Configuring Flow Analysis

Configuring Depth of Flow Analysis

Flow Analysis builds paths through the analyzed code to detect different kinds of problems. Since the analysis of all possible paths that span through the whole application may be infeasible, you can set up the desired level of depth of analysis. A deeper analysis will result in more findings, but the performance will be slower and the memory consumption will increase slightly.

You can specify the depth of analysis by using the test configuration interface in DTP. Go to **Report Center> Test Configurations> Static Analysis> Flow Analysis Advanced Settings> Performance> Depth of analysis** and choose one of the following options by selecting a radio button:

- **Shallowest (fastest)**: Finds only the most obvious problems in the source code. It is limited to cases where the cause of the problem is located close to the code where the problem occurs. The execution paths of violations found by this type of analysis normally span several lines of code in a single function. Only rarely will they span more than 3 function calls.
- **Shallow (fast)**: Like the "Shallowest" analysis type, finds only the most obvious problems in the source code. However, it produces a greater overall number of findings and allows for examination of somewhat longer execution paths.
- **Standard**: Finds many complicated problems with execution paths containing tens of elements. The standard analysis goes beyond shallow analysis and also looks for more complicated problems, which can occur because of bad flow in a single function or due to improper interaction between different functions in different parts of the analyzed project. Violations found by this type of analysis often reveal non-trivial bugs in the analyzed source code and often span tens of lines of code.
- **Deep (slow)**: Allows for detection of a greater number of problems of the same complexity and nature as those defined for "Standard" depth. This type of analysis is slower than the standard one.
- **Thorough (slowest)**: Finds more complicated problems. This type of analysis will perform a thorough scan of the code base; this requires more time, but will uncover many very complicated problems whose violation paths can span more than a hundred lines of code in different parts of the scanned application. This option is recommended for nightly runs.

The depth of Flow Analysis is set to **Standard** by default.

Setting Timeout Strategy

Apart from the depth of analysis, Flow Analysis uses an additional timeout guard to ensure the analysis completes within a reasonable time. An appropriate strategy can be set by using the test configuration interface in DTP. Go to **Report Center> Test Configurations> Static Analysis> Flow Analysis Advanced Settings> Performance> Strategy for Timeouts** and choose one of the follow-ing options by selecting a radio button:

- **time**: Analysis of the given hotspot is stopped after spending the defined amount of time on it. Note: in some cases, using this option can result in a slightly unstable number of violations being reported.
- **instructions**: Analysis of the given hotspot is stopped after executing the defined number of Flow Analysis instructions.Note: to determine the proper number of instructions to be set up for your environment, review information about timeouts in the Setup Problems section of the generated report.
- **off**: No timeout. Note: using this option may require significantly more time to finish the analysis.

The default timeout option is **time** set to 60 seconds. To get information about the Flow Analysis timeouts that occurred during the analysis, review the Setup Problems section of the report generated after the analysis.

Running Flow Analysis in Incremental Mode

By default, Flow Analysis performs a complete analysis of the scope it is run on. This can take considerable time when running on large code bases.

The most common way of performing Flow Analysis analysis is to run nightly tests on a single code base that changes slightly from day to day. Flow Analysis's incremental analysis mode is designed to reduce the time required to run analysis in this typical scenario. With incremental analysis mode, Analysis memorizes important analysis data during the initial run, then reuses it during the subsequent runs — rerunning analysis only for parts of the code that have been modified or are tightly connected to the modified code.

The initial run of Flow Analysis may be slightly slower than running without incremental analysis. This is because Flow Analysis in addition to performing a complete analysis of the code base, Flow Analysis saves data to be reused in subsequent runs.

Disk space is required to store the necessary data.

Incremental analysis options control the incremental analysis feature. Available options are:

- **Enable incremental analysis**: Determines whether the incremental analysis is used.
- **Compact incremental caches every [days]**: Determines how often compactization of incremental caches is run. Incremental analysis is optimized for speed; although Flow Analysis strives to always keep cache sizes small and remove unnecessary data, source code changes may result in these caches containing some data that will no longer be used. Compactization, which is run regularly as defined by this parameter, removes all outdated data. More precisely, if the time that has elapsed since the previous compactization is greater than the number of days specified for this option, compactization is performed immediately after the incremental run of Flow Analysis.

Running Flow Analysis with Swapping of Analysis Data Enabled

In this mode, analysis data is written to disk. Swapping of analysis data uses the same persistent storage and is done in a similar process as incremental analysis. If the analysis is run on a large project, the analysis data that represents a semantical model of the analyzed source code may consume all the memory available for running Flow Analysis. If this occurs, Flow Analysis will remove from memory parts of the analysis data that are not currently necessary and reread it from disk later.

