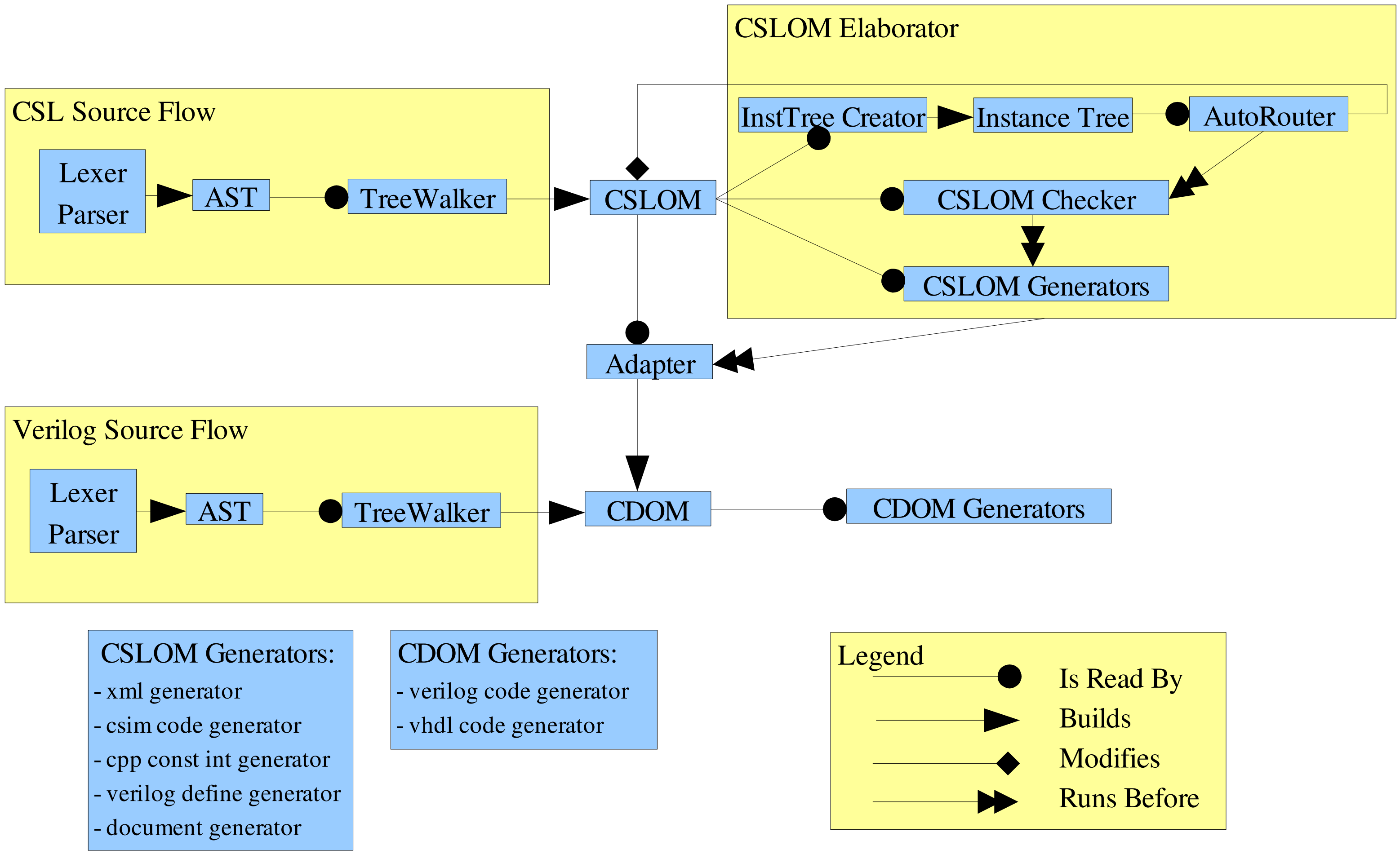
Internal Architecture of CSLC

CSLC need-to-know

To better understand what is explained in this document please take some time and learn about the following topics:

- design patterns: visitor, singleton
- data structures: trees, colored trees.
- c++ inheritance
- c++ virtual functions
- c++ classes
- c++ members and methods flags: static, const, mutable.
- c++ typedefs, and preprocessor directives
- c++ standard template library
- c++ STL containers and their complexity: vector, map, bitset, string, iterators.
- c++ streams
- boost libraries: counted references: shared_ptr, weak_ptr.
- csl objects and components.
- lexers, parsers and ASTs(abstract syntax tree)


CSL Source Flow
Lexer
Parser
AST
TreeWalker
CSLOM
CSLOM Elaborator
InstTree Creator
Instance Tree
AutoRouter
CSLOM Checker
CSLOM Generators
Adapter
Verilog Source Flow
Lexer
Parser
AST
TreeWalker
CDOM
CDOM Generators
CSLOM Generators:
- xml generator
- csim code generator
- cpp const int generator
- verilog define generator
- document generator
CDOM Generators:
- verilog code generator
- vhdl code generator
Legend
Is Read By
Builds
Modifies
Runs Before

Build System

The Build System uses 'ant'. Since 'ant' was build to compile Java code, a new feature needed to be added 'ant-contrib' (http://ant-contrib.sourceforge.net) which has defined tasks for handling C++ code compilation. The build files are in XML format.

