscope installation directory where default Rule Set files are available ,
2. in one of the directories in the environment variable LOG_RULE_ENV. The syntax of LOG_RULE_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.
   Directories in LOG_RULE_ENV should contain the subdirectory "**RuleSets/C**".

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 C.

6.2 Modifying Standard Rules

6.2.1 Rule File Location

Each rule must be stored in a Rule file (extension **".std"**).

The rule file should be placed in one of the following places:

1. in *log_installation_dir*/**Ref/Rules/C/** where *log_installation_dir* is the Logiscope installation directory
2. in one of the directories in the environment variable LOG_RULE_ENV (see Section 1.3 - Environment Variables. Directories in LOG_RULE_ENV should contain the subdiretory "**Rules/C**".

6.2.2 Rule File Syntax

A rule file is organized into fields following the syntax described below.

[.COMMENT *comment*]*
.DOMAIN [File | Application]
.KEY *key_of_rule*
.NAME *name_of_rule*
.SEVERITY *severity_of_rule*
.TITLE *title*
free_text]+
.COMMAND [**log_rchk_cc** | **r_perl_checker**]
.CODE
code_of_rule

where:

comment is a one-line character string,

key_of_rule is a printable character string, including no spaces, which identifies the rule,

name_of_rule is a one-line definition of the rule,

severity_of_rule is an string defining the level of severity of the rule,

title is a character string followed by a carriage return (,

free_text is plain text, which can be written over more than one line, provides a description of the rule,

log_rchk_cc: to activate the Logiscope Tcl Verifier if the rule Code is written in Tcl,

r_perl_checker: to activate the Perl Verifier if the rule Code is written in Perl,

code_of_rule is the code of the rule written in Tcl or Perl according to the Logiscope Verifier specified in the .COMMAND section.

Refer to the next chapter to more details on the Logiscope Tcl and Perl Verifiers.

Note1: *name_of_rule, severity_of_rule, title, free_text* fieds are not significant for *Logiscope RuleChecker C* on Windows.

Note2: **.DOMAIN** is no longer used by the checking mechanism which is now always performed on the full project.

Example of a Standard Rule

The Rule "*Identifiers must not start or end with the character "_"*," looks like this:

.COMMENT Naming_2_Underscore.rl

.DOMAIN `File`

.KEY `Naming_2_Underscore`

.SEVERITY 3

.NAME `It is illegal to use '_' character at the beginning or at the end of an identifier`

.TITLE `Description`
`Identifiers must not start or end with the character '_'`

.TITLE `Role`
`Makes code easier to read. For example, the 3 identifiers name, _name and name_ could easily be confused.`

.COMMAND `log_rchk_cc`

.CODE

```
proc noBeginOrEndUnderscore {identObj} {
    global thisRule
    set name [Get $identObj name]
    if { [string match _* $name] || [ string match *_ $name ] }
  {
      Violation $identObj $thisRule \
          "$name starts or finishes with character '_'."
    }
    return 1
  }
  # Running noBeginOrEndUnderscore on Symbol
  Maprole application symbol noBeginOrEndUnderscore
```

6.2.3 Creating a New Rule from a Standard Rule

For example, if the rule to be checked is
"*`It is illegal to use '%' character at the beginning or at the`*

end of an identifier”,

it can be written by changing the rule
“*It is illegal to use ’_’ character at the beginning or at the end of an identifier”*.

To do this change:

1. Duplicate the **.std** file containing the standard rule to be modified.
2. Use a text editor to edit this file.
3. Modify the .NAME field and write `It is illegal to use ’%’ character at the beginning or at the end of an identifier.`
4. Modify the relevant text fields.
5. Modify the .CODE field lines, replacing three ’_’ character occurrences by ’%’ character.
6. To improve the analysability of the rule, enter relevant information in the .KEY and .TITLE field lines.
7. Save the file.
8. Add description of the modified rule to the **.rst** file(s) the modified rule will belong to.
9. The new rule can now be loaded and be part of the rule list.

6.2.4 Renaming Rules

It is possible to rename standard rules to have as many versions of them as needed. The renamed rules have their own definition. Creating rules in this way 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.

The rule file format

A rule file containing a renamed rule description should be created. It should be nammed *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’s 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.

rename is the type of command used for this rule, and should not be changed.

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 Portability_1_C++Keywords rule):

```
.NAME No C++ keywords
.DESCRIPTION
In our standard no C++ keywords should be used.
.COMMAND rename Portability_1_C++Keywords
```

Activating the new rule

The new rule must be added to the Rule Set file (**.rst**) using the following syntax:

```
STANDARD new_std RENAMING old_std ON END STANDARD
```

where

`new_std` is the name of the rule being created.

`old_std` is the name of the existing rule.

Example:

```
STANDARD noC++ RENAMING Portability_1_CKeywords ON END STANDARD
```

6.2.5 Changing Rule Classification

It is possible to rename standard rules to have as many versions of them as needed. The renamed rules have their own definition. Creating rules in this way 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.

6.2.6 Changing Rule Severity

It is possible to rename standard rules to have as many versions of them as needed. The renamed rules have their own definition. Creating rules in this way 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.

Chapter 7

Developing New Rule Scripts

7.1 Introduction

Two verifiers are available in Logiscope *RuleChecker C*:

- the **Tcl verifier**: **log_rchk_cc**
- the **Perl verifier**: **r_perl_checker**

Apart from the different scripting languages used by these two verifiers, their purpose and inner working are very different: the **Tcl verifier** is based on a semantic data model that is akin to an abstract syntax tree that closely follows the C ISO standard. On the other hand, the **Perl verifier** is aimed to permit the lexical verification of the source code.

When using the **Tcl verifier**, macros are expanded and `#if` constructs taken into account.

When using the **Perl verifier**, macros are not expanded and `#if` constructs not taken into account.

Choosing the Right Verifier

Given the above characteristics, you will want to use the **Tcl verifier** when you need semantic and syntactical information to detect bad constructs, and the **Perl verifier** when you need the exact layout of the file content or that macros not be expanded.

This, of course, is a simplification, since you may as well open and scan the files directly from a **Tcl verifier** rule, and you can do the parsing from a **Perl verifier** rule. Thus the domains of application of these two verifiers indeed overlap; in these cases, the choice depends on which scripting language you feel the most comfortable with.

Examples:

Rule1: the `goto` instruction goto is forbidden.

There are two easy ways to check this rule:

- With the **Tcl verifier**, search for `InstructionGoto` objects.
- With the **Perl verifier**, search for the `\bgoto\b` pattern.

The results may be different: the **Tcl verifier** way will flag `goto` usage induced by

macro (macros defined in system include files included) expansion at the point of expansion of the macro, and `#ifdef`'ed out code will not be flagged; on the other hand, the **Perl verifier** will flag `goto` usage at the point the `goto` instruction appears in the code (for `goto`s in macros, at the point of definition).

Depending on the exact specification, and the compromises that are considered acceptable, one or the other solution may be choosen.

Rule2: goto labels begin at the start of a line.

Here we have a condition on the physical layout of a construct. The easiest way is to go with the **Perl verifier**, and check for the pattern `^(\s+)\w+\s*:`; if `$1` does not have zero length, this is a violation.

Rule3: only tabs may be used for indentation.

A code layout question: the **Perl verifier** is thus the best fit: search for the pattern `^\s*[ ]`.

Rule4: structure field identifiers are all lowercase.

A semantic question. The **Tcl verifier** is thus the best fit: search for `SymbolField` objects and check the conformance of their `name` attributes.

7.2 Using the Perl Verifier

The main support subroutines and variables used by the **Perl verifier** are the following:

`@cList`

The global array `@cList` contains the path names of all the files contained in the application: C files and header files found in `#include` directives, provided these paths do not match the `NoReportList` found in the file *procedures.tcl*.

This array may be used whenever it is useful to inspect the raw content of the files.

Example:

```
for my $pathName (@cList) {
open(C, "<$pathName") || warn "$pathName: cannot read: $!\n";
# Do something with the content of the file.
close(F);
```

`%TabPreprocessFile`

The global hash `%TabPreprocessFile` is indexed by the path names of the files of the application. The values are the contents of the files with backslash-newline sequences and comments removed, and string and character literals contents removed.

