

DCFG

Generated by Doxygen 1.8.2

Tue Jun 2 2015 16:28:25

Contents

1	Main Page	1
1.1	Introduction	1
2	Hierarchical Index	3
2.1	Class Hierarchy	3
3	Class Index	5
3.1	Class List	5
4	Class Documentation	7
4.1	dcfg_api::DCFG_BASIC_BLOCK Class Reference	7
4.1.1	Detailed Description	8
4.1.2	Member Function Documentation	8
4.1.2.1	get_basic_block_id	8
4.1.2.2	get_exec_count	8
4.1.2.3	get_exec_count_for_thread	8
4.1.2.4	get_first_instr_addr	9
4.1.2.5	get_image_id	9
4.1.2.6	get_inner_loop_id	9
4.1.2.7	get_last_instr_addr	9
4.1.2.8	get_num_instrs	9
4.1.2.9	get_process_id	10
4.1.2.10	get_routine_id	10
4.1.2.11	get_size	10
4.1.2.12	get_source_filename	10
4.1.2.13	get_source_line_number	10
4.1.2.14	get_symbol_name	10
4.1.2.15	get_symbol_offset	11
4.2	dcfg_api::DCFG_DATA Class Reference	11

4.2.1	Detailed Description	11
4.2.2	Member Function Documentation	12
4.2.2.1	get_process_ids	12
4.2.2.2	get_process_info	12
4.2.2.3	new_dcfg	12
4.2.2.4	read	12
4.2.2.5	read	13
4.2.2.6	write	13
4.2.2.7	write	13
4.3	dcfg_api::DCFG_EDGE Class Reference	14
4.3.1	Detailed Description	15
4.3.2	Member Function Documentation	15
4.3.2.1	get_edge_id	15
4.3.2.2	get_edge_type	16
4.3.2.3	get_exec_count	16
4.3.2.4	get_exec_count_for_thread	16
4.3.2.5	get_source_node_id	16
4.3.2.6	get_target_node_id	16
4.3.2.7	is_any_branch_type	17
4.3.2.8	is_any_bypass_type	17
4.3.2.9	is_any_call_type	17
4.3.2.10	is_any_inter_routine_type	17
4.3.2.11	is_any_return_type	17
4.3.2.12	is_branch_edge_type	17
4.3.2.13	is_call_bypass_edge_type	18
4.3.2.14	is_call_edge_type	18
4.3.2.15	is_conditional_branch_edge_type	18
4.3.2.16	is_context_bypass_edge_type	18
4.3.2.17	is_context_edge_type	18
4.3.2.18	is_context_return_edge_type	18
4.3.2.19	is_direct_branch_edge_type	19
4.3.2.20	is_direct_call_edge_type	19
4.3.2.21	is_direct_conditional_branch_edge_type	19
4.3.2.22	is_direct_unconditional_branch_edge_type	19
4.3.2.23	is_entry_edge_type	19
4.3.2.24	is_excluded_bypass_edge_type	19
4.3.2.25	is_exit_edge_type	20

4.3.2.26	is_fall_thru_edge_type	20
4.3.2.27	is_indirect_branch_edge_type	20
4.3.2.28	is_indirect_call_edge_type	20
4.3.2.29	is_indirect_conditional_branch_edge_type	20
4.3.2.30	is_indirect_unconditional_branch_edge_type	20
4.3.2.31	is_rep_edge_type	21
4.3.2.32	is_return_edge_type	21
4.3.2.33	is_sys_call_bypass_edge_type	21
4.3.2.34	is_sys_call_edge_type	21
4.3.2.35	is_sys_return_edge_type	21
4.3.2.36	is_unconditional_branch_edge_type	21
4.3.2.37	is_unknown_edge_type	22
4.4	dcfg_api::DCFG_GRAPH_BASE Class Reference	22
4.4.1	Detailed Description	22
4.4.2	Member Function Documentation	23
4.4.2.1	get_basic_block_ids	23
4.4.2.2	get_inbound_edge_ids	23
4.4.2.3	get_instr_count	23
4.4.2.4	get_instr_count_for_thread	23
4.4.2.5	get_internal_edge_ids	24
4.4.2.6	get_outbound_edge_ids	24
4.5	dcfg_api::DCFG_ID_CONTAINER Class Reference	25
4.5.1	Detailed Description	25
4.5.2	Member Function Documentation	25
4.5.2.1	add_id	25
4.6	dcfg_api::DCFG_ID_SET Class Reference	25
4.6.1	Detailed Description	26
4.6.2	Member Function Documentation	26
4.6.2.1	add_id	26
4.7	dcfg_api::DCFG_ID_VECTOR Class Reference	26
4.7.1	Detailed Description	27
4.7.2	Member Function Documentation	27
4.7.2.1	add_id	27
4.8	dcfg_api::DCFG_IMAGE Class Reference	27
4.8.1	Detailed Description	28
4.8.2	Member Function Documentation	28
4.8.2.1	get_base_address	28

4.8.2.2	get_basic_block_ids_by_addr	28
4.8.2.3	get_filename	28
4.8.2.4	get_image_id	29
4.8.2.5	get_process_id	29
4.8.2.6	get_size	29
4.9	dcfg_api::DCFG_IMAGE_CONTAINER Class Reference	29
4.9.1	Detailed Description	30
4.9.2	Member Function Documentation	30
4.9.2.1	get_image_ids	30
4.9.2.2	get_image_info	30
4.10	dcfg_api::DCFG_LOOP Class Reference	31
4.10.1	Detailed Description	31
4.10.2	Member Function Documentation	31
4.10.2.1	get_back_edge_ids	31
4.10.2.2	get_entry_edge_ids	32
4.10.2.3	get_exit_edge_ids	32
4.10.2.4	get_image_id	33
4.10.2.5	get_iteration_count	33
4.10.2.6	get_iteration_count_for_thread	33
4.10.2.7	get_loop_id	33
4.10.2.8	get_parent_loop_id	33
4.10.2.9	get_process_id	34
4.10.2.10	get_routine_id	34
4.11	dcfg_api::DCFG_LOOP_CONTAINER Class Reference	34
4.11.1	Detailed Description	35
4.11.2	Member Function Documentation	35
4.11.2.1	get_loop_ids	35
4.11.2.2	get_loop_info	35
4.12	dcfg_pin_api::DCFG_PIN_MANAGER Class Reference	35
4.12.1	Detailed Description	36
4.12.2	Member Function Documentation	36
4.12.2.1	activate	36
4.12.2.2	activate	36
4.12.2.3	dcfg_enable_knob	36
4.12.2.4	get_dcfg_data	37
4.12.2.5	new_manager	37
4.12.2.6	set_cfg_collection	37