In general, we recommend runningC/C++testin a large JVM heap configured with the Xmx JVM option. This is to minimize swapping, which results in greater performance. If sufficient memory is available, swapping of analysis data may be disabled, which may speed up code analysis.

You can enable or disable the mode by using the test configuration interface in DTP:

Enable swapping of analysis data to disk:Enabledby default. If this option is disabled, it may result in faster analysis, if you are running Flow Analysis analysis on small to moderate size projects that do not require a lot of memory or when plenty of memory is available (for example, for 64-bit systems).

Configuring Verbosity of Flow Analysis

You can configure the following options by using the test configuration interface in DTP:

- **Do not report violations when cause cannot be shown**: Determines whether Flow Analysis reports violations where causes cannot be shown. Some Flow Analysis rules require that Flow Analysis checks all the possible paths leading to a certain point and verifies that a certain condition is met for all those paths. In such cases, a violation is associated with a set of paths (whereas in simple cases, a violation is represented by only one path). All of the paths in such a violation end with the violation point common to all the paths in the violation. However, different paths may start at different points in code. The beginning of each path is a violation cause (a point in code which stipulates a violation of a certain condition later in the code at the violation point). If a multipath violation's different paths have different causes, Flow Analysis will show only the violation point (and not the violation causes).
  Violations containing only the violation point may be difficult to understand (compared to regular cases where Flow Analysis shows complete paths starting from violation causes and leading to violation points). That’s why we provide an option to hide violations where the cause cannot be shown.
- **Do not show more than one violation per point:** Restricts reporting to one violation (for a single rule) per violation point. For example, one violation will be reported when Flow Analysis detects a potential null dereference with multiple sources of the null value. When verbosity is set to this level, Flow Analysis performance is somewhat faster.

Specifying Terminating Functions

You can define functions that terminate application execution. C/C++ developers sometimes use functions that terminate application execution in the event of a fatal error from which recovery is impossible. Examples of such functions are `abort()` and `exit()` from the standard library. Since Flow Analysis analyzes the application’s execution flow, it's important for it to be aware of the terminating functions that break execution flow by immediately stopping the application. Without such knowledge, Flow Analysis might make incorrect assumptions about the application flow.

Flow Analysis is aware of the terminating functions that are defined in the standard library. However, this is often not sufficient because non-standard libraries define their own terminating functions. If your application uses one of these functions, you should communicate that to Flow Analysis by specifying the custom terminating function in the Terminators tab. Otherwise, Flow Analysis may produce false positives with execution paths passing by terminating functions.

Use the table listing supported APIs to enable/disable terminators from various APIs as well as to define your own APIs containing terminating functions. To add information about terminating functions from a certain library:

1. Click the **+** button in the top row of the table.
2. Click the arrow to expand the **Functions that terminate application execution** tab.
3. Complete the table that opens; the table has the following columns:

   • **Enabled**: Specifies whether a built-in or custom terminator should be considered during the analysis.
   • **Fully qualified type name or namespace (wildcard)**: Specifies the entity for a particular terminator. If this field is left empty, only the global function with the name specified in the 'Function name' column will be considered a terminator. For example: The field value may be "myNameSpace::myClass" if the terminator is declared in 'myClass' coming from the 'myNameSpace' namespace. Or, it may be "myNameSpace" if it is not declared in a type, but belongs only to the 'myNameSpace'.
   • **Function name (wildcard)**: Specifies the name of the terminating function.
   • **+ definitions in subclasses**: Indicates whether the terminating function definitions in subclasses should be considered terminating functions as well. Thisappliestobothinstanceandnon-instancefunctions, and makes sense only if its fully qualified type name is specified.

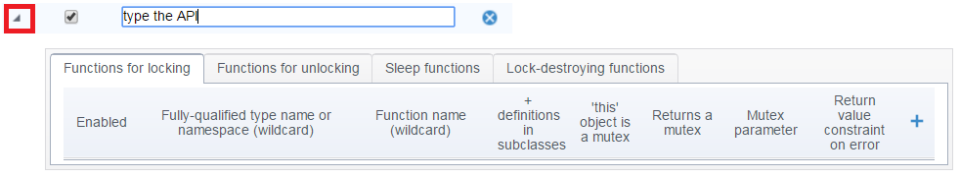
Specifying Multithreading Options

The **Multithreading** tab allows you to define functions for synchronization between threads as well as to activate/deactivate known multithreading functions. The information defined here affects the behavior of rules from the BD.TRS (Threads and Synchronization) category. These rules will check all the functions that are defined and activated on this tab.