Line numbers are preserved.

These values are useful for searching for a pattern in the code without fearing that the pattern may appear in a comment or a string literal.

Beware that this is not preprocessing in the C sense.

Example:

```
# search for gotos
my $lineNumber = 1;
for my $pathName (keys %TabPreprocessFile) {
   my $content = $TabPreprocessFile{$pathName};
   while ($content =~ m{\bgoto\b}g) {
      # Do something.
   }
}
```

If the content of the source file is:

```
#include "a.h"
C90comment1 /*
                C90 comment
*/ C90comment2
C99comment1
// C99 comment
C99comment2
string1 "string" string2
char1 'char' char2
 # include   <b.h>
```

then the content of the corresponding value of `%TabPreprocessFile` is:

```
#include ""
C90comment1   C90comment2



C99comment1

C99comment2
string1 "" string2
```

```
char1 '' char2
 # include   <b.h>
```

Violation

The `Violation` subroutine emits a violation notice. It takes three parameters:

- the path name of the file for which a violation was detected,
- the line number of the file of the occurrence of the violation (use `0` to designate the whole file),
- a message string that is to be associated with this instance of violation (without new-lines)

The `Violation` subroutine takes care of adding the rule `.KEY` to the violation report.

Preprocessor

The `PreProcessor` subroutine processes a string in the manner of the values of the hash `TabPreprocessFile`. Use it to get the same result as a value of `%TabPreprocessFile` for a file that is not in the application.

Example:

```
my $prepro = &PreProcessor($rawText);
```

7.3 Using the Tcl Verifier

Commands described below will let define personal programming rules.

There are three types of TCL Verifier commands:

- Access commands to data about elements in the application code (its internal representation is produced as per the data model described in Chapter 2).
- Commands to check progress reports.
- Debugging aid commands.

Tcl language [TCL94] typographical conventions are used for command syntax.

Examples below show how the data model is used by checker commands.

Naming and identifying

Any data model object is identifiable.

Any objects that can be designated by a key in the source code can be named. The absolute name can be broken down as per its access path:

Example:

- void f()
- {
  - int i;
  - i = 2;
- }

The instruction `i=2` cannot be named, but it can be identified. The variable path `f/i`, can be named and identified.

The application pseudo-object

All data model abstract classes can be scanned from the application pseudo-object.

7.3.1 Access commands

Access to the class attribute

Class*object*

Returns the name of the class of *object*. An error is reported if *object* is not a valid key.

Access to other attributes

Get *object attribute*

Returns the value of attributes of *object* designated by *attribute*. An error is reported if *attribute* does not designate an attribute of *object* or if *object* is not a valid key.

Access to a single cardinality role

GetRole *source_object target_role*

Applies to associations whose target class has cardinality 0 or 1( ).

Returns the key of the object which has the *target_role* in one of the associations of *source_object*, or an empty string if there are no such associations. An error is reported if *source_object* has no association with *target_role* as a role.

Access to a multiple cardinality role

MapRole *source_object target_role* -filter *fscript script*

fscript and script represent a sequence of commands.

Applies to associations whose cardinality is greater than or equal to 0( ).

It scans objects associated with the *source_object* which have *target_role* as a role.

For each object which is the *target_role* in one of the associations of *source_object*, the *fscript* command sequence is evaluated:

- if *fscript* returns a value greater than 0, the *script* sequence is evaluated,
- if *fscript* returns a value equal to 0, the *script* sequence is not evaluated.

If *fscript* is not present, *script* is always evaluated.

If *script* returns a value equal to 0, the MapRole command stops immediately.

At each evaluation, *fscript* and *script* receive as a parameter the identifier of the object to process.

The MapRole command returns the number of times *script* has been evaluated. This number represents the overall number of objects which have *target_role* as a role in one of the associations of *source_object* or, if a filter is specified, it represents the number of objects that match the filtering condition. An error is reported:

- if *source_object* has no association with, as a role, a target object: *target_role*,
- if *fscript* and *script* end with an uncontrolled value.

7.3.2 Report commands

Internal error display

- Internal Error *message*

message **is a character string between quotes (“”).**

Errors detected during checking are reported. The message entered as a parameter is sent as the error message.

Rule violation display

- Violation *object rule message*

message **is a character string between quotes (“”).**

Reports a rule violation identified by *rule* and located by *object*. The optional `message` parameter lets add specific information about the violation.

If a rule violation cannot be located (for example, if a limited number of files is exceeded in an application), the value of *object* is **application**.

7.3.3 Debugging aid commands

Roles of a class

- Roles Of *object*

Returns the role list for the class of which *object* is an instance.

Attributes of a class

- Attributes Of *object*

Returns the attribute list for the class of which *object* is an instance.

7.4 Using *RuleChecker* Libraries

Tcl Rules

Some functions used more than once in the code of rules can be stored in a specific file called **procedures.tcl**. This file is loaded at the beginning of a Logiscope *RuleChecker C* session. The user can write and add personal global functions to this file.

This file is searched in the following locations and in the following order:

1. in the *Rule*Checker startup directory,
2. in the <*log_install_dir*>/**util** directory.

Perl rules

Some functions used more than once in the code of rules are stored in a specific file called **r_perl_checker.perl**. This file is used to check Perl rules. The user can write and add personalized global functions to this file.

This file is sought in the <*log_install_dir*>/**util** directory.

Chapter 8

Logiscope C Data Model

8.1 Introduction

The Logiscope C data model is the result of C language modelization in a class diagram. Each time a Logiscope C project is analyzed, Logiscope *RuleChecker C* instantiates this data model with information found in C source files of the project.

The Logiscope C data model is then questionned by the Logiscope Tcl Verifier to locate and report all violations of the programming rules selected in the Rules Set files based on the Tcl code specified in each of the corresponding Rule files.

For more details on how to use the Logiscope C data model and the RuleChecker Tcl Verifier, please refer to the *IBM Rational Logiscope - Writing C Rules Using RuleChecker Tcl Verifier* advanced guide.

The next section explains symbols used in the data model representation. Then, the data model itself is specified, first in its graphic form, then in text format.

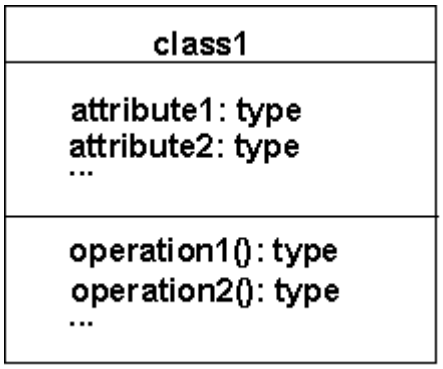
8.2 Concepts and Symbolism

The data model is represented as a class diagram.

Here is the definition and representation of object-oriented concepts appearing in the graphic form of the data model.

8.2.1 Class

A class is a set of objects with similar properties (attributes), common behaviors (operations) and share relations with other objects.



8.2.2 Attribute

An attribute is a data item specific to objects of a given class. Each attribute name is unique in its class. Each attribute has a value of the specified type (string, integer, etc.) for every object instance.

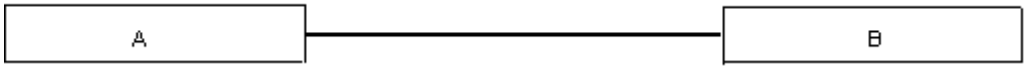
8.2.3 Operation

An operation is a function or transformation that can be applied to objects of a class or carried out by them. All of the objects in a given class share the same operations. The type associated with an operation indicates the type of value returned by the operation.