4.13	dcfg_api::DCFG_PROCESS Class Reference	37
4.13.1	Detailed Description	39
4.13.2	Member Function Documentation	39
4.13.2.1	get_basic_block_ids_by_addr	39
4.13.2.2	get_basic_block_info	39
4.13.2.3	get_edge_id	39
4.13.2.4	get_edge_info	40
4.13.2.5	get_end_node_id	40
4.13.2.6	get_highest_thread_id	40
4.13.2.7	get_predecessor_node_ids	40
4.13.2.8	get_process_id	41
4.13.2.9	get_start_node_id	41
4.13.2.10	get_successor_node_ids	41
4.13.2.11	get_unknown_node_id	41
4.13.2.12	is_end_node	42
4.13.2.13	is_special_node	42
4.13.2.14	is_start_node	42
4.13.2.15	is_unknown_node	42
4.14	dcfg_api::DCFG_ROUTINE Class Reference	43
4.14.1	Detailed Description	44
4.14.2	Member Function Documentation	44
4.14.2.1	get_entry_count	44
4.14.2.2	get_entry_count_for_thread	44
4.14.2.3	get_entry_edge_ids	44
4.14.2.4	get_exit_edge_ids	45
4.14.2.5	get_idom_node_id	45
4.14.2.6	get_image_id	45
4.14.2.7	get_process_id	46
4.14.2.8	get_routine_id	46
4.14.2.9	get_symbol_name	46
4.15	dcfg_api::DCFG_ROUTINE_CONTAINER Class Reference	46
4.15.1	Detailed Description	47
4.15.2	Member Function Documentation	47
4.15.2.1	get_routine_ids	47
4.15.2.2	get_routine_info	47
4.16	dcfg_trace_api::DCFG_TRACE_READER Class Reference	48
4.16.1	Detailed Description	48

4.16.2 Member Function Documentation	48
4.16.2.1 get_edge_ids	48
4.16.2.2 new_reader	49
4.16.2.3 open	49
Index	49

Chapter 1

Main Page

1.1 Introduction

A control-flow graph (CFG) is a fundamental structure used in computer science and engineering for describing and analyzing the structure of an algorithm or program. A dynamic control-flow graph (DCFG) is a specialized CFG that adds data from a specific execution of a program. This application-programmer interface (API) provides access to the DCFG data from within a Pin tool or a standalone program.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

dcfg_api::DCFG_DATA	11
dcfg_api::DCFG_EDGE	14
dcfg_api::DCFG_GRAPH_BASE	22
dcfg_api::DCFG_BASIC_BLOCK	7
dcfg_api::DCFG_LOOP	31
dcfg_api::DCFG_LOOP_CONTAINER	34
dcfg_api::DCFG_ROUTINE	43
dcfg_api::DCFG_ROUTINE_CONTAINER	46
dcfg_api::DCFG_IMAGE	27
dcfg_api::DCFG_IMAGE_CONTAINER	29
dcfg_api::DCFG_PROCESS	37
dcfg_api::DCFG_ID_CONTAINER	25
dcfg_api::DCFG_ID_SET	25
dcfg_api::DCFG_ID_VECTOR	26
dcfg_pin_api::DCFG_PIN_MANAGER	35
dcfg_trace_api::DCFG_TRACE_READER	48
std::set< K >	
dcfg_api::DCFG_ID_SET	25
std::vector< T >	
dcfg_api::DCFG_ID_VECTOR	26

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

dcfg_api::DCFG_BASIC_BLOCK	Interface to information about a basic block	7
dcfg_api::DCFG_DATA	Interface to all data in a DCFG	11
dcfg_api::DCFG_EDGE	Interface to information about an edge between basic blocks and/or special nodes	14
dcfg_api::DCFG_GRAPH_BASE	Common interface to any structure containing nodes and edges between them, i.e., processes, images, routines, loops and basic blocks	22
dcfg_api::DCFG_ID_CONTAINER	Interface for any container of ID numbers	25
dcfg_api::DCFG_ID_SET	Set of ID numbers	25
dcfg_api::DCFG_ID_VECTOR	Vector of ID numbers	26
dcfg_api::DCFG_IMAGE	Interface to information about a binary image within a process	27
dcfg_api::DCFG_IMAGE_CONTAINER	Common interface to any structure containing images, i.e., processes	29
dcfg_api::DCFG_LOOP	Interface to information about a loop	31
dcfg_api::DCFG_LOOP_CONTAINER	Common interface to any structure containing loops, i.e., routines, images, and processes	34
dcfg_pin_api::DCFG_PIN_MANAGER	Connection between a Pin tool and a DCFG_DATA object	35
dcfg_api::DCFG_PROCESS	Interface to information about an O/S process	37
dcfg_api::DCFG_ROUTINE	Interface to information about a routine in an image	43
dcfg_api::DCFG_ROUTINE_CONTAINER	Common interface to any structure containing routines, i.e., images and processes	46
dcfg_trace_api::DCFG_TRACE_READER	Interface to all data in a DCFG edge trace	48

Chapter 4

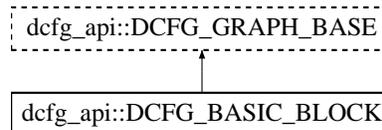
Class Documentation

4.1 dcfg_api::DCFG_BASIC_BLOCK Class Reference

Interface to information about a basic block.

```
#include <dcfg_api.H>
```

Inheritance diagram for dcfg_api::DCFG_BASIC_BLOCK:



Public Member Functions

- virtual DCFG_ID [get_basic_block_id](#) () const =0
Get basic-block ID number.
- virtual DCFG_ID [get_process_id](#) () const =0
Get the process ID.
- virtual DCFG_ID [get_image_id](#) () const =0
Get the image ID.
- virtual DCFG_ID [get_routine_id](#) () const =0
Get routine ID.
- virtual DCFG_ID [get_inner_loop_id](#) () const =0
Get innermost loop ID.
- virtual UINT64 [get_first_instr_addr](#) () const =0
Get starting or base address.
- virtual UINT64 [get_last_instr_addr](#) () const =0
Get the address of the last instruction.
- virtual UINT32 [get_size](#) () const =0
Get size.
- virtual UINT32 [get_num_instrs](#) () const =0
Get static number of instructions in the block.

- virtual const std::string * [get_symbol_name](#) () const =0
Get symbol name of this block.
- virtual UINT32 [get_symbol_offset](#) () const =0
Get symbol offset of this block.
- virtual const std::string * [get_source_filename](#) () const =0
Get name of source file for this block.
- virtual UINT32 [get_source_line_number](#) () const =0
Get line number in source file this block.
- virtual UINT64 [get_exec_count](#) () const =0
Get dynamic execution count.
- virtual UINT64 [get_exec_count_for_thread](#) (UINT32 thread_id) const =0
Get dynamic execution count.

4.1.1 Detailed Description

Interface to information about a basic block.

4.1.2 Member Function Documentation

4.1.2.1 virtual DCFG_ID dcfg_api::DCFG_BASIC_BLOCK::get_basic_block_id () const [pure virtual]

Get basic-block ID number.

Basic-block ID numbers are unique within a process.

Returns

ID number of this basic block.

4.1.2.2 virtual UINT64 dcfg_api::DCFG_BASIC_BLOCK::get_exec_count () const [pure virtual]

Get dynamic execution count.

Returns

Number of times the block was executed, summed across all threads.

4.1.2.3 virtual UINT64 dcfg_api::DCFG_BASIC_BLOCK::get_exec_count_for_thread (UINT32 thread_id) const [pure virtual]

Get dynamic execution count.

Returns

Number of times the block was executed in given thread.

Parameters

in	<i>thread_id</i>	Thread number. Typically, threads are consecutively numbered from zero to DCFG_PROCESS::get_highest_thread_id() .
----	------------------	---

4.1.2.4 virtual UINT64 dcfg_api::DCFG_BASIC_BLOCK::get_first_instr_addr () const [pure virtual]

Get starting or base address.