There are several subprojects that can be independently built (listed in the order of dependencies, not the order they are run):

- csl_xml_warn_error (generates 'libWE.a')
- support (generates 'libSupport.a')
- parser (generates 'libVerilogParser.a' and 'libCslParser.a')
- preproc (generates 'libParser.a')
- cdom (generates 'liCdom.a')
- cslom (generates 'libCslom.a')
- cslom2cdom_adapter (generates 'libAdapter.a')
- cslom_generators (generates 'libCslomGen.a')
- autorouter (generates 'libAR.a')
- cslom_design_checker (generates 'libCslomChecker.a')
- cslc (generates CSLC binary / executable)
- csim (actually built separately from the project; can be seen as a project of its own)

For each of these subprojects there is a 'build.xml' file, all of those being based on the 'build.xml.template' file.

There is also a dependecies XML file that checks for platform, compiler version and ANTLR version compatibility.

The master build file, located in the root of the project, is responsible for the building order, creating output directories and checking dependencies.

Note that all the subprojects build a library each and having no executable output no linking is needed.

The 'cslc' (main) links all the libraries and creates the executable.

These are the libraries dynamically linked:

- 'stdc++'
- 'm'
- 'pthread'
- 'hpdf'
- 'png'
- 'xerces-c'
- 'boost_filesystem'

These are the libraries statically linked (the order is important):

- 'gcov'
- 'WE'
- 'Preproc'
- 'Cdom'
- 'VerilogParser'
- 'CslParser'
- 'Cslom'
- 'Adapter'
- 'CslomChecker'
- 'CslomGen'
- 'AR'
- 'Support'
- 'antlr'
- 'rlm'

Note that all the libraries listed above are named without the prefix in which they are inserted in the command line by the 'ant', that being '-l<name>'. The complete name of the libraries would be 'lib<name>.so' for the shared (dynamically linked) ones and 'lib<name>.a' for the static ones.

CSLC Main

The CSLC Main is responsible for calling all the other components and has integrated within it the RLM APIs for checking out and in the CSLC licenses. First the license is checked to be valid and if it is invalid then the CSLC exists with a message stating the reason why the license was invalid.

Classes

There are only three classes:

- 'CSLcSignal' which is a class used to be thrown and caught as an exception in case the execution stops unexpectedly
- 'CSLcMessages' is the class used to handle error messages
- 'CSLcMain' is the class that rounds up and calls all the other components

CLI

CLI stands for Command Line Interface and it is the part of the CSLC that parses, interprets and resolves the arguments passed to the compiler from the command line.

There are several types of arguments:

- arguments that have no parameters (i.e. '--help')
- arguments that require a number (i.e. '--csl_max_error')
- arguments that require a choice from a list (i.e. '--csl_pp')
- arguments that require a path to a file (i.e. '--f')
- arguments that require a path to a directory (i.e. '--dir')
- arguments that describe a preprocessor define (i.e. '--D')
- arguments that require a list of extensions (i.e. '+libext+')
- arguments that require a list of directory paths (i.e. '+incdir+')

Classes

There is a base class called 'CLiArgumentBase' for all the classes that describe the behavior of an argument type (the name of the classes should be straight forward), those being:

- 'CLiArgumentEmpty'
- 'CLiArgumentNumber'
- 'CLiArgumentOption'
- 'CLiArgumentFileName'
- 'CLiArgumentDirName'
- 'CLiArgumentCslDefine'
- 'CLiArgumentVerilogDefine'
- 'CLiArgumentVerilogDirList'
- 'CLiArgumentVerilogExtList'

There are also some 'side' classes:

- 'CLiToken' that simply stores a CLI token, being a string, and its origin, that is command line arguments can be passed to the CSLC through files and in case an error occurs in one of those files the CLI should track it to its source and correctly report the error
- 'CLiArgumentList' which is the class that is used for interpreting the 'raw' arguments passed from the command line; this is the class that gets instantiated in the CSLC's main
- 'CLiError' which was created to report the errors, since it was decided that the CLI's errors should not interfere with the errors in the WE system

Other features

The 'CLiCommon' structure was created as an utility package; here are some features implemented so far:

- conversion from string to int
- get the status of a certain file (that is if the file exists, what access rights does the current user have on the file)
- open a file
- delete a file
- get the value of an environment variable
- generate random filename
- get the relative path to a file
- check the extension of a file

This structure should be included into the 'support' group since it is quite useful.

The Preprocessor

Unlike other preprocessors the CSLC Preprocessor only parses the input files, executes the preprocessing directives and optimizes the content into a new file that is passed to the other CSLC components (Lexer, Parser).