Use the table that lists supported APIs to enable/disable synchronization functions from various APIs as well as to define your own APIs containing synchronization functions. To add information about synchronization functions from a certain library:

1. Click the + button in the top row of the table.
2. Type the name of the library in the API field.
3. Click the arrow to expand the tabs and complete the tables to define the following types of functions (details about completing the tables are provided below):

   • Functions for locking (for instance, obtaining a mutex)
   • Functions for unlocking (for instance, releasing a mutex)
   • Sleep functions
   • Destroy lock functions



Functions for locking

Complete the table with the following information:

- **Enabled**: specifies whether the locking function should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the locking function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses**:a check box that indicates whether the definitions (of the function with the given name) in subclasses should be considered locking functions as well. Note that this applies to both instance and static functions.
- **"this" object is a mutex**:a check box that indicates that the function locks a mutex in the object on which the function is called.
- **Returns a mutex**:a check box that indicates that the function returns a mutex.
- **Return value constraint on error**: specifies a return value constraint in case of allocation failure if a resource allocator returns an integral value. Enter the condition in the following format: <comparison operator><integer value>. For example, if the function returns a non-zero value on failure, enter "!=0" (without quotes) into the field. If return code on error is -1, type "==-1" there. In addition to "!=" and "==", you can use the following operators for specifying error conditions: ">", ">=", "<", and "<=".
- **Mutex parameter**: specifies that the function locks a mutex in one of its parameters.

Functions for unlocking

Complete the table with the following information:

- **Enabled**: specifies whether the unlocking function should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the unlocking function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses**: a checkbox that indicates whether the definitions (of the function with the given name) in subclasses should be considered unlocking functions as well. Note that this applies to both instance and static functions.
- **"this" object is a mutex**: a check box that indicates that a mutex in the object on which the function is called is unlocked
- **Mutex parameter**: specifies that a mutex in one of the parameters is unlocked.

Sleep functions

Complete the table with the following information:

- **Enabled**: specifies whether the sleep function should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the sleep function. '*' can be used to denote any num-ber of any symbols.
- **+ definitions in subclasses**: a check box that indicates whether the definitions (of the function with the given name) in subclasses should be considered sleep functions as well. Note that this applies to both instance and static functions.

Destroy lock functions

Complete the table with the following information:

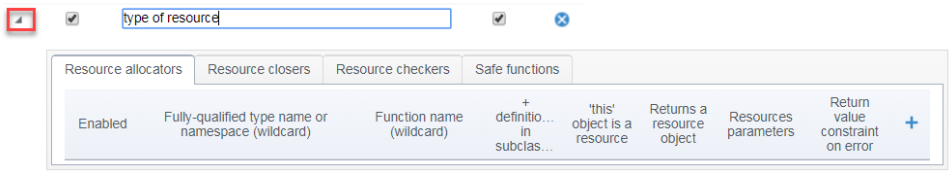
- **Enabled**: specifies whether the lock-destroying function should be considered during analysis.

- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the lock-destroying function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses**: a check box that indicates whether the definitions (of the function with the given name) in subclasses should be considered lock-destroying functions as well. Note that this applies to both instance and static functions.
- **"this" object is a mutex**: a check box that indicates that a mutex in the object on which the function is called is destroyed.
- **Mutex parameter**: specifies that a mutex in one of the parameters is destroyed.

Specifying Resources

The **Resources** tab allows you to define which resources the BD.RES category (Resources) rules should check. These rules check for the correct usage of all resources that are defined and enabled on this tab.

1. Specify the type of resource.
2. Select the **Enabled** checkbox.
3. If appropriate/desired, disable the **Do not report leaks at termination** option.
4. Click the arrow to expand the **Resource Allocators**, **Resource Closers** , **Resource checkers**, and **Safe functions** tabs and complete the tables that open with the information about allocators, closers, checkers, and safe functions. Details about completing these tabs are provided below.



Configuring Resource Allocators

The **Resource** allocators table can be completed with the descriptors of functions that can produce a resource. The table has the following columns:

- **Enabled**: specifies whether the allocator should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the allocating function. '*' can be used to denote any number of any symbols.
- **Resource parameters**: specifies that the function allocates a resource in one or more of its parameters. In this case, either specify a 1-based number of the parameter that is allocated by the function, or use '*' to denote that all of the parameters are allocated.
- **+ definitions in subclasses**: a check box that indicates whether the definitions (of functions with the given name) in subclasses should be considered allocators as well. Note that this applies to both instance and static functions.
- **"this" object is a resource**: a check box that indicates that the function allocates a resource in the object on which the function is called.
- **Returns a resource object**: a check box that indicates that the function returns an allocated resource.
- **Return value constraint on error**: specifies a return value constraint in case of allocation fail-ure if a resource allocator returns an integral value. Enter the condition in the following format: <comparison operator><integer value>. For example, if the function returns non-zero value on failure, enter "!=0" (without quotes) into the field. If return code on error is -1, type "==-1" there. In addition to "!=" and "==", you can use the following operators for specifying error conditions: ">", ">=", "<", and "<=".