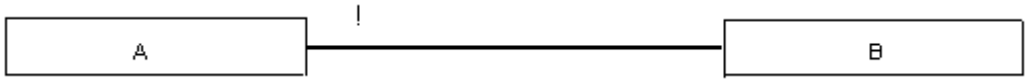
8.2.4 Link and association

A link is a physical or conceptual connection between two instances of an object:

- A-to-B link and B-to-A link:



- A-to-B link only (the origin side of the link is indicated by the exclamation point!):

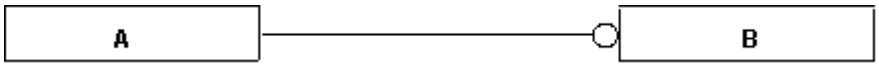


An association describes a set of links, just as a class describes a set of objects.

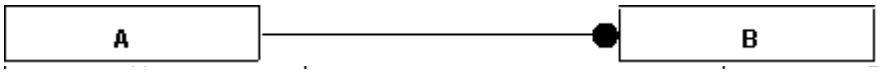
8.2.5 Multiplicity

The multiplicity specifies how many instances of a class are related to an instance of the associated class. Multiplicity (or cardinality) can be a range of values, a set of values or a specific number.

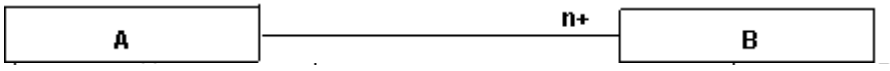
- 1 instance of *A* is linked to 0 or 1 instance of *B*:



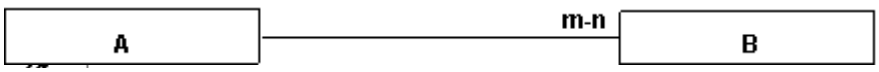
- 1 instance of A is linked to 0 or more instances of B:



- 1 instance of *A* is linked to at least *n* instances of *B* ($n > 0$):

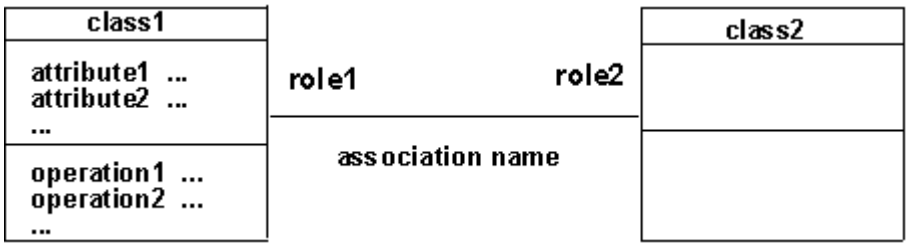


- 1 instance of *A* is linked to a number of instances of *B* between *m* and *n* inclusive:



8.2.6 Role

A role is one end of an association. A binary association has two roles, each with its own name. The name of a role is a name which clearly identifies one end of an association. Roles make possible to consider a binary association as the link of one object to an associated set of objects. Each role in a binary association identifies an object or set of objects associated with an object at the other end.



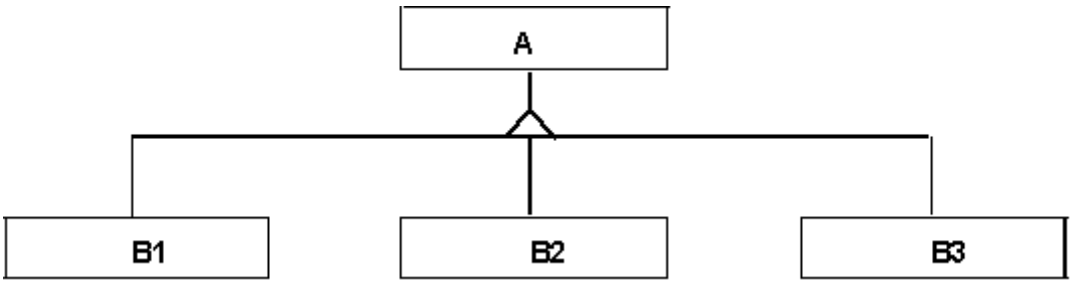
The name of a role is a derivative attribute whose value is a set of associated objects. There are two cases for which roles must absolutely be named:

- recursive associations,
- several associations involving the same classes.

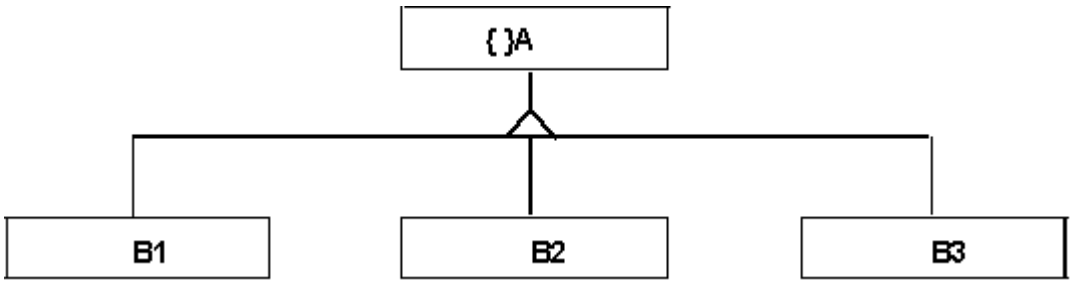
If roles are not named, the class name is taken as the role name, with the first letter changed to lower-case.

8.2.7 Inheritance

The “is a”, “kind of” relation allows classes to share similarities and retain their differences.