The optimizations performed:

- inserts special line that signifies the current file ('#filename line_number')
- reduces unnecessary white characters at the end of line (the white characters inside the

lines are kept because often they are used for indentation)
- reduces the empty lines if the number of consecutive empty lines is greater than 8 and replaces it with a line that signifies the current file and line number
- transforms the commented lines or regions into white spaces then applies the rules above
- replaces the macros with their definitions at the time
- executes the directives

The preprocessor directives supported:
- 'include' inserts the requested file in the current file (maximum include depth is 100)
- 'line' that does nothing more that place a line '#filename line_number'
- 'define' that adds a macro definition
- 'undef' unlinks the macro definition
- 'ifdef' checks whether the macro is defined or not
- 'ifndef' checks that the macro was not yet defined
- 'else' must be coupled with a 'ifdef' or 'ifndef' directive and checks in if the opposite of the condition is evaluated
- 'endif' must be coupled with a 'ifdef' or 'ifndef' directive and ends the condition

Classes

There three classes:
- 'CSLcPp' the main class, that gets instantiated in the CSLC's main
- 'CSLcPpFile' a class that represents a source file being added in the compiler's flow (the ones included from other files also)
- 'CSLcPpCommon' a Singleton class that stores some common features for all files being preprocessed

CSLOM object tree and scoping tree

CSLOM is divided in two trees: the object tree and the scoping tree.

The object tree. Design and Preliminary Structure.

The object tree is defined by the CSLOmBase class, which is the base class of all CSLOM objects except one, CSLOmScope(class that links the objects in the scoping tree).

The base class defines the CSLOM object tree as an colored tree. To do that, the base class has three protected members:

- `m_parent` - weak reference to base class
- `m_children` - strong reference to vector of strong references to base (typedefed as `RefTVec_RefCSLOmBase`)
- `m_type` - enumerated type typedefed as `ECSLOmType`

The `m_parent` member is a reference to the parent node of current object. The reference is null if the current object is the root of the object tree.

The `m_children` member is a vector of references to the children of the current object. The reference to the vector is null if the object is a leaf level node.

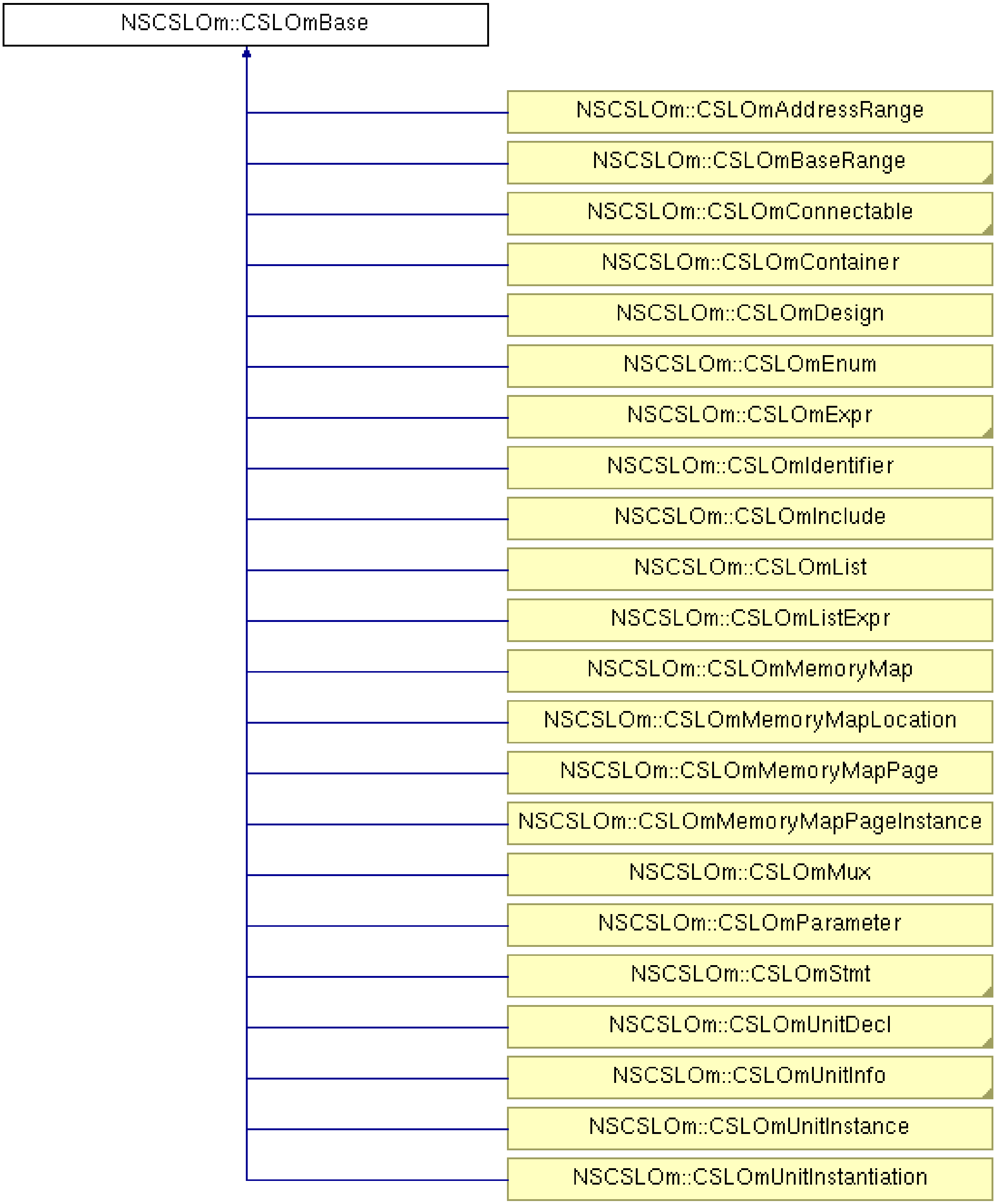
The `m_type` member represent the "color" of the current node(color - which of the derived types). Is initialized in the constructor. The constructor of the base class is protected so it can be called only from the derived classes.