Returns

Address of first instruction in this block.

4.1.2.5 virtual DCFG_ID dcfg_api::DCFG_BASIC_BLOCK::get_image_id () const [pure virtual]

Get the image ID.

Returns

Image ID of this block.

4.1.2.6 virtual DCFG_ID dcfg_api::DCFG_BASIC_BLOCK::get_inner_loop_id () const [pure virtual]

Get innermost loop ID.

To find all loops containing this block, get the innermost loop and then follow the parent loop IDs until there are no parents.

Returns

ID number of innermost loop containing this block or zero (0) if none.

4.1.2.7 virtual UINT64 dcfg_api::DCFG_BASIC_BLOCK::get_last_instr_addr () const [pure virtual]

Get the address of the last instruction.

This is *not* the address of the last byte in the block unless the last instruction is exactly one byte long. The address of the last byte is [DCFG_BASIC_BLOCK::get_first_instr_addr\(\)](#) + [DCFG_BASIC_BLOCK::get_size\(\)](#) - 1.

Returns

Address of last instruction in this block.

4.1.2.8 virtual UINT32 dcfg_api::DCFG_BASIC_BLOCK::get_num_instrs () const [pure virtual]

Get *static* number of instructions in the block.

To get the dynamic count of instructions executed, use [DCFG_GRAPH_BASE::get_instr_count\(\)](#) or [DCFG_GRAPH_BASE::get_instr_count_for_thread\(\)](#).

Returns

Static number of instructions in this block.

4.1.2.9 `virtual DCFG_ID dcfg_api::DCFG_BASIC_BLOCK::get_process_id () const [pure virtual]`

Get the process ID.

Returns

Process ID of this block.

4.1.2.10 `virtual DCFG_ID dcfg_api::DCFG_BASIC_BLOCK::get_routine_id () const [pure virtual]`

Get routine ID.

Returns

routine ID number of this block or zero (0) if none.

4.1.2.11 `virtual UINT32 dcfg_api::DCFG_BASIC_BLOCK::get_size () const [pure virtual]`

Get size.

Returns

Size of this block in bytes.

4.1.2.12 `virtual const std::string* dcfg_api::DCFG_BASIC_BLOCK::get_source_filename () const [pure virtual]`

Get name of source file for this block.

Returns

Pointer to name of the source filename at the base address of this block if it exists, `NULL` otherwise.

4.1.2.13 `virtual UINT32 dcfg_api::DCFG_BASIC_BLOCK::get_source_line_number () const [pure virtual]`

Get line number in source file this block.

Returns

Line number at the base address of this block if it exists, zero (0) otherwise.

4.1.2.14 `virtual const std::string* dcfg_api::DCFG_BASIC_BLOCK::get_symbol_name () const [pure virtual]`

Get symbol name of this block.

Returns

Pointer to name of the symbol at the base address of this block if one exists, `NULL` otherwise.

4.1.2.15 virtual UINT32 dcfg_api::DCFG_BASIC_BLOCK::get_symbol_offset () const [pure virtual]

Get symbol offset of this block.

Returns

Difference between base address of the symbol returned in [DCFG_BASIC_BLOCK::get_symbol_name\(\)](#) and the base address of this block or zero (0) if no symbol exists.

The documentation for this class was generated from the following file:

- dcfg_api.H

4.2 dcfg_api::DCFG_DATA Class Reference

Interface to all data in a DCFG.

```
#include <dcfg_api.H>
```

Public Member Functions

- virtual bool [read](#) (std::istream &strm, std::string &errMsg, bool readToEof=true)=0
Set internal DCFG data from a C++ istream.
- virtual bool [read](#) (const std::string filename, std::string &errMsg)=0
Open a file for reading and set internal DCFG data from its contents.
- virtual void [write](#) (std::ostream &strm) const =0
Write internal DCFG data to a C++ ostream.
- virtual bool [write](#) (const std::string &filename, std::string &errMsg) const =0
Write internal DCFG data to a file.
- virtual void [clearCounts](#) ()=0
Set all dynamic counts to zero.
- virtual UINT32 [get_process_ids](#) (DCFG_ID_CONTAINER &process_ids) const =0
Get list of process IDs.
- virtual [DCFG_PROCESS_CPTR](#) [get_process_info](#) (DCFG_ID process_id) const =0
Get access to data for a process.

Static Public Member Functions

- static [DCFG_DATA](#) * [new_dcfg](#) ()
Create a new DCFG.

4.2.1 Detailed Description

Interface to all data in a DCFG.

This is an interface; use [DCFG_DATA::new_dcfg\(\)](#) to create an object that implements the interface.

4.2.2 Member Function Documentation

4.2.2.1 `virtual UINT32 dcfg_api::DCFG_DATA::get_process_ids (DCFG_ID_CONTAINER & process_ids) const` [pure virtual]

Get list of process IDs.

Returns

Number of IDs that were added to `process_ids`.

Parameters

out	<i>process_ids</i>	Container to which process IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
-----	--------------------	--

4.2.2.2 `virtual DCFG_PROCESS_CPTR dcfg_api::DCFG_DATA::get_process_info (DCFG_ID process_id) const` [pure virtual]

Get access to data for a process.

Returns

Pointer to interface object for specified process or NULL if `process_id` is invalid.

Parameters

in	<i>process_id</i>	ID of desired process.
----	-------------------	------------------------

4.2.2.3 `static DCFG_DATA* dcfg_api::DCFG_DATA::new_dcfg ()` [static]

Create a new DCFG.

This is a factory method to create a new object that implements the [DCFG_DATA](#) interface.

Returns

Pointer to new object. It can be freed with `delete`.

4.2.2.4 `virtual bool dcfg_api::DCFG_DATA::read (std::istream & strm, std::string & errMsg, bool readToEof = true)` [pure virtual]

Set internal DCFG data from a C++ `istream`.

Returns

`true` on success, `false` otherwise (and sets `errMsg`).

Parameters

in	<i>strm</i>	Stream to read from. Reads one JSON value, which must follow the DCFG JSON format.
out	<i>errMsg</i>	Contains error message upon failure.
in	<i>readToEof</i>	Defines what to do after the JSON value is read. If <code>true</code> , continue reading to end of input stream and fail if any non-whitespace characters are found. If <code>false</code> , stop reading after the the JSON value.

4.2.2.5 `virtual bool dcfg_api::DCFG_DATA::read (const std::string filename, std::string & errMsg) [pure virtual]`

Open a file for reading and set internal DCFG data from its contents.

Returns

`true` on success, `false` otherwise (and sets `errMsg`).

Parameters

in	<i>filename</i>	Name of file to open. Reads one JSON value, which must follow the DCFG JSON format.
out	<i>errMsg</i>	Contains error message upon failure.

4.2.2.6 `virtual void dcfg_api::DCFG_DATA::write (std::ostream & strm) const [pure virtual]`

Write internal DCFG data to a C++ `ostream`.

Parameters

out	<i>strm</i>	Stream to write to. Output will conform to the DCFG JSON format.
-----	-------------	--

4.2.2.7 `virtual bool dcfg_api::DCFG_DATA::write (const std::string & filename, std::string & errMsg) const [pure virtual]`

Write internal DCFG data to a file.