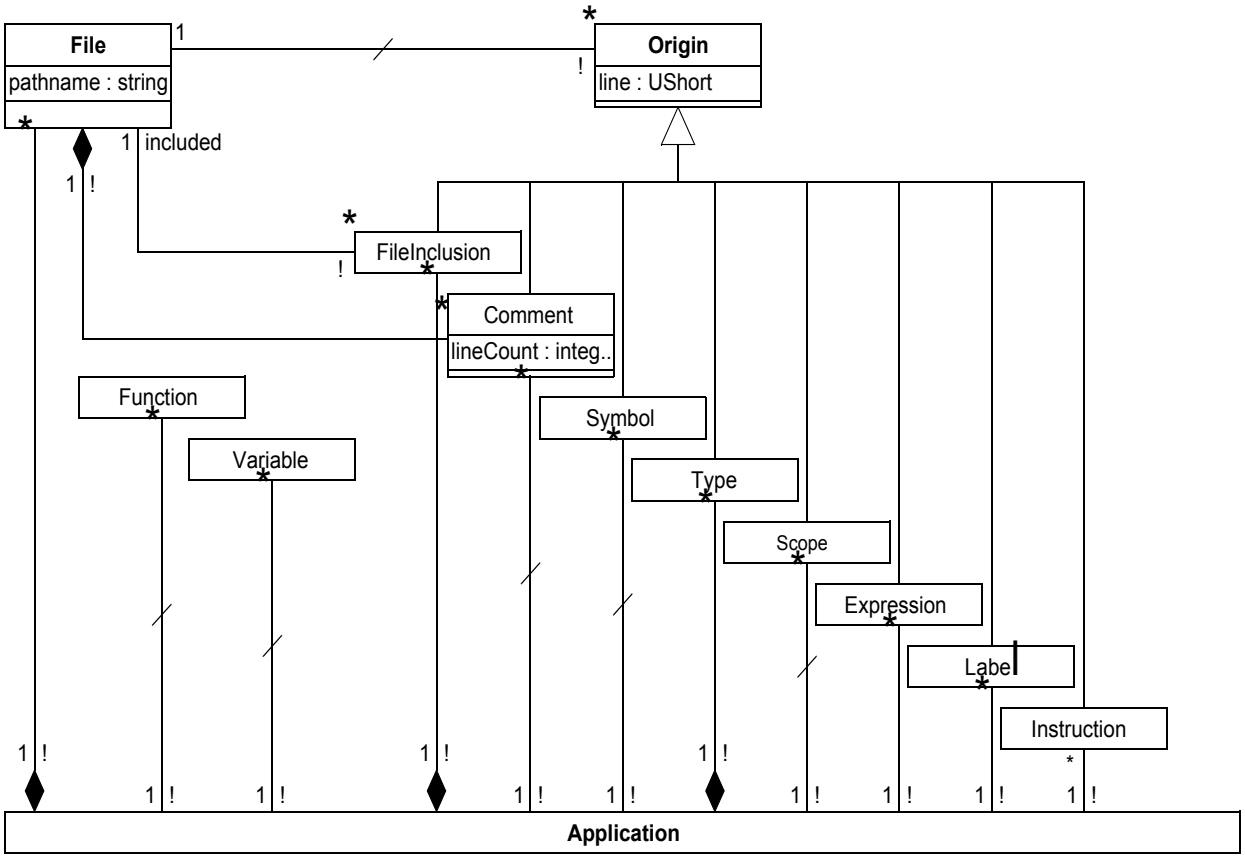
8.2.8 Abstract class

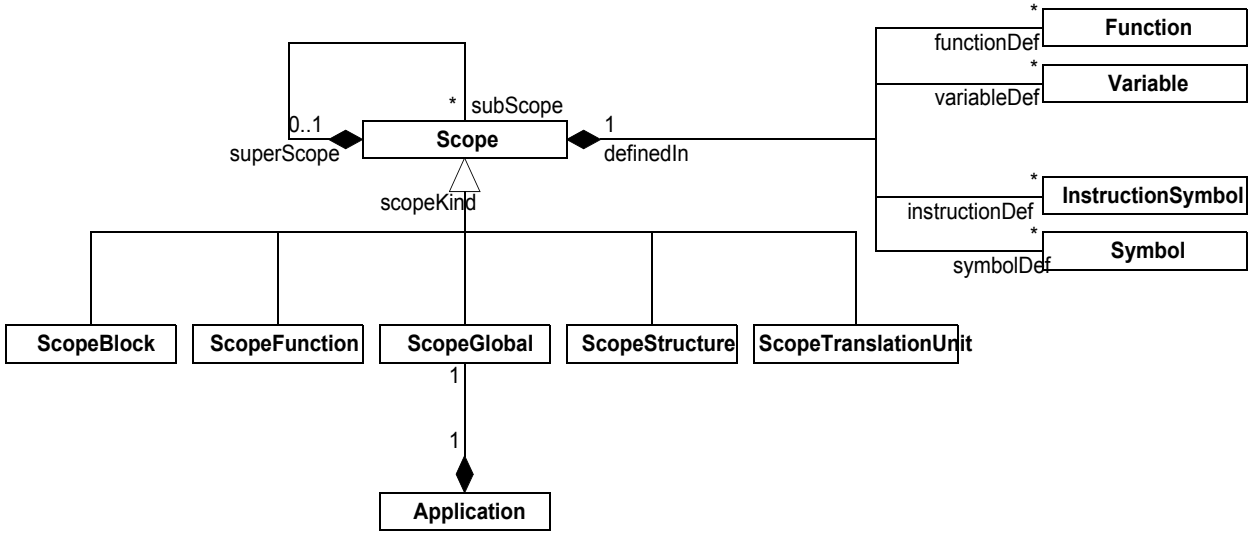


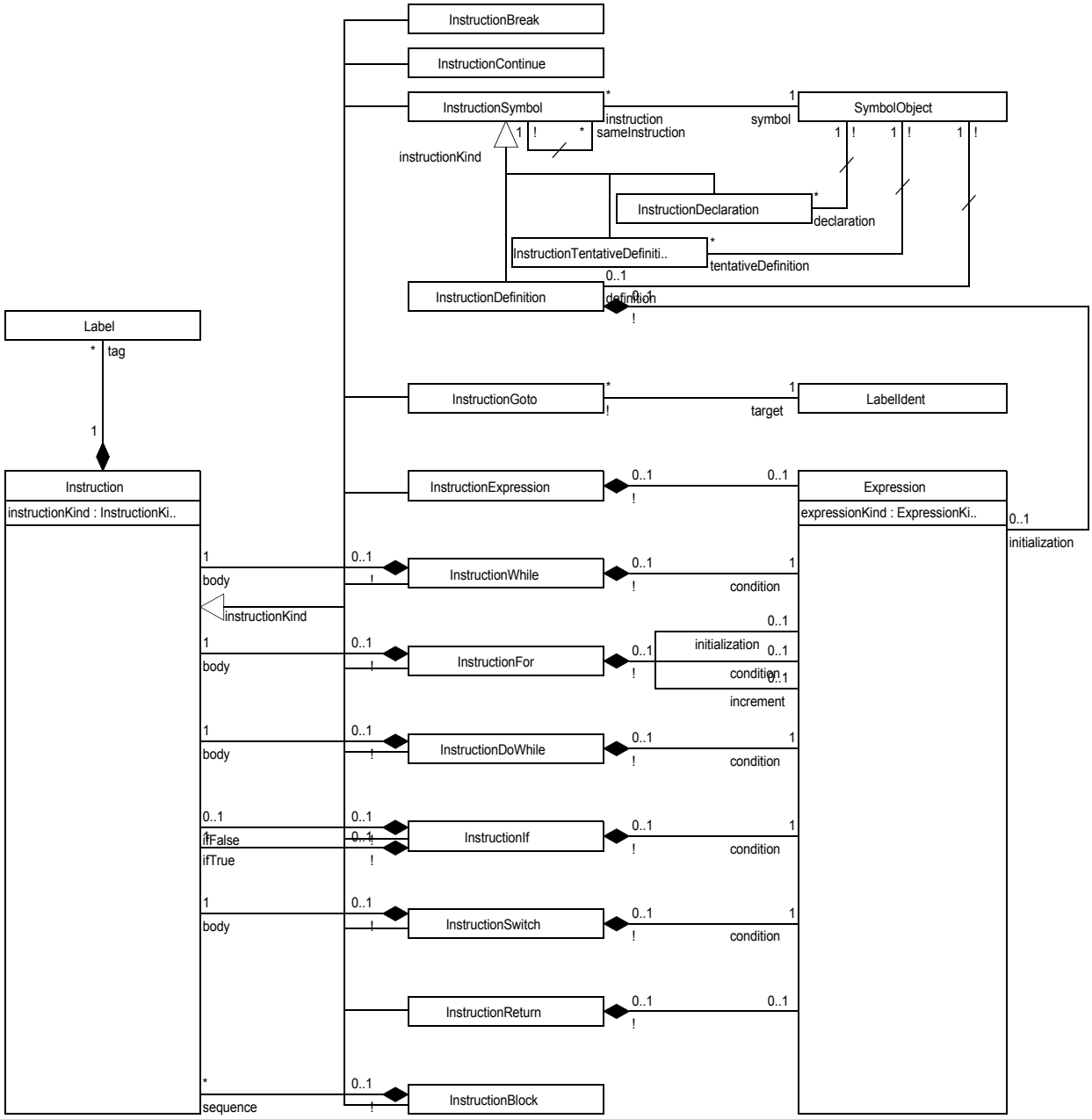
An abstract class is a class with no instantiated objects. Attributes and operations it describes are inherited by its sub-classes.