The `ECSLOmType` has values only for the classes derived directly from the base class. For instance, all the expression classes have the same color: `TYPE_EXPR`. There is another member in the base class for all expressions, CSLOmExpr, that specify what kind of expression the current node is. This way we maintain the same color for similar object and differentiate them at a higher level in the hierarchy.

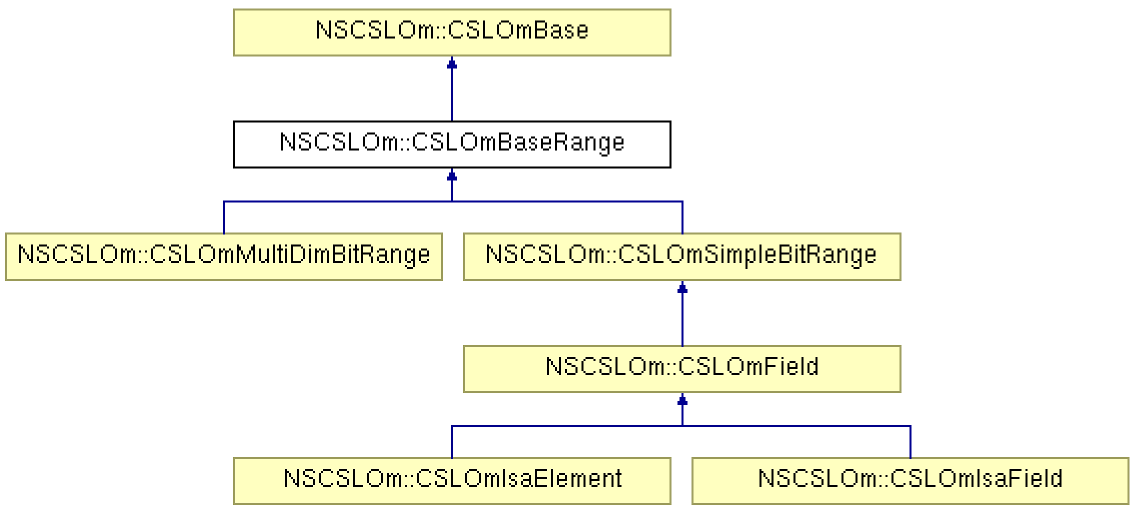
The above coloring strategy is hard to use when trying to get the identity of an expression for instance when you have a base object. That is why there where implemented predicates in the base class for every derived type from base, directly or indirectly. For instance if we have a base object and we want to test it if it is of CSLOmExprOp type it is only necessary to call the correct predicate, in this case isExprOp(). If these predicates weren't implemented then to test the type of a base object to see if it is of CSLOmExprOp type then you would have to test the `m_type` to be `TYPE_EXPR` then cast the object to expression and test the `m_exprType` member to be `EXPR_OP`.

The coloring rules of CSLOM are subject to change since new information is to be added in the object-model with any new feature of the cslc.

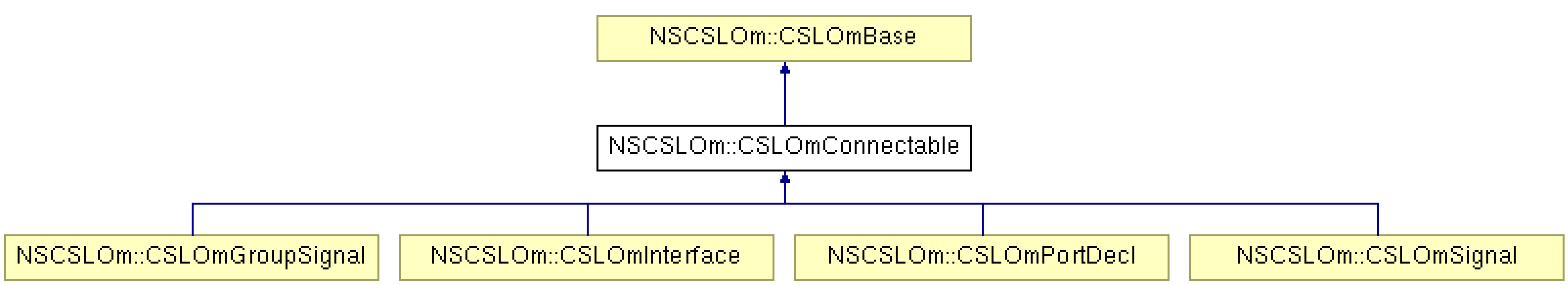
The following image presents the inheritance diagram for the base class. The classes derived from the base class also represent the base colors on the object model. The classes with the right-down corner clipped expand into other colors presented in the other diagrams.