Returns

`true` on success, `false` otherwise (and sets `errMsg`).

Parameters

in	<i>filename</i>	Name of file to open. Output will conform to the DCFG JSON format.
out	<i>errMsg</i>	Contains error message upon failure.

The documentation for this class was generated from the following file:

- `dcfg_api.H`

4.3 dcfg_api::DCFG_EDGE Class Reference

Interface to information about an edge between basic blocks and/or special nodes.

```
#include <dcfg_api.H>
```

Public Member Functions

- virtual DCFG_ID [get_edge_id](#) () const =0
Get ID number of edge.
- virtual DCFG_ID [get_source_node_id](#) () const =0
Get node ID of edge source.
- virtual DCFG_ID [get_target_node_id](#) () const =0
Get node ID of edge target.
- virtual UINT64 [get_exec_count](#) () const =0
Get edge count.
- virtual UINT64 [get_exec_count_for_thread](#) (UINT32 thread_id) const =0
Get edge count per thread.
- virtual const std::string * [get_edge_type](#) () const =0
Get edge type.
- virtual bool [is_any_branch_type](#) () const =0
Determine whether this edge is any type of branch.
- virtual bool [is_any_call_type](#) () const =0
Determine whether this edge is any type of call.
- virtual bool [is_any_return_type](#) () const =0
Determine whether this edge is any type of return.
- virtual bool [is_any_inter_routine_type](#) () const =0
Determine whether this edge is any type of call or return or an edge from the start node or to the exit node.
- virtual bool [is_any_bypass_type](#) () const =0
Determine whether this edge is any type of bypass, which is a "fabricated" edge across call/return pairs, etc.
- virtual bool [is_branch_edge_type](#) () const =0
Determine whether this edge is a branch edge type.
- virtual bool [is_call_edge_type](#) () const =0
Determine whether this edge is a call edge type.
- virtual bool [is_return_edge_type](#) () const =0
Determine whether this edge is a return edge type.
- virtual bool [is_call_bypass_edge_type](#) () const =0
Determine whether this edge is a call bypass edge type.
- virtual bool [is_conditional_branch_edge_type](#) () const =0
Determine whether this edge is a conditional branch edge type.
- virtual bool [is_context_bypass_edge_type](#) () const =0
Determine whether this edge is a context bypass edge type.
- virtual bool [is_context_edge_type](#) () const =0
Determine whether this edge is a context edge type.
- virtual bool [is_context_return_edge_type](#) () const =0
Determine whether this edge is a context return edge type.
- virtual bool [is_direct_branch_edge_type](#) () const =0

- Determine whether this edge is a direct branch edge type.*

 - virtual bool `is_direct_call_edge_type` () const =0
- Determine whether this edge is a direct call edge type.*

 - virtual bool `is_direct_conditional_branch_edge_type` () const =0
- Determine whether this edge is a direct conditional branch edge type.*

 - virtual bool `is_direct_unconditional_branch_edge_type` () const =0
- Determine whether this edge is a direct unconditional branch edge type.*

 - virtual bool `is_entry_edge_type` () const =0
- Determine whether this edge is an entry edge type.*

 - virtual bool `is_excluded_bypass_edge_type` () const =0
- Determine whether this edge is an excluded bypass edge type.*

 - virtual bool `is_exit_edge_type` () const =0
- Determine whether this edge is an exit edge type.*

 - virtual bool `is_fall_thru_edge_type` () const =0
- Determine whether this edge is a fall thru edge type.*

 - virtual bool `is_indirect_branch_edge_type` () const =0
- Determine whether this edge is an indirect branch edge type.*

 - virtual bool `is_indirect_call_edge_type` () const =0
- Determine whether this edge is an indirect call edge type.*

 - virtual bool `is_indirect_conditional_branch_edge_type` () const =0
- Determine whether this edge is an indirect conditional branch edge type.*

 - virtual bool `is_indirect_unconditional_branch_edge_type` () const =0
- Determine whether this edge is an indirect unconditional branch edge type.*

 - virtual bool `is_rep_edge_type` () const =0
- Determine whether this edge is a rep-prefix edge type.*

 - virtual bool `is_sys_call_bypass_edge_type` () const =0
- Determine whether this edge is a system call bypass edge type.*

 - virtual bool `is_sys_call_edge_type` () const =0
- Determine whether this edge is a system call edge type.*

 - virtual bool `is_sys_return_edge_type` () const =0
- Determine whether this edge is a system return edge type.*

 - virtual bool `is_unconditional_branch_edge_type` () const =0
- Determine whether this edge is an unconditional branch edge type.*

 - virtual bool `is_unknown_edge_type` () const =0
- Determine whether this edge is an unknown edge type.*

4.3.1 Detailed Description

Interface to information about an edge between basic blocks and/or special nodes.

4.3.2 Member Function Documentation

4.3.2.1 virtual DCFG_ID dcfg_api::DCFG_EDGE::get_edge_id () const [pure virtual]

Get ID number of edge.

Returns

ID number for this edge, unique within a process.

4.3.2.2 `virtual const std::string* dcfg_api::DCFG_EDGE::get_edge_type () const [pure virtual]`

Get edge type.

Returns

Pointer to string describing edge type per DCFG format documentation or NULL if type data is internally inconsistent (should not happen).

4.3.2.3 `virtual UINT64 dcfg_api::DCFG_EDGE::get_exec_count () const [pure virtual]`

Get edge count.

Returns

Number of times edge was taken, summed across all threads.

4.3.2.4 `virtual UINT64 dcfg_api::DCFG_EDGE::get_exec_count_for_thread (UINT32 thread_id) const [pure virtual]`

Get edge count per thread.

Returns

Number of times edge was taken on given thread.

Parameters

in	<i>thread_id</i>	Thread number. Typically, threads are consecutively numbered from zero to DCFG_PROCESS::get_highest_thread_id() .
----	------------------	---

4.3.2.5 `virtual DCFG_ID dcfg_api::DCFG_EDGE::get_source_node_id () const [pure virtual]`

Get node ID of edge source.

This is the node node the edge is "coming from". Most node IDs correspond to basic-blocks, but they could also be for special nodes. In particular, a source node ID could be the "START" node.

Returns

ID of source node.

4.3.2.6 `virtual DCFG_ID dcfg_api::DCFG_EDGE::get_target_node_id () const [pure virtual]`

Get node ID of edge target.

This is the node node the edge is "going to". Most node IDs correspond to basic-blocks, but they could also be for special nodes. In particular, a source node ID could be the "START" node.

Returns

ID of target node.

4.3.2.7 virtual bool dcfg_api::DCFG_EDGE::is_any_branch_type () const [pure virtual]

Determine whether this edge is *any* type of branch.

Returns

true if branch, false otherwise.

4.3.2.8 virtual bool dcfg_api::DCFG_EDGE::is_any_bypass_type () const [pure virtual]

Determine whether this edge is *any* type of bypass, which is a "fabricated" edge across call/return pairs, etc. See the DCFG documentation for more information on bypasses.

Returns

true if bypass, false otherwise.

4.3.2.9 virtual bool dcfg_api::DCFG_EDGE::is_any_call_type () const [pure virtual]

Determine whether this edge is *any* type of call.