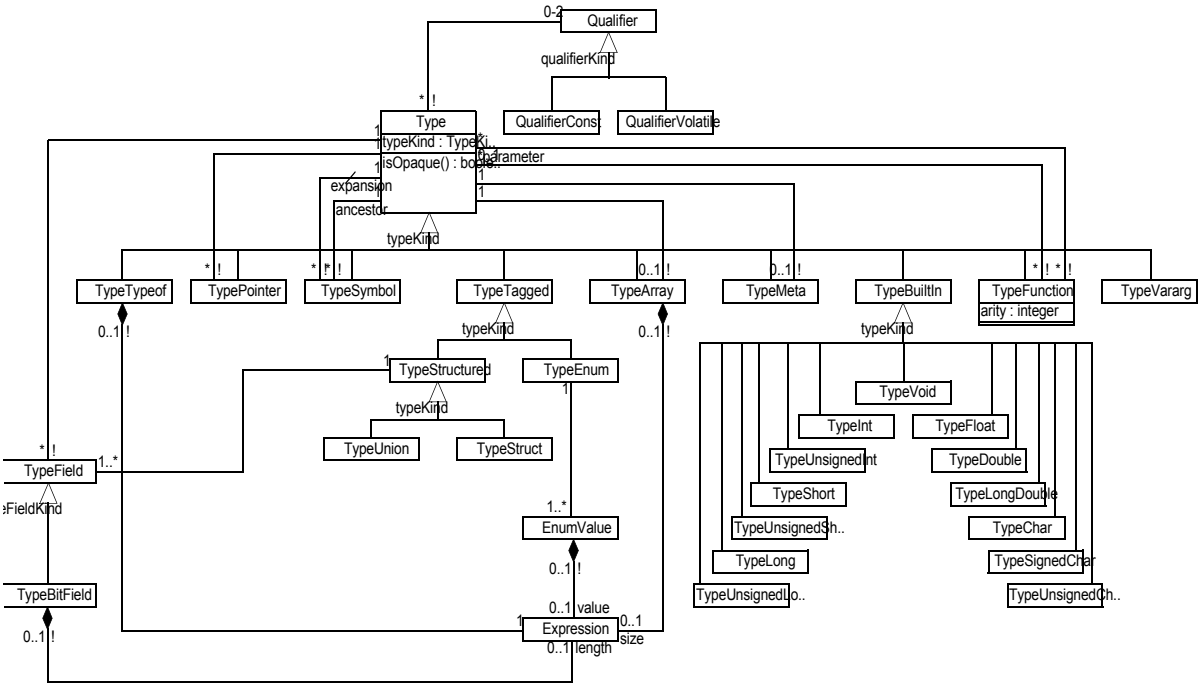
8.3 The data model

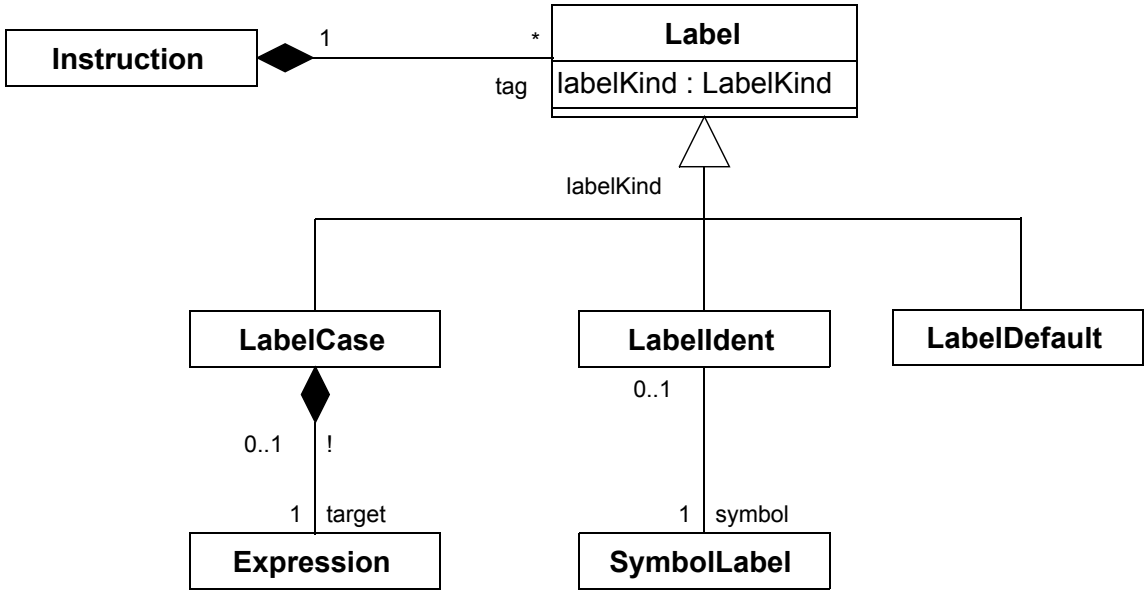
8.3.1 Graphic Representation

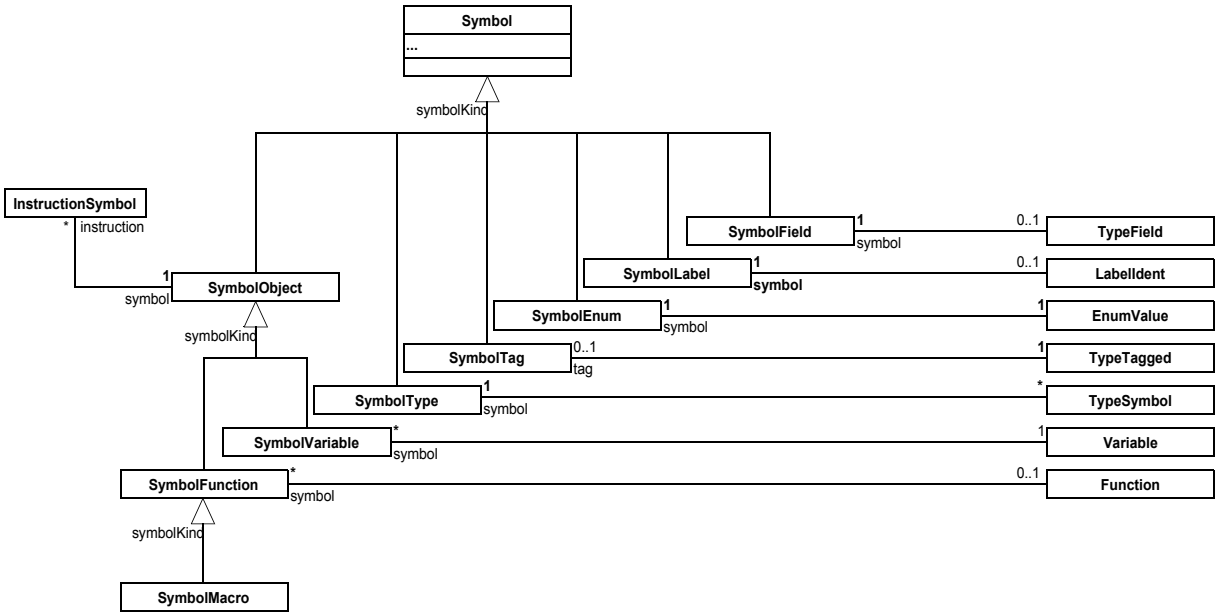


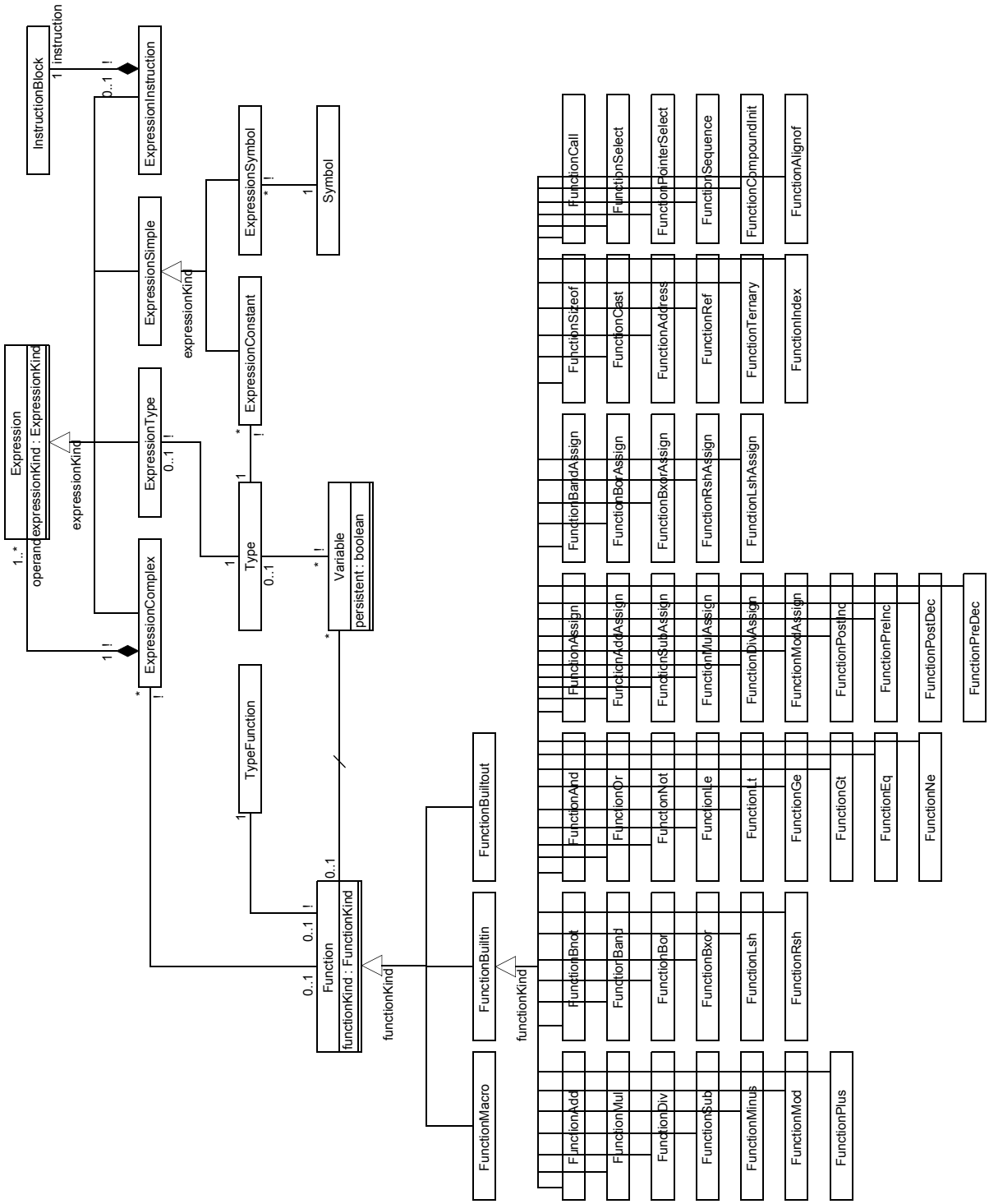


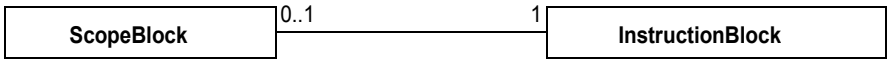

InstructionBreak
InstructionContinue
InstructionSymbol
SymbolObject
instruction
sameInstruction
symbol
instructionKind
InstructionDeclaration
declaration
InstructionTentativeDefiniti..
tentativeDefinition
InstructionDefinition
definition
Label
tag
InstructionGoto
target
LabelIdent
Instruction
instructionKind : InstructionKi..
InstructionExpression
Expression
expressionKind : ExpressionKi..
initialization
body
InstructionWhile
condition
InstructionFor
increment
InstructionDoWhile
InstructionIf
ifFalse
ifTrue
InstructionSwitch
InstructionReturn
sequence
InstructionBlock

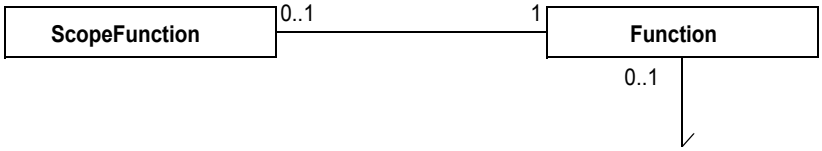

Qualifier
qualifierKind
Type
QualifierConst
QualifierVolatile
typeKind : TypeKi..
isOpaque() : boole..
parameter