NSCSLOm::CSLOmBase
NSCSLOm::CSLOmAddressRange
NSCSLOm::CSLOmBaseRange
NSCSLOm::CSLOmConnectable
NSCSLOm::CSLOmContainer
NSCSLOm::CSLOmDesign
NSCSLOm::CSLOmEnum
NSCSLOm::CSLOmExpr
NSCSLOm::CSLOmIdentifier
NSCSLOm::CSLOmInclude
NSCSLOm::CSLOmList
NSCSLOm::CSLOmListExpr
NSCSLOm::CSLOmMemoryMap
NSCSLOm::CSLOmMemoryMapLocation
NSCSLOm::CSLOmMemoryMapPage
NSCSLOm::CSLOmMemoryMapPageInstance
NSCSLOm::CSLOmMux
NSCSLOm::CSLOmParameter
NSCSLOm::CSLOmStmt
NSCSLOm::CSLOmUnitDecl
NSCSLOm::CSLOmUnitInfo
NSCSLOm::CSLOmUnitInstance
NSCSLOm::CSLOmUnitInstantiation

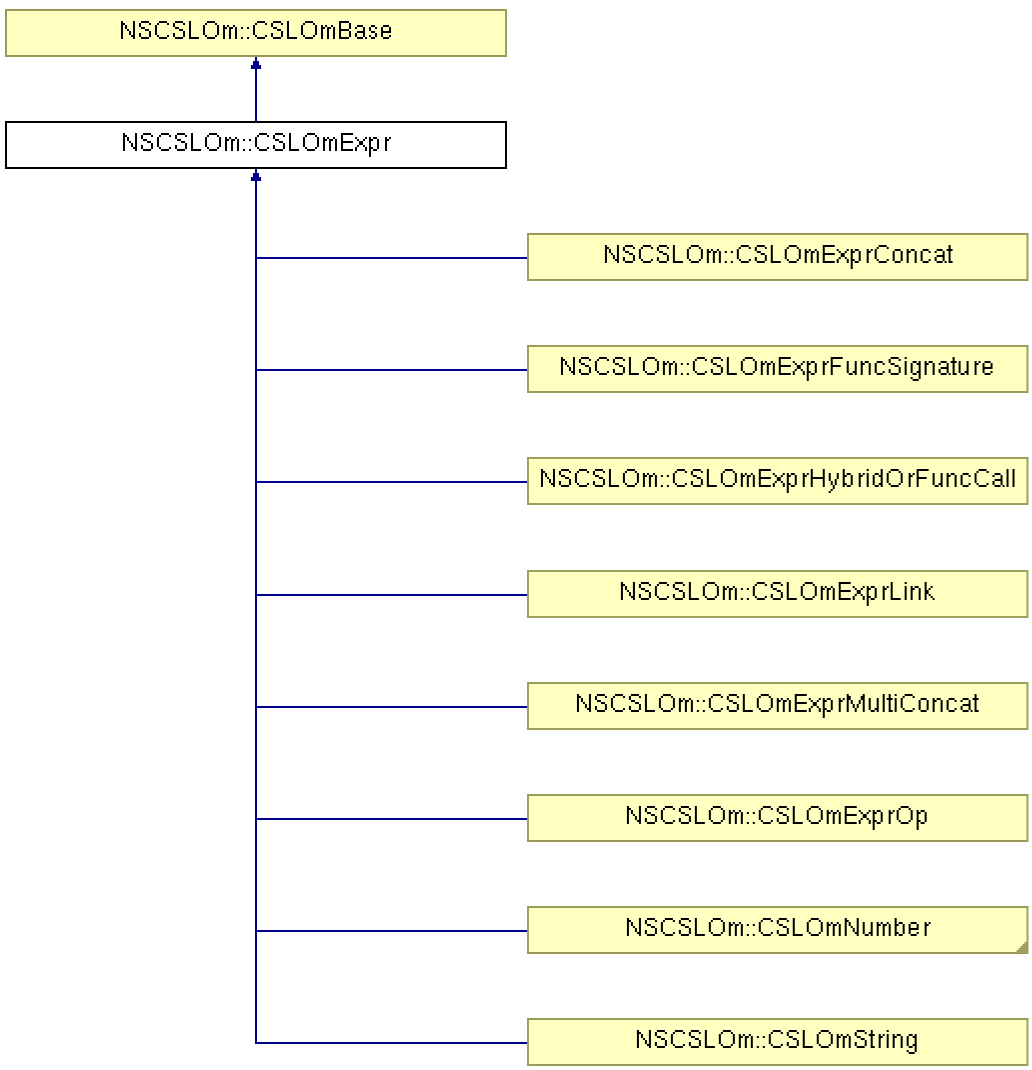
Inheritance diagram/coloring for ranges and fields:



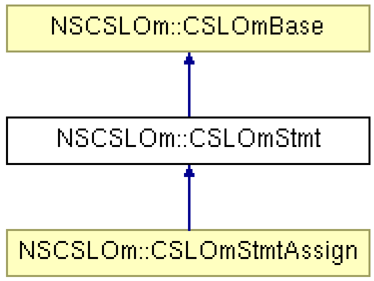
Inheritance diagram/coloring for connectible object:



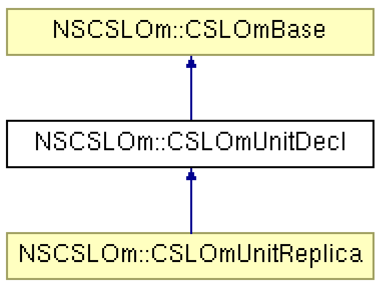
Inheritance diagram/coloring for expressions:



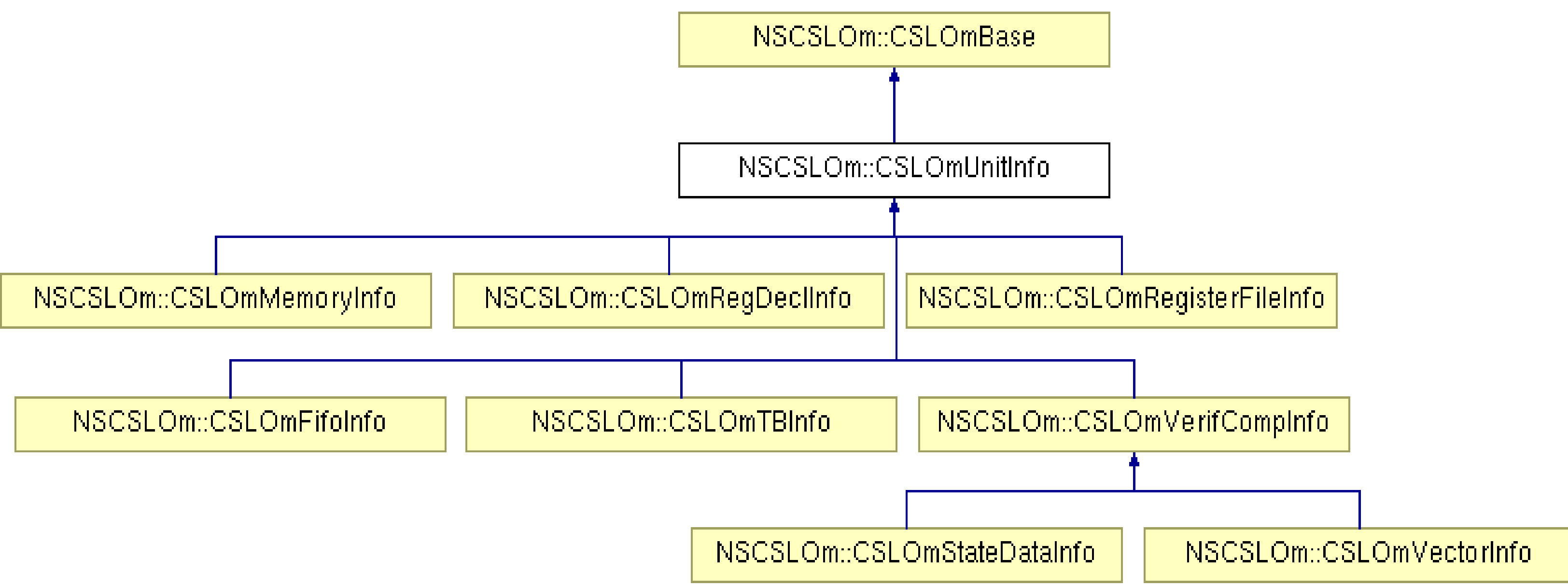
Inheritance diagram/coloring for statements:



Inheritance diagram/coloring for unit declaration:



Inheritance diagram/coloring for unit info:



The scoping tree. Design, Structure, Uses and Traversal.

The scoping tree is a colored tree also. The scoping tree is “hidden” in the object tree and it does not use a base class for all it's nodes as in the object tree.

There are three kind of nodes in the scoping tree:

- scope nodes: defined by `CSLOmScope` class
- identifier nodes: defined by `CSLOmIdentifier` class(part of the object tree; holds the name of the object; are situated as the first child of the named node in the object tree).
- scope holder nodes: defined by multiple classes in the object tree. Predicate `isScopeHolder`() in the `CSLOmBase` class can identify a scope holder in the object tree. Every scope holder object contains as a member a reference to a `CSLOmScope` object.

`CSLOmScope` class contains a STL map between a string and a reference to a `CSLOmIdentifier` object. The map represents the current named objects in the current scope. We used a map for the low search complexity, log(n).

Another member of the `CSLOmScope` class is `m_id` a reference to the identifier of the scope holder object. This reference is null only in the case of the design scope, which represents the scope of the entire project compiled, also knows as the global scope. The design scope is also the root of the scoping tree. Its scope holder is a `CSLOmDesign` object which is the root of the object tree.