This includes routine calls, system calls, etc.

Returns

true if call, false otherwise.

4.3.2.10 virtual bool dcfg_api::DCFG_EDGE::is_any_inter_routine_type () const [pure virtual]

Determine whether this edge is *any* type of call or return *or* an edge from the start node or to the exit node.

Returns

true if inter-routine, false otherwise.

4.3.2.11 virtual bool dcfg_api::DCFG_EDGE::is_any_return_type () const [pure virtual]

Determine whether this edge is *any* type of return.

This includes routine returns, system returns, etc.

Returns

true if return, false otherwise.

4.3.2.12 virtual bool dcfg_api::DCFG_EDGE::is_branch_edge_type () const [pure virtual]

Determine whether this edge is a branch edge type.

Returns

true if branch edge, false otherwise.

4.3.2.13 `virtual bool dcfg_api::DCFG_EDGE::is_call_bypass_edge_type () const [pure virtual]`

Determine whether this edge is a call bypass edge type.

Returns

`true` if call bypass edge, `false` otherwise.

4.3.2.14 `virtual bool dcfg_api::DCFG_EDGE::is_call_edge_type () const [pure virtual]`

Determine whether this edge is a call edge type.

Returns

`true` if call edge, `false` otherwise.

4.3.2.15 `virtual bool dcfg_api::DCFG_EDGE::is_conditional_branch_edge_type () const [pure virtual]`

Determine whether this edge is a conditional branch edge type.

Returns

`true` if conditional branch edge, `false` otherwise.

4.3.2.16 `virtual bool dcfg_api::DCFG_EDGE::is_context_bypass_edge_type () const [pure virtual]`

Determine whether this edge is a context bypass edge type.

Returns

`true` if context bypass edge, `false` otherwise.

4.3.2.17 `virtual bool dcfg_api::DCFG_EDGE::is_context_edge_type () const [pure virtual]`

Determine whether this edge is a context edge type.

Returns

`true` if context edge, `false` otherwise.

4.3.2.18 `virtual bool dcfg_api::DCFG_EDGE::is_context_return_edge_type () const [pure virtual]`

Determine whether this edge is a context return edge type.

Returns

`true` if context return edge, `false` otherwise.

4.3.2.19 `virtual bool dcfg_api::DCFG_EDGE::is_direct_branch_edge_type () const` [pure virtual]

Determine whether this edge is a direct branch edge type.

Returns

`true` if direct branch edge, `false` otherwise.

4.3.2.20 `virtual bool dcfg_api::DCFG_EDGE::is_direct_call_edge_type () const` [pure virtual]

Determine whether this edge is a direct call edge type.

Returns

`true` if direct call edge, `false` otherwise.

4.3.2.21 `virtual bool dcfg_api::DCFG_EDGE::is_direct_conditional_branch_edge_type () const` [pure virtual]

Determine whether this edge is a direct conditional branch edge type.

Returns

`true` if direct conditional branch edge, `false` otherwise.

4.3.2.22 `virtual bool dcfg_api::DCFG_EDGE::is_direct_unconditional_branch_edge_type () const` [pure virtual]

Determine whether this edge is a direct unconditional branch edge type.

Returns

`true` if direct unconditional branch edge, `false` otherwise.

4.3.2.23 `virtual bool dcfg_api::DCFG_EDGE::is_entry_edge_type () const` [pure virtual]

Determine whether this edge is an entry edge type.

Returns

`true` if entry edge, `false` otherwise.

4.3.2.24 `virtual bool dcfg_api::DCFG_EDGE::is_excluded_bypass_edge_type () const` [pure virtual]

Determine whether this edge is an excluded bypass edge type.

Returns

`true` if excluded bypass edge, `false` otherwise.

4.3.2.25 `virtual bool dcfg_api::DCFG_EDGE::is_exit_edge_type () const` [pure virtual]

Determine whether this edge is an exit edge type.

Returns

`true` if exit edge, `false` otherwise.

4.3.2.26 `virtual bool dcfg_api::DCFG_EDGE::is_fall_thru_edge_type () const` [pure virtual]

Determine whether this edge is a fall thru edge type.

Returns

`true` if fall thru edge, `false` otherwise.

4.3.2.27 `virtual bool dcfg_api::DCFG_EDGE::is_indirect_branch_edge_type () const` [pure virtual]

Determine whether this edge is an indirect branch edge type.

Returns

`true` if indirect branch edge, `false` otherwise.

4.3.2.28 `virtual bool dcfg_api::DCFG_EDGE::is_indirect_call_edge_type () const` [pure virtual]

Determine whether this edge is an indirect call edge type.

Returns

`true` if indirect call edge, `false` otherwise.

4.3.2.29 `virtual bool dcfg_api::DCFG_EDGE::is_indirect_conditional_branch_edge_type () const` [pure virtual]

Determine whether this edge is an indirect conditional branch edge type.

Returns

`true` if indirect conditional branch edge, `false` otherwise.

4.3.2.30 `virtual bool dcfg_api::DCFG_EDGE::is_indirect_unconditional_branch_edge_type () const` [pure virtual]

Determine whether this edge is an indirect unconditional branch edge type.

Returns

`true` if indirect unconditional branch edge, `false` otherwise.

4.3.2.31 `virtual bool dcfg_api::DCFG_EDGE::is_rep_edge_type () const [pure virtual]`

Determine whether this edge is a rep-prefix edge type.

Returns

`true` if rep edge, `false` otherwise.

4.3.2.32 `virtual bool dcfg_api::DCFG_EDGE::is_return_edge_type () const [pure virtual]`

Determine whether this edge is a return edge type.

Returns

`true` if return edge, `false` otherwise.

4.3.2.33 `virtual bool dcfg_api::DCFG_EDGE::is_sys_call_bypass_edge_type () const [pure virtual]`

Determine whether this edge is a system call bypass edge type.

Returns

`true` if sys call bypass edge, `false` otherwise.

4.3.2.34 `virtual bool dcfg_api::DCFG_EDGE::is_sys_call_edge_type () const [pure virtual]`

Determine whether this edge is a system call edge type.

Returns

`true` if sys call edge, `false` otherwise.

4.3.2.35 `virtual bool dcfg_api::DCFG_EDGE::is_sys_return_edge_type () const [pure virtual]`

Determine whether this edge is a system return edge type.

Returns

`true` if sys return edge, `false` otherwise.

4.3.2.36 `virtual bool dcfg_api::DCFG_EDGE::is_unconditional_branch_edge_type () const [pure virtual]`

Determine whether this edge is an unconditional branch edge type.

Returns

`true` if unconditional branch edge, `false` otherwise.

4.3.2.37 `virtual bool dcfg_api::DCFG_EDGE::is_unknown_edge_type () const` [pure virtual]

Determine whether this edge is an unknown edge type.

Returns

`true` if unknown edge, `false` otherwise.