expansion
ancestor
typeKind
TypeTypeof
TypePointer
TypeSymbol
TypeTagged
TypeArray
TypeMeta
TypeBuiltIn
TypeFunction
arity : integer
TypeVararg
typeKind
TypeStructured
TypeEnum
typeKind
TypeUnion
TypeStruct
TypeField
EnumValue
TypeBitField
value
Expression
length
size
typeKind
TypeVoid
TypeInt
TypeFloat
TypeUnsignedInt
TypeDouble
TypeShort
TypeLongDouble
TypeUnsignedSh..
TypeChar
TypeLong
TypeSignedChar
TypeUnsignedLo..
TypeUnsignedCh..


Instruction
1
*
Label
labelKind : LabelKind
tag
labelKind
LabelCase
LabelIdent
LabelDefault
0..1
!
1
target
0..1
1
symbol
Expression
SymbolLabel

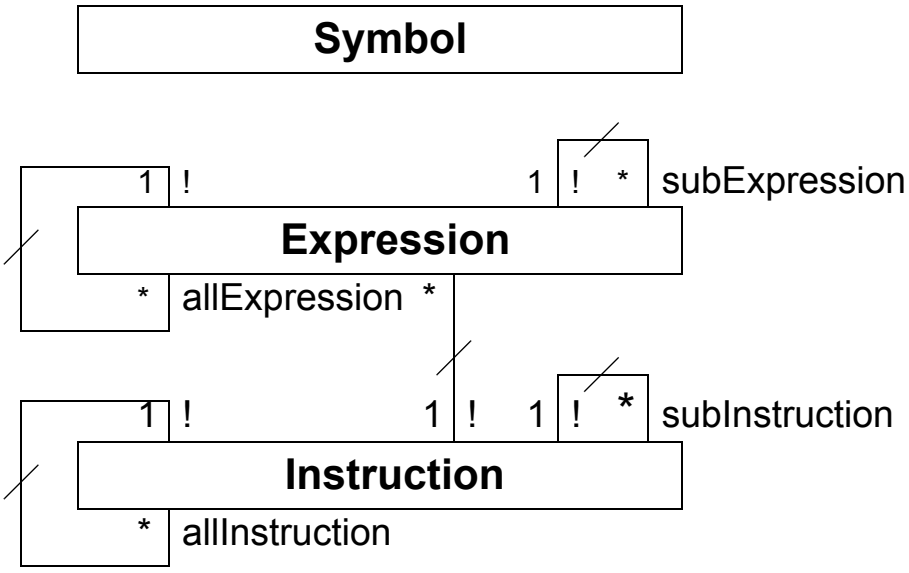

Symbol
...
symbolKind
InstructionSymbol
* instruction
1
symbol
SymbolObject
symbolKind
SymbolField
1
symbol
0..1
TypeField
SymbolLabel
1
symbol
0..1
LabelIdent
SymbolEnum
1
symbol
1
EnumValue
SymbolTag
0..1
tag
1
TypeTagged
SymbolType
1
symbol
*
TypeSymbol
SymbolVariable
*
symbol
1
Variable
SymbolFunction
*
symbol
0..1
Function
symbolKind
SymbolMacro