It is common that allocation functions return an error code to indicate allocation failure. When an allocation function returns a pointer to a resource, a NULL pointer normally indicates an allocation failure. When Flow Analysis is looking for resource leaks, it needs to understand if allocation succeeded or failed; this helps it report only missing calls to deallocation functions on paths where allocation actually occurred. In cases where a resource allocator function returns a pointer to a resource, Flow Analysis assumes that the resource is successfully allocated if the pointer is not NULL.

Configuring Resource Closers

The **Resource** closers table can be completed with the descriptors of functions that can close a resource. The table has the following columns:

- **Enabled**: specifies whether the closer should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the closing function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses**: a check box that indicates whether the definitions (of functions with the given name) in subclasses should be considered closers as well. Note that this applies to both instance and static functions.
- **"this" object is a resource**: a check box that indicates that a resource in the object on which the function is called is closed.
- **Resource parameters**: specifies that a resource in one or more of its parameters is closed. In this case, either specify a 1-based number of the parameter that is closed by the function, or use '*' to denote that all of the parameters are allocated.

Configuring Resource Checkers

The **Resource checkers** table can be completed with the descriptors of functions that can check if the resource is open. The table has the following columns:

- **Enabled**: specifies whether the checker should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the checking function. '*' can be used to denote any number of any symbols.

- **+ definitions in subclasses**: a check box that indicates whether the definitions (of functions with the given name) in subclasses should be considered checkers as well. Note that this applies to both instance and static functions.
- **'this' object is a resource**: a check box that indicates that the function checks a resource in the object on which the function is called.
- **Resources parameters**: specifies that the function checks a resource in one or more of its parameters. In this case, either specify a 1-based number of the parameter that is checked by the function, or use '*' to denote that all of the parameters are checked.
- **Returned value when resource is open**: specifies the return value of checking the function when the resource is open. Acceptable values are `true` or `false`.

Configuring Safe Functions

The **Safe functions** table can be completed with the descriptors of functions that can be safely called on a closed resource – without triggering the BD-RES-FREE rule. The table has the following columns:

- **Enabled**: specifies whether the safe function should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the safe function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses**: a check box that indicates whether the definitions (of functions with the given name) in subclasses should be considered safe as well. Note that this applies to both instance and static functions.

Extending Scope of Analysis

When performing code analysis, Flow Analysis processes definitions of functions that are defined in source and header files under test. Functions that are defined in header files outside the testing scope are not analyzed, and Flow Analysis is not aware of their semantics. If Flow Analysis requires information about function definitions that are defined in header files outside the testing scope, you can configure the following options:

External files to analyze: Specifies absolute paths to additional header files to be analyzed by Flow Analysis. Use wildcards to specify the pattern.

External functions to analyze: Specifies additional functions to be analyzed by Flow Analysis. Complete the table with the following information:

- **Enabled**: specifies whether the function should be considered during analysis
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the function. '*' can be used to denote any number of any symbols.
- **Number of parameters**: specifies the number of function's parameters. '-1' can be used to denote any number of parameters.
- **+ definitions in subclasses**: a checkbox that indicates whether the definitions (of functions with the given name) in subclasses should be included as well. Note that this applies to both instance and static functions.

Compiler-specific Settings

Internal representation of the "errno**" value**: The Standarddefineserno to be a modifiable lvalue of type int. It is unspecified whether errno is a macro or an identifier declared with an external linkage. Implementations may use the global variable "errno" or "__errno", or apply the "(*errno_function())" pattern with different names of the called functions. This option allows you to specify the names of these variables and functions with regular expressions:

- **Function name pattern**: The name of the function that is called when the "errno" value is used. The name must be specified with regular expressions.
- **Variable name pattern**: The name of the variable that is called when the "errno" value is used. The name must be specified with regular expressions.

Internal representation of the call to a function from the header <ctype.h>: The Standard specifies several functions to be defined in the header <ctype.h>. Some implementations (e.g GNU GCC in the C mode) define these functions as macros that expand to the code which tests an element of the internal array against some flags. This can be either a global array or a pointer returned by a function. This option allows you to specify names of these variables and functions with regular expressions:

- **Function name pattern**: The name of the function that is invoked internally instead of one of the functions from the header <ctype.h> (define with regular expressions). The name must be specified with regular expressions.
- **Variable name pattern**: The name of the variable that is used internally after a call to one of the functions from the header <ctype.h>. The name must be specified with regular expressions.

Reusing Flow Analysis Data for Desktop Analysis

One way to improve desktop performance with Flow Analysis is to reuse the server analysis data on the desktop. To do this, you need to define a mapping that allows Flow Analysis to match server file paths with corresponding desktop file paths.

Additionally, you can reuse data to run the analysis on a small scope (for example, one file) and build paths that include methods defined in files outside the defined scope of analysis, provided that these files have been analyzed.

Please, contact Parasoft Support for more information on how to use this functionality.