The documentation for this class was generated from the following file:

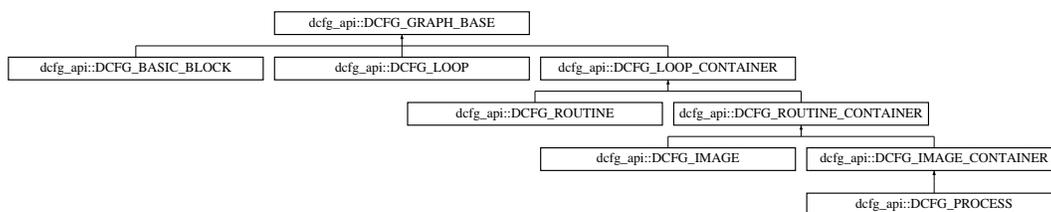
- `dcfg_api.H`

4.4 dcfg_api::DCFG_GRAPH_BASE Class Reference

Common interface to any structure containing nodes and edges between them, i.e., processes, images, routines, loops and basic blocks.

```
#include <dcfg_api.H>
```

Inheritance diagram for `dcfg_api::DCFG_GRAPH_BASE`:



Public Member Functions

- virtual `UINT32 get_basic_block_ids (DCFG_ID_CONTAINER &node_ids) const =0`
Get IDs of all basic blocks in the structure.
- virtual `UINT32 get_internal_edge_ids (DCFG_ID_CONTAINER &edge_ids) const =0`
Get list of internal edge IDs.
- virtual `UINT32 get_inbound_edge_ids (DCFG_ID_CONTAINER &edge_ids) const =0`
Get list of in-bound edge IDs.
- virtual `UINT32 get_outbound_edge_ids (DCFG_ID_CONTAINER &edge_ids) const =0`
Get list of out-bound edge IDs.
- virtual `UINT64 get_instr_count () const =0`
Get the total dynamic instruction count.
- virtual `UINT64 get_instr_count_for_thread (UINT32 thread_id) const =0`
Get per-thread dynamic instruction count.

4.4.1 Detailed Description

Common interface to any structure containing nodes and edges between them, i.e., processes, images, routines, loops and basic blocks.

A single basic block is a special case consisting of one block and no internal edges. Most nodes correspond to basic blocks of the executed binary, but some nodes are "special". See the DCFG documentation for more information.

4.4.2 Member Function Documentation

4.4.2.1 `virtual UINT32 dcfg_api::DCFG_GRAPH_BASE::get_basic_block_ids (DCFG_ID_CONTAINER & node_ids) const`
[pure virtual]

Get IDs of all basic blocks in the structure.

Returns

Number of IDs that were added to `node_ids`.

Parameters

<code>out</code>	<code>node_ids</code>	Container to which IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
------------------	-----------------------	--

4.4.2.2 `virtual UINT32 dcfg_api::DCFG_GRAPH_BASE::get_inbound_edge_ids (DCFG_ID_CONTAINER & edge_ids) const`
[pure virtual]

Get list of in-bound edge IDs.

These are all edges such that the source node *is not* within the structure and the target node *is* within the structure. Note that this set contains *all* the edges that terminate within the structure, including returns from calls, interrupts, etc., not only those that are considered to "enter" the structure.

Returns

Number of IDs that were added to `edge_ids`.

Parameters

<code>out</code>	<code>edge_ids</code>	Container to which IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
------------------	-----------------------	--

4.4.2.3 `virtual UINT64 dcfg_api::DCFG_GRAPH_BASE::get_instr_count () const` [pure virtual]

Get the total dynamic instruction count.

Returns

Count of instructions executed in this structure across all threads.

4.4.2.4 `virtual UINT64 dcfg_api::DCFG_GRAPH_BASE::get_instr_count_for_thread (UINT32 thread_id) const` [pure virtual]

Get per-thread dynamic instruction count.

Returns

Count of instructions executed in this structure on specified thread or zero (0) if thread is invalid.

Parameters

in	<i>thread_id</i>	Thread number. Typically, threads are consecutively numbered from zero to DCFG_PROCESS::get_highest_thread_id() .
----	------------------	---

4.4.2.5 `virtual UINT32 dcfg_api::DCFG_GRAPH_BASE::get_internal_edge_ids (DCFG_ID_CONTAINER & edge_ids) const`
`[pure virtual]`

Get list of internal edge IDs.

These are all edges such that both the source and target nodes are within the structure.

Returns

Number of IDs that were added to `edge_ids`.

Parameters

out	<i>edge_ids</i>	Container to which IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
-----	-----------------	--

4.4.2.6 `virtual UINT32 dcfg_api::DCFG_GRAPH_BASE::get_outbound_edge_ids (DCFG_ID_CONTAINER & edge_ids) const`
`[pure virtual]`

Get list of out-bound edge IDs.

These are all edges such that the source node *is* within the structure and the target node *is not* within the structure. Note that this set contains *all* the edges that originate within the structure, including calls, interrupts, etc., not only those that are considered to "exit" the structure.

Returns

Number of IDs that were added to `edge_ids`.

Parameters

out	<i>edge_ids</i>	Container to which edge IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
-----	-----------------	---

The documentation for this class was generated from the following file:

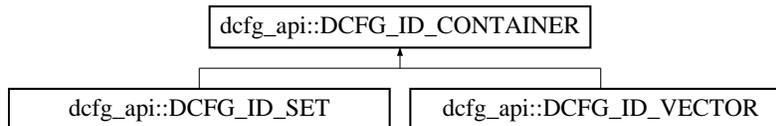
- `dcfg_api.H`

4.5 dcfg_api::DCFG_ID_CONTAINER Class Reference

Interface for any container of ID numbers.

```
#include <dcfg_api.H>
```

Inheritance diagram for dcfg_api::DCFG_ID_CONTAINER:



Public Member Functions

- virtual void [add_id](#) (DCFG_ID id)=0
Add one ID to the container.

4.5.1 Detailed Description

Interface for any container of ID numbers.

The API programmer is free to define and use any class that implements this interface to collect ID numbers (process IDs, basic blocks, etc.) from the relevant DCFG APIs.

4.5.2 Member Function Documentation

4.5.2.1 virtual void dcfg_api::DCFG_ID_CONTAINER::add_id (DCFG_ID *id*) [pure virtual]

Add one ID to the container.

All concrete implementations must define this method.

Parameters

<i>in</i>	<i>id</i>	ID to add.
-----------	-----------	------------

Implemented in [dcfg_api::DCFG_ID_SET](#), and [dcfg_api::DCFG_ID_VECTOR](#).

The documentation for this class was generated from the following file:

- [dcfg_api.H](#)

4.6 dcfg_api::DCFG_ID_SET Class Reference

Set of ID numbers.

```
#include <dcfg_api.H>
```

Inheritance diagram for dcfg_api::DCFG_ID_SET:



Public Member Functions

- virtual void [add_id](#) (DCFG_ID id)
Add one ID to the container.

Additional Inherited Members

4.6.1 Detailed Description

Set of ID numbers.

This is an example of a [DCFG_ID_CONTAINER](#) implementation based on an STL `set`. This is useful for iterating through the added IDs in numerical order and checking whether a certain ID exists.

4.6.2 Member Function Documentation

4.6.2.1 virtual void `dcfg_api::DCFG_ID_SET::add_id (DCFG_ID id)` `[inline]`, `[virtual]`

Add one ID to the container.

All concrete implementations must define this method.

Implements [dcfg_api::DCFG_ID_CONTAINER](#).

The documentation for this class was generated from the following file:

- `dcfg_api.H`

4.7 dcfg_api::DCFG_ID_VECTOR Class Reference

Vector of ID numbers.

```
#include <dcfg_api.H>
```

Inheritance diagram for `dcfg_api::DCFG_ID_VECTOR`:



Public Member Functions

- virtual void [add_id](#) (DCFG_ID id)

Add one ID to the container.