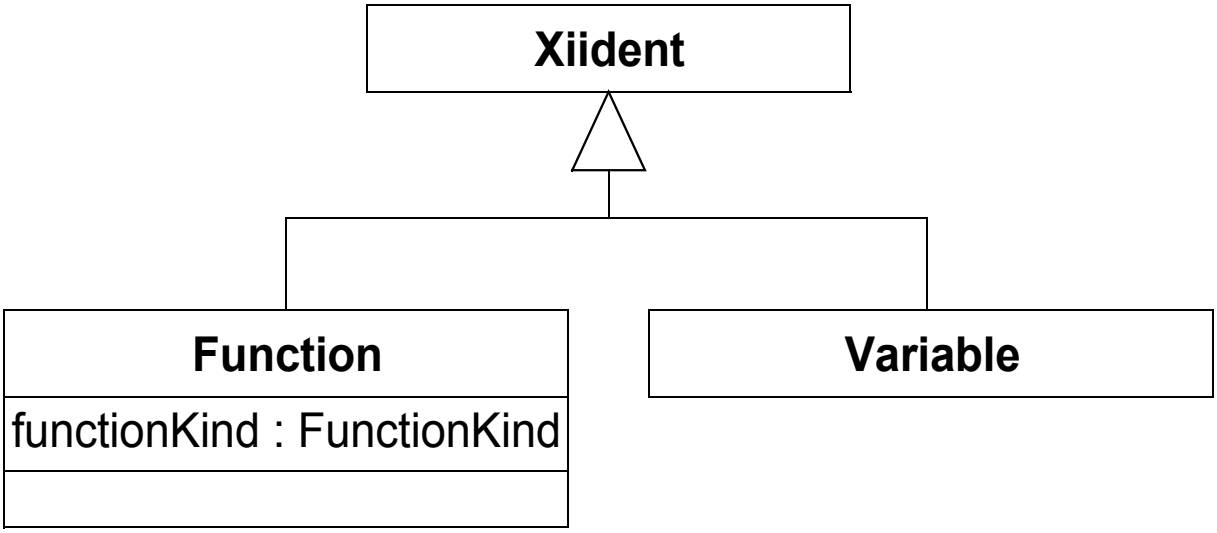

InstructionBlock
instruction
ExpressionInstruction
Expression
expressionKind : ExpressionKind
operand
expressionKind
ExpressionSimple
ExpressionType
ExpressionComplex
expressionKind
ExpressionSymbol
ExpressionConstant
Symbol
Type
Variable
persistent : boolean
TypeFunction
Function
functionKind : FunctionKind
functionKind
FunctionBuiltout
FunctionBuiltin
FunctionMacro
functionKind
FunctionCall
FunctionSelect
FunctionPointerSelect
FunctionSequence
FunctionCompoundInit
FunctionAlignof
FunctionSizeof
FunctionCast
FunctionAddress
FunctionRef
FunctionTernary
FunctionIndex
FunctionBandAssign
FunctionBorAssign
FunctionBxorAssign
FunctionRshAssign
FunctionLshAssign
FunctionAssign
FunctionAddAssign
FunctionSubAssign
FunctionMulAssign
FunctionDivAssign
FunctionModAssign
FunctionPostInc
FunctionPreInc
FunctionPostDec
FunctionPreDec
FunctionAnd
FunctionOr
FunctionNot
FunctionLe
FunctionLt
FunctionGe
FunctionGt
FunctionEq
FunctionNe
FunctionBnot
FunctionBand
FunctionBor
FunctionBxor
FunctionLsh
FunctionRsh
FunctionAdd
FunctionMul
FunctionDiv
FunctionSub
FunctionMinus
FunctionMod
FunctionPlus


ScopeBlock
0..1
1
InstructionBlock


ScopeFunction
0..1
1
Function
0..1


1
!
Scope
*
allScope


Symbol
1
!
1
!
*
subExpression
Expression
*
allExpression
*
1
!
1
!
1
!
*
subInstruction
Instruction
*
allInstruction


Xiident
Function
functionKind : FunctionKind
Variable

8.3.2 Text presentation

The data model is presented class by class. Classes appear in alphabetical order.

For each class, existing associations and attributes are listed in the following format:

class_name class

Associations with:

target_class_nametarget_role number_instances_target_class

Attributes:

attribute_name

Application class

Associations with:

```
Comment  comment n
Expression  expression n
File  file n
FileInclusion  fileInclusion n
Function  function n
Instruction  instruction n
Label  label n
Scope  scope n
ScopeGlobal  scopeGlobal 1
Symbol  symbol n
Type type n
Variable variable n
```

Comment class

Associations with:

```
File file 1
```

Attributes:

```
line
lineCount
```

EnumValue class

Associations with:

```
Expression value 1
SymbolEnum symbol 1
TypeEnum typeEnum 1
```

Expression class

Associations with:

```
Expression allExpression n
Expression subExpression n
File file 1
```

Attributes:

```
line
```

ExpressionComplex class

Associations with:

```
Expression allExpression n
Expression operand n
Expression subExpression n
File file 1
Function function 1
```

Attributes:

```
line
```

ExpressionConstant class

Associations with:

```
Expression allExpression n
Expression subExpression n
File file 1
Type type 1
```

Attributes:

```
line
value
```

ExpressionInstruction class

Associations with:

```
Expression allExpression n
Expression subExpression n
File file 1
InstructionBlock instruction 1
```

Attributes:

```
line
```

ExpressionSimple class

Associations with:

```
Expression allExpression n
Expression subExpression n
File file 1
```

Attributes:

```
line
```

ExpressionSymbol class

Associations with:

```
Expression allExpression n
Expression SubExpression n
File file 1
Symbol symbol 1
```

Attributes:

```
line
```

ExpressionType class

Associations with:

```
Expression allExpression n
Expression subExpression n
File file 1
Type type 1
```

Attributes:

```
line
```

File class

Associations with:

```
Comment comment n
```

Attributes:

```
pathname
```

FileInclusion class

Associations with:

```
File file 1
File included 1
```

Attributes:

```
line
```

Function class

Associations with:

```
Scope definedIn 1
ScopeFunction scopeFunction 1
SymbolFunction symbol n
TypeFunction typeFunction 1
Variable variable n
```

The list of roles of the abstract class Functions applies for all its sub-classes:

```
FunctionAdd, FunctionAddAssign, FunctionAddress,
```

```
FunctionAlignof, FunctionAnd, FunctionAssign,
FunctionBand, FunctionBandAssign, FunctionBnot,
FunctionBor, FunctionBorAssign, FunctionBuiltin,
FunctionBuiltout, FunctionBxor, FunctionBxorAssign,
FunctionCall, FunctionCast, FunctionCompoundInit,
FunctionDiv, FunctionDivAssign, FunctionEq, FunctionGe,
FunctionGt, FunctionIndex, FunctionLe, FunctionLsh,
FunctionLshAssign, FunctionLt, FunctionMacro,
FunctionMinus, FunctionMod, FunctionModAssign,
FunctionMul, FunctionMulAssign, FunctionNe, FunctionNot,
FunctionOr, FunctionPlus, FunctionPointerSelect,
FunctionPostDec, FunctionPostInc, FunctionPreDec,
FunctionPreInc, FunctionRef, FunctionRsh,
FunctionRshAssign, FunctionSelect, FunctionSequence,
FunctionSizeof, FunctionSub, FunctionSubAssign,
FunctionTernary.
```

Instruction class

Associations with:

```
Expression expression n
File file 1
Instruction allInstruction n
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionBlock class

Associations with:

```
Expression expression n
File file 1
Instruction allInstruction n
ScopeBlock scopeBlock 1
Instruction sequence n
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionBreak class

Associations with:

```
Expression expression n
File file 1
Instruction allInstruction n
```

```
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionContinue class

Associations with:

```
Expression expression n
File file 1
Instruction allInstruction n
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionDeclaration class

Associations with:

```
Expression expression n
File file 1
Instruction allInstruction n
Instruction subInstruction n
Label tag n
Scope definedIn 1
SymbolObject symbol 1
```

Attributes:

```
line
```

InstructionDefinition class

Associations with:

```
Expression expression n
Expression initialization 1
File file 1
Instruction allInstruction n
Instruction subInstruction n
Label tag n
Scope definedIn 1
SymbolObject symbol 1
```

Attributes:

```
line
```

InstructionDoWhile class

Associations with:

```
Expression condition 1
Expression expression n
File file 1
Instruction allInstruction n
Instruction body 1
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionExpression class

Associations with:

```
Expression expression n
Expression expression 1
File file 1
Instruction allInstruction n
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionFor class

Associations with:

```
Expression condition 1
Expression expression n
Expression increment 1
Expression initialization 1
File file 1
Instruction allInstruction n
Instruction body 1
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionGoto class

Associations with:

```
Expression expression n
File file 1
Instruction allInstruction n
Instruction subInstruction n
LabelIdent target 1
Label tag n
```

Attributes:

```
line
```

InstructionIf class

Associations with:

```
Expression condition 1
Expression expression n
File file 1
Instruction allInstruction n
Instruction ifFalse 1
Instruction ifTrue 1
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionReturn class

Associations with:

```
Expression expression n
Expression expression 1
File file 1
Instruction allInstruction n
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionSwitch class

Associations with:

```
Expression condition 1
Expression expression n
File file 1
Instruction allInstruction n
Instruction body 1
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