For easy access to the current parent scope in a subtree of the object tree, it has been added another member to the `CSLOmBase class` called `m_cachedParentScope`, a weak reference to a `CSLOmScope` scope, which represents the scope of the first scope holder object found by going upwards in the object tree hierarchy. This member is only set to not-null only when it is required at least once. To get the parent scope of the current object, just call `CSLOmBase::getParentScope()` method on that object. If the `m_cachedParentScope` is not yet set, the method will go recursively and search for the first one set or for a scope holder.

To registration of a new name in a scope is done automatically when building the identifier. In the build function of the `CSLOmIdentifier` class, there is a call to the `CSLOmBase::getParentScope()` which - as explained above - returns the first scope it found by searching upwards in the object tree hierarchy. After the scope is found, the `CSLOmScope` method `CSLOmScope::registerId()` is called.

VERY IMPORTANT: Because of the use of `getParentScope()` in the CSLOmIdentifier build method, it is mandatory to set the parent in the object tree to the current object BEFORE building the identifier.

There are two cases when searching for an object inside of a scope:

- local search: the object you are looking for is in the current scope;
- down search: the object is represented by a hierarchical identifier, and is located in one of the scopes from the subtree of the current scope.

There are several methods in the scope class, that implement the above searches:

- `CSLOmScope::lookupLocal(string)` - searches the name in the current scope(calls find method in the name map); the parameter is the name of the object; returns the identifier with the specified name; if not found returns a null reference;

- `CSLOmScope::lookupDownward(string)` -
  1. searches the name(delimited by the '.' character) in the current scope(using lookupLocal);
  2. if the returned identifier is the id of a scope holder it changes the current scope to the scope of the found scope holder;
  3. repeats steps 1 and 2 until no more '.' are found;
  4. if there are still names in the HID('.' are found) and the returned id is not of a scope holder, it returns a null reference;
  5. if lookupLocal returns a null reference(name not found in current scope), then lookupDownwarn returns a null reference;

- `CSLOmScope::lookupDownward(vector of strings)` -
  1. searches the name(each element of the vector is one name) in the current scope(using lookupLocal);
  2. if the returned identifier is the id of a scope holder it changes the current scope to the scope of the found scope holder;
  3. repeats steps 1 and 2 until no strings are left in the vector;
  4. if there are still names in the HID(not reached the end of the vector) and the returned id is not of a scope holder, it returns a null reference;
  5. if lookupLocal returns a null reference(name not found in current scope), then lookupDownwarn returns a null reference;

VERY IMPORTANT: Because of large scale use of lookups no errors are throws from inside the function, that is why after a search, always check the result.

VERY IMPORTANT: For CSLOmExprLink do not use the lookup methods for searching the object it points to. Use doEval methods, because the expression evaluation is a lot more complex then a simple search.

Boost shared and weak pointers.

In cslc the majority of objects are built and accessed using boost pointers. We are using the boost shared and weak pointers through typedefs and build functions.

VHDL Generator

The VHDL Generator is the component that traverses the CDOM tree and creates VHDL output. The implementation is based on the visitor design pattern.

Classes

There are two classes:

- 'CVHDLVisitorTraversal' an abstract class that inherits 'CVisitor'
- 'CVHDLGenerator' the class that handles the code generation; inherits 'CVHDLVisitorTraversal'

The 'CVHDLVisitorTraversal' class implements the visit functions for each CDOM class and decides the traversal order of the CDOM tree. It also has some flags to help custom traversal algorithms, that is one can state that a component's visit function should be skipped. The flags are as follow:

- 'TRAVERSAL_FLAG_MODULE', 'TRAVERSAL_FLAG_MODULE_BEFORE', 'TRAVERSAL_FLAG_MODULE_IN' and 'TRAVERSAL_FLAG_MODULE_AFTER' for 'CDOmModuleDecl'
- 'TRAVERSAL_FLAG_UDP', 'TRAVERSAL_FLAG_UDP_BEFORE', 'TRAVERSAL_FLAG_UDP_IN' and 'TRAVERSAL_FLAG_UDP_AFTER' for 'CDOmUdpDecl'
- 'TRAVERSAL_FLAG_ID', 'TRAVERSAL_FLAG_ID_BEFORE', 'TRAVERSAL_FLAG_ID_IN' and 'TRAVERSAL_FLAG_ID_AFTER' for 'CDOmIdentifier'
- ''TRAVERSAL_FLAG_PARAM', 'TRAVERSAL_FLAG_PARAM_BEFORE', 'TRAVERSAL_FLAG_PARAM_IN' and 'TRAVERSAL_FLAG_PARAM_AFTER' for 'CDOmParamDecl'
- 'TRAVERSAL_FLAG_PORT', 'TRAVERSAL_FLAG_PORT_BEFORE', 'TRAVERSAL_FLAG_PORT_IN' and 'TRAVERSAL_FLAG_PORT_AFTER' for 'CDOmPortItem', 'CDOmPortDecl', 'CDOmUdpPortDeclOutput', 'CDOmUdpPortDeclInput' and 'CDOmTFPortDecl'