Additional Inherited Members

4.7.1 Detailed Description

Vector of ID numbers.

This is an example of a [DCFG_ID_CONTAINER](#) implementation based on an STL `vector`. This is useful when it is important to maintain the order in which elements are added.

4.7.2 Member Function Documentation

4.7.2.1 `virtual void dcfg_api::DCFG_ID_VECTOR::add_id (DCFG_ID id) [inline],[virtual]`

Add one ID to the container.

All concrete implementations must define this method.

Implements [dcfg_api::DCFG_ID_CONTAINER](#).

The documentation for this class was generated from the following file:

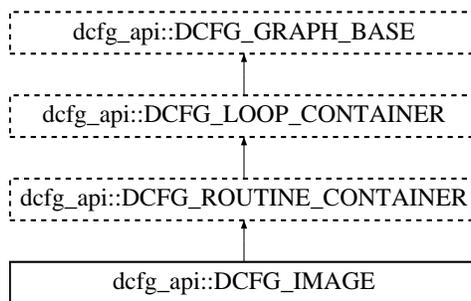
- `dcfg_api.H`

4.8 dcfg_api::DCFG_IMAGE Class Reference

Interface to information about a binary image within a process.

```
#include <dcfg_api.H>
```

Inheritance diagram for `dcfg_api::DCFG_IMAGE`:



Public Member Functions

- virtual DCFG_ID [get_process_id](#) () const =0
Get the process ID.
- virtual DCFG_ID [get_image_id](#) () const =0
Get the image ID.
- virtual const std::string * [get_filename](#) () const =0

Get the filename of the image.

- virtual UINT64 `get_base_address` () const =0

Get base address of image.

- virtual UINT64 `get_size` () const =0

Get size of image.

- virtual UINT32 `get_basic_block_ids_by_addr` (UINT64 *addr*, DCFG_ID_CONTAINER &*node_ids*) const =0

Get basic block ID(s) containing given address in this image.

4.8.1 Detailed Description

Interface to information about a binary image within a process.

4.8.2 Member Function Documentation

4.8.2.1 virtual UINT64 `dcfg_api::DCFG_IMAGE::get_base_address` () const [pure virtual]

Get base address of image.

Returns

Address where image was loaded into memory by O/S.

4.8.2.2 virtual UINT32 `dcfg_api::DCFG_IMAGE::get_basic_block_ids_by_addr` (UINT64 *addr*, DCFG_ID_CONTAINER & *node_ids*) const [pure virtual]

Get basic block ID(s) containing given address in this image.

It is possible to get zero or more IDs returned: zero if the address appears in no basic blocks, one if it appears in exactly one block in one image, and more than one if it is not unique. Basic blocks may not be unique if an image uses self-modifying code (SMC) or other mechanisms that replace code regions. For most images, this will not be the case, and addresses will be unique for a given image.

Returns

Number of IDs that were added to `node_ids`.

Parameters

in	<i>addr</i>	Virtual address that can appear anywhere within a basic block.
out	<i>node_ids</i>	Container to which basic-block IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER: DCFG_ID_VECTOR, DCFG_ID_SET, etc.

4.8.2.3 virtual const std::string* `dcfg_api::DCFG_IMAGE::get_filename` () const [pure virtual]

Get the filename of the image.

Returns

Pointer to string containing full pathname of image or base name if pathname not available or NULL if no name is available.

4.8.2.4 `virtual DCFG_ID dcfg_api::DCFG_IMAGE::get_image_id () const [pure virtual]`

Get the image ID.

Returns

ID of this image.

4.8.2.5 `virtual DCFG_ID dcfg_api::DCFG_IMAGE::get_process_id () const [pure virtual]`

Get the process ID.

Returns

Process ID of this image.

4.8.2.6 `virtual UINT64 dcfg_api::DCFG_IMAGE::get_size () const [pure virtual]`

Get size of image.

Returns

Size of image as loaded into memory by O/S, in bytes.

The documentation for this class was generated from the following file:

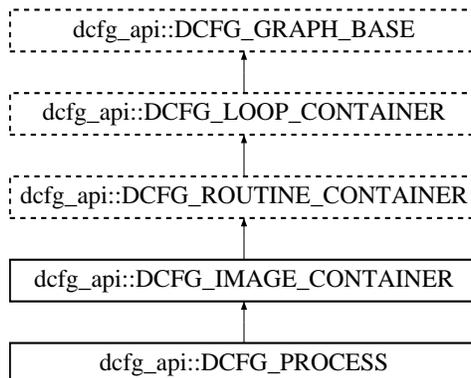
- dcfg_api.H

4.9 dcfg_api::DCFG_IMAGE_CONTAINER Class Reference

Common interface to any structure containing images, i.e., processes.

```
#include <dcfg_api.H>
```

Inheritance diagram for dcfg_api::DCFG_IMAGE_CONTAINER:



Public Member Functions

- virtual UINT32 [get_image_ids](#) (DCFG_ID_CONTAINER &image_ids) const =0
Get the set of image IDs.
- virtual [DCFG_IMAGE_CPTR get_image_info](#) (DCFG_ID image_id) const =0
Get access to data for an image.

4.9.1 Detailed Description

Common interface to any structure containing images, i.e., processes.

4.9.2 Member Function Documentation

4.9.2.1 virtual UINT32 `dcfg_api::DCFG_IMAGE_CONTAINER::get_image_ids (DCFG_ID_CONTAINER & image_ids) const`
[pure virtual]

Get the set of image IDs.

Get IDs of all images seen, not just the ones that are active at any given time. The address ranges of two or more of the the images may overlap if an image was loaded after another was unloaded.

Returns

Number of IDs that were added to `image_ids`.

Parameters

out	<i>image_ids</i>	Container to which image IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
-----	------------------	--

4.9.2.2 virtual [DCFG_IMAGE_CPTR dcfg_api::DCFG_IMAGE_CONTAINER::get_image_info \(DCFG_ID image_id \) const](#)
[pure virtual]

Get access to data for an image.

Returns

Pointer to interface object for specified image or NULL if `image_id` is invalid.

Parameters

in	<i>image_id</i>	ID of desired image.
----	-----------------	----------------------

The documentation for this class was generated from the following file:

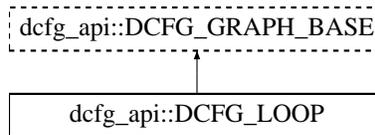
- `dcfg_api.H`

4.10 dcfg_api::DCFG_LOOP Class Reference

Interface to information about a loop.

```
#include <dcfg_api.H>
```

Inheritance diagram for dcfg_api::DCFG_LOOP:



Public Member Functions

- virtual DCFG_ID [get_process_id](#) () const =0
Get the process ID.
- virtual DCFG_ID [get_image_id](#) () const =0
Get the image ID.
- virtual DCFG_ID [get_routine_id](#) () const =0
Get routine ID.
- virtual DCFG_ID [get_loop_id](#) () const =0
Get loop ID, which equals the basic-block ID of the head node.
- virtual UINT32 [get_entry_edge_ids](#) (DCFG_ID_CONTAINER &edge_ids) const =0
Get set of IDs of the entry edges.
- virtual UINT32 [get_exit_edge_ids](#) (DCFG_ID_CONTAINER &edge_ids) const =0
Get set of IDs of the exit edges.
- virtual UINT32 [get_back_edge_ids](#) (DCFG_ID_CONTAINER &edge_ids) const =0
Get set of IDs of the back-edges.
- virtual DCFG_ID [get_parent_loop_id](#) () const =0
Get head node ID of most immediate containing loop, if any.
- virtual UINT64 [get_iteration_count](#) () const =0
Get dynamic iteration count.
- virtual UINT64 [get_iteration_count_for_thread](#) (UINT32 thread_id) const =0
Get dynamic execution count per thread.