InstructionSymbol class

Associations with:

```
Expression expression n
File file 1
Instruction allInstruction n
```

```
Instruction subInstruction n
Label tag n
Scope definedIn 1
SymbolObject symbol 1
```

Attributes:

```
line
```

InstructionTentativeDefinition class

Associations with:

```
Expression expression n
File file 1
Instruction allInstruction n
Instruction subInstruction n
Label tag n
Scope definedIn 1
SymbolObject symbol 1
```

Attributes:

```
line
```

InstructionWhile class

Associations with:

```
Expression condition 1
Expression expression n
File file 1
Instruction allInstruction n
Instruction body 1
Instruction subInstruction n
Label tag n
```

Attributes:

```
line
```

Label class

Associations with:

```
File file 1
Instruction instruction 1
```

Attributes:

```
line
```

LabelCase class

Associations with:

```
Expression target 1
File file 1
Instruction instruction 1
```

Attributes:

```
line
```

LabelDefault class

Associations with:

```
File file 1
Instruction instruction 1
```

Attributes:

```
line
```

LabelIdent class

Associations with:

```
File file 1
Instruction instruction 1
SymbolLabel symbol 1
```

Attributes:

```
line
```

Origin class

Associations with:

```
File file 1
```

Attributes:

```
line
```

Scope class

Associations with:

```
File file 1
Function functionDef n
InstructionSymbol instructionDef n
Scope allScope n
Scope subScope n
Scope superScope 1
Symbol symbolDef n
Type typeDef n
Variable variableDef n
```

Attributes:

```
line
```

ScopeBlock class

Associations with:

```
File file 1
Function functionDef n
InstructionBlock instructionBlock 1
InstructionSymbol instructionDef n
Scope allScope n
Scope subScope n
Scope superScope 1
Symbol symbolDef n
Type typeDef n
Variable variableDef n
```

Attributes:

```
line
```

ScopeFunction class

Associations with:

```
File file 1
Function function 1
Function functionDef n
InstructionSymbol instructionDef n
Scope allScope n
Scope subScope n
Scope superScope 1
Symbol symbolDef n
Variable variableDef n
```

Attributes:

```
line
```

ScopeGlobal class

Associations with:

```
Application application 1
File file 1
Function functionDef n
InstructionSymbol instructionDef n
Scope allScope n
Scope subScope n
Scope superScope 1
Symbol symbolDef n
Variable variableDef n
```

Attributes:

```
line
```

ScopeStructure class

Associations with:

```
File file 1
Function functionDef n
```

```
InstructionSymbol instructionDef n
Scope allScope n
Scope subScope n
Scope superScope 1
Symbol symbolDef n
TypeStructured typeStructured 1
Variable variableDef n
```

Attributes:

```
line
```

ScopeTranslation class

Associations with:

```
File file 1
Function functionDef n
InstructionSymbol instructionDef n
Scope allScope n
Scope subScope n
Scope superScope 1
Symbol symbolDef n
Variable variableDef n
```

Attributes:

```
line
```

Symbol class

Associations with:

```
File file 1
Scope definedIn 1
```

Attributes:

```
line
name
```

SymbolEnum class

Associations with:

```
EnumValue enumValue 1
File file 1
Scope definedIn 1
```

Attributes:

```
line
name
```

SymbolField class

Associations with:

```
File file 1
Scope definedIn 1
TypeField typeField 1
```

Attributes:

```
line
name
```

SymbolFunction class

Associations with:

```
File file 1
Function function 1
InstructionDeclaration declaration n
InstructionDefinition definition 1
InstructionSymbol instruction n
InstructionTentativeDefinition tentativeDefinition n
Scope definedIn 1
```

Attributes:

```
line
name
```

SymbolLabel class

Associations with:

```
File file 1
LabelIdent labelIdent 1
Scope definedIn 1
```

Attributes:

```
line
name
```

SymbolMacro class

Associations with:

```
File file 1
Function function 1
InstructionDeclaration declaration n
InstructionDefinition definition 1
InstructionSymbol instruction n
InstructionTentativeDefinition tentativeDefinition n
Scope definedIn 1
```

Attributes:

```
line
name
```

SymbolObject class

Associations with:

```
Definition Scope definedIn 1
File file 1
Function function 1
InstructionDeclaration declaration n
InstructionDefinition definition 1
InstructionSymbol instruction n
InstructionTentativeDefinition tentative
```

Attributes:

```
line
name
```

SymbolTag class

Associations with:

```
File file 1
Scope definedIn 1
TypeTagged typeTagged 1
```

Attributes:

```
line
name
```

SymbolType class

Associations with:

```
File file 1
Scope definedIn 1
TypeSymbol typeSymbol 1
```

Attributes:

```
line
name
```

SymbolVariable class

Associations with:

```
File file 1
InstructionDeclaration declaration n
InstructionDefinition definition 1
InstructionSymbol instruction n
InstructionTentativeDefinition tentativeDefinition n
Scope definedIn 1
Variable variable 1
```

Attributes:

```
line
name
```

Type class

Associations with:

```
File file 1
Qualifier qualifier n
```

Attributes:

```
line
```

TypeArray class

Associations with:

```
Expression size 1
File file 1
Qualifier qualifier n
Type type 1
```

Attributes:

```
line
```

TypeBitField class

Associations with:

```
Expression length 1
SymbolField symbol 1
Type type 1
TypeStructured typeStructured 1
```

TypeBuiltIn class

Associations with:

```
File file 1
Qualifier qualifier n
```

Attributes:

```
line
```

The lists of roles and attributes of the abstract class `TypeBuiltIn` apply to all its sub-classes:

```
TypeChar, TypeDouble, TypeFloat, TypeInt,
TypeLong, TypeLongDouble, TypeShort,
TypeSignedChar, TypeUnsignedChar,
TypeUnsignedInt, TypeUnsignedLong,
TypeUnsignedShort, TypeVararg, TypeVoid.
```

TypeEnum class

Associations with:

```
EnumValue enumValue n
File file 1
```

```
Qualifier qualifier n
SymbolTag tag 1
```

Attributes:

```
line
```

TypeField class

Associations with:

```
SymbolField symbol 1
Type type 1
TypeStructured typeStructured 1
```

TypeFunction class

Associations with:

```
File file 1
Qualifier qualifier n
Type parameter n
Type type 1
```

Attributes:

```
arity
line
```

TypeMeta class

Associations with:

```
File file 1
Qualifier qualifier n
Type type 1
```

Attributes:

```
line
```

TypeOf class

Associations with:

```
Expression expression 1
File file 1
Qualifier qualifier n
Type type 1
```

Attributes:

```
line
```

TypePointer class

Associations with:

```
File file 1
Qualifier qualifier n
Scope definedIn 1
Type type 1
```

Attributes:

```
line
```

TypeStructured class

Associations with:

```
File file 1
Qualifier qualifier n
ScopeStructure scopeStructure 1
SymbolTag tag 1
TypeField typeField n
```

Attributes:

```
line
```

The lists of roles and attributes of the abstract class `TypeStructured` apply for all its sub-classes: `TypeStruct, TypeUnion`.

TypeSymbol class

Associations with:

```
File file 1
Qualifier qualifier n
SymbolType symbol 1
Type ancestor 1
Type expansion 1
```

Attributes:

```
line
```

TypeTagged class

Associations with:

```
File file 1
Qualifier qualifier n
SymbolTag tag 1
```

Attributes:

```
line
```

TypeVararg class

Associations with:

```
File file 1
Qualifier qualifier n
Scope definedIn1
```

Attributes:

```
line
```

Variable class

Associations with:

```
Function function 1
Scope definedIn 1
SymbolVariable symbol n
Type type 1
```

Notices

U.S. Government Users Restricted Rights - Use, duplication, or disclosure restricted by GSA ADP Schedule Contract with IBM corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM 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-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM 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.

Trademarks

IBM, the IBM logo, ibm.com are trademarks or registered trademarks of International Business Machine Corp., registered in many jurisdictions worldwide. Other product and services names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at:

www.ibm.com/legal/copytrade.html.

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 trademarks, 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.