- 'TRAVERSAL_FLAG_RANGE', 'TRAVERSAL_FLAG_RANGE_BEFORE', 'TRAVERSAL_FLAG_RANGE_IN' and 'TRAVERSAL_FLAG_RANGE_AFTER' for 'CDOmRange'
- 'TRAVERSAL_FLAG_RANGE_LIST', 'TRAVERSAL_FLAG_RANGE_LIST_BEFORE', 'TRAVERSAL_FLAG_RANGE_LIST_IN' and 'TRAVERSAL_FLAG_RANGE_LIST_AFTER' for 'CDOmRangeList'
- 'TRAVERSAL_FLAG_EXPR', 'TRAVERSAL_FLAG_EXPR_BEFORE', 'TRAVERSAL_FLAG_EXPR_IN' and 'TRAVERSAL_FLAG_EXPR_AFTER' for 'CDOmExprOp', 'CDOmListExpr', 'CDOmMinTypMax', 'CDOmExprConcat', 'CDOmExprMultiConcat', 'CDOmFunctionCall' and 'CDOmMinTypMaxList'
- 'TRAVERSAL_FLAG_EXPR_LINK', 'TRAVERSAL_FLAG_EXPR_LINK_BEFORE', 'TRAVERSAL_FLAG_EXPR_LINK_IN' and 'TRAVERSAL_FLAG_EXPR_LINK_AFTER' for 'CDOmExprLink'
- 'TRAVERSAL_FLAG_EXPR_CONST', 'TRAVERSAL_FLAG_EXPR_CONST_BEFORE', 'TRAVERSAL_FLAG_EXPR_CONST_IN' and 'TRAVERSAL_FLAG_EXPR_CONST_AFTER' for 'CDOmNum32', 'CDOmVeriNum', 'CDOmReal' and 'CDOmString'
- 'TRAVERSAL_FLAG_SIGNAL', 'TRAVERSAL_FLAG_SIGNAL_BEFORE', 'TRAVERSAL_FLAG_SIGNAL_IN' and ''TRAVERSAL_FLAG_SIGNAL_AFTER' for 'CDOmNetDecl' and 'CDOmVarDecl'
- 'TRAVERSAL_FLAG_INSTANTIATION', 'TRAVERSAL_FLAG_INSTANTIATION_BEFORE', 'TRAVERSAL_FLAG_INSTANTIATION_IN' and 'TRAVERSAL_FLAG_INSTANTIATION_AFTER' for 'CDOmModuleOrUdpInstantiation'
- 'TRAVERSAL_FLAG_INSTANCE', 'TRAVERSAL_FLAG_INSTANCE_BEFORE', 'TRAVERSAL_FLAG_INSTANCE_IN' and 'TRAVERSAL_FLAG_INSTANCE_AFTER' for 'CDOmModuleOrUdpInstance'
- 'TRAVERSAL_FLAG_ASSN', 'TRAVERSAL_FLAG_ASSN_BEFORE', 'TRAVERSAL_FLAG_ASSN_IN' and 'TRAVERSAL_FLAG_ASSN_AFTER' for 'CDOmAssn'
- 'TRAVERSAL_FLAG_OBJ', 'TRAVERSAL_FLAG_OBJ_BEFORE', 'TRAVERSAL_FLAG_OBJ_IN' and 'TRAVERSAL_FLAG_OBJ_AFTER' for 'CDOmScope', 'CDOmComment', 'CDOmInclude', 'CDOmDesign', 'CDOmParamDeclCollection', 'CDOmParamOverride', 'CDOmDelay', 'CDOmInitOrAlways', 'CDOmContAssn', 'CDOmStmt', 'CDOmStmtBlock', 'CDOmStmtAssn', 'CDOmEventControl', 'CDOmEventExpr', 'CDOmDelayControl', 'CDOmStmtProcContAssn', 'CDOmStmtProcTimingControl', 'CDOmStmtCase', 'CDOmStmtIf', 'CDOmStmtLoop', 'CDOmStmtTaskEnable', 'CDOmStmtWait', 'CDOmStmtDisable', 'CDOmStmtEventTrigger', 'CDOmRangeExpr', 'CDOmUdpCombEntry', 'CDOmUdpSeqEntry', 'CDOmUdpInitStmt', 'CDOmTaskDecl', 'CDOmFuncDecl', 'CDOmGenvarDecl', 'CDOmGenInst',

'CDOmGenItemNull', 'CDOmGenItemIf', 'CDOmGenItemCase', 'CDOmGenItemLoop', 'CDOmGenItemBlock', 'CDOmEventDecl', 'CDOmSpecifyBlock', 'CDOmPulseStyleOrShowCancelledDecl', 'CDOmPathDecl', 'CDOmPathDelayValue', 'CDOmSpecifyTerminalList', 'CDOmDelayedDataOrReference', 'CDOmTimingCheckEventControl', 'CDOmTimingCheckEvent', 'CDOmSystemTimingCheck', 'CDOmGateInstantiation', 'CDOmPulseControl', 'CDOmAttrList', 'CDOmAttrListCollection' and 'CDOmDefine'

Verilog Generator

The Verilog Generator is based on the visitor design pattern.

Classes

There is only one class, 'CVerilogGenerator' that inherits 'CVisitorTraversal'. Since the CDOM is built as a Verilog syntax tree the code generation is pretty straightforward.