4.10.1 Detailed Description

Interface to information about a loop.

4.10.2 Member Function Documentation

4.10.2.1 virtual UINT32 dcfg_api::DCFG_LOOP::get_back_edge_ids (DCFG_ID_CONTAINER & edge_ids) const [pure virtual]

Get set of IDs of the back-edges.

These are the edges that are traversed when a loop is repeated following an entry. The target node of each back edge will be the head node of the loop by definition. By definition, an edge $n \rightarrow h$ is a back edge if h dominates n , where h is the head node.

Returns

Number of IDs that were added to `edge_ids`.

Parameters

<code>out</code>	<code>edge_ids</code>	Container to which edge IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
------------------	-----------------------	---

```
4.10.2.2 virtual UINT32 dcfg_api::DCFG_LOOP::get_entry_edge_ids ( DCFG_ID_CONTAINER & edge_ids ) const [pure virtual]
```

Get set of IDs of the entry edges.

These are the edges that are traversed when a loop is entered from somewhere outside the loop. This set does *not* include back edges.

Returns

Number of IDs that were added to `edge_ids`.

Parameters

<code>out</code>	<code>edge_ids</code>	Container to which edge IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
------------------	-----------------------	---

```
4.10.2.3 virtual UINT32 dcfg_api::DCFG_LOOP::get_exit_edge_ids ( DCFG_ID_CONTAINER & edge_ids ) const [pure virtual]
```

Get set of IDs of the exit edges.

These are the edges that are traversed when a loop is exited. This set does *not* include call edges from the loop. If you also want call edges, use [DCFG_GRAPH_BASE::get_outbound_edge_ids\(\)](#). Note that any given edge may exit more than one loop when loops are nested.

Returns

Number of IDs that were added to `edge_ids`.

Parameters

<code>out</code>	<code>edge_ids</code>	Container to which edge IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
------------------	-----------------------	---

4.10.2.4 `virtual DCFG_ID dcfg_api::DCFG_LOOP::get_image_id () const [pure virtual]`

Get the image ID.

Returns

Image ID of this loop.

4.10.2.5 `virtual UINT64 dcfg_api::DCFG_LOOP::get_iteration_count () const [pure virtual]`

Get dynamic iteration count.

This is the number of times the loop was executed, including entry from outside the loop and via its back edges. By definition, a loop can only be entered at its head node.

Returns

Number of times loop was executed, summed across all threads.

4.10.2.6 `virtual UINT64 dcfg_api::DCFG_LOOP::get_iteration_count_for_thread (UINT32 thread_id) const [pure virtual]`

Get dynamic execution count per thread.

See [DCFG_LOOP::get_iteration_count\(\)](#) for iteration-count definition.

Returns

Number of times loop was executed in given thread.

Parameters

<code>in</code>	<code>thread_id</code>	Thread number. Typically, threads are consecutively numbered from zero to DCFG_PROCESS::get_highest_thread_id() .
-----------------	------------------------	---

4.10.2.7 `virtual DCFG_ID dcfg_api::DCFG_LOOP::get_loop_id () const [pure virtual]`

Get loop ID, which equals the basic-block ID of the head node.

The head node is the common target of all the back edges in the loop.

Returns

ID number of head node.

4.10.2.8 `virtual DCFG_ID dcfg_api::DCFG_LOOP::get_parent_loop_id () const [pure virtual]`

Get head node ID of most immediate containing loop, if any.

This indicates loop nesting.

Returns

ID number of head node of parent loop or zero (0) if there is no parent loop.

4.10.2.9 `virtual DCFG_ID dcfg_api::DCFG_LOOP::get_process_id() const` [pure virtual]

Get the process ID.

Returns

Process ID of this loop.

4.10.2.10 `virtual DCFG_ID dcfg_api::DCFG_LOOP::get_routine_id() const` [pure virtual]

Get routine ID.

Returns

routine ID number of this loop.

The documentation for this class was generated from the following file:

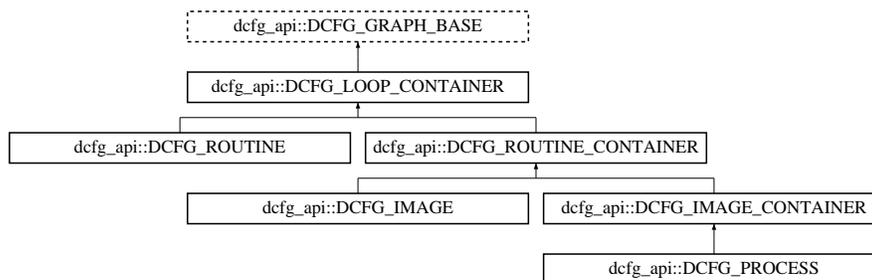
- dcfg_api.H

4.11 dcfg_api::DCFG_LOOP_CONTAINER Class Reference

Common interface to any structure containing loops, i.e., routines, images, and processes.

```
#include <dcfg_api.H>
```

Inheritance diagram for dcfg_api::DCFG_LOOP_CONTAINER:

**Public Member Functions**

- virtual `UINT32 get_loop_ids (DCFG_ID_CONTAINER &node_ids) const =0`
Get the set of loop IDs.
- virtual `DCFG_LOOP_CPTR get_loop_info (DCFG_ID loop_id) const =0`
Get access to data for a loop.

4.11.1 Detailed Description

Common interface to any structure containing loops, i.e., routines, images, and processes.

Note: even though loops can be nested, a loop is not considered a [DCFG_LOOP_CONTAINER](#). Loop nesting structure can be determined by querying the parent loop id from a [DCFG_LOOP](#) object.

4.11.2 Member Function Documentation

4.11.2.1 `virtual UINT32 dcfg_api::DCFG_LOOP_CONTAINER::get_loop_ids (DCFG_ID_CONTAINER & node_ids) const` [pure virtual]

Get the set of loop IDs.

Get IDs of all loops in the structure. A loop ID is the same as the ID of the basic block at its head node.

Returns

Number of IDs that were added to `node_ids`.

Parameters

<code>out</code>	<code>node_ids</code>	Container to which IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
------------------	-----------------------	--

4.11.2.2 `virtual DCFG_LOOP_CPTR dcfg_api::DCFG_LOOP_CONTAINER::get_loop_info (DCFG_ID loop_id) const` [pure virtual]

Get access to data for a loop.

Returns

Pointer to interface object for specified loop or NULL if `loop_id` is invalid.

Parameters

<code>in</code>	<code>loop_id</code>	ID of desired loop.
-----------------	----------------------	---------------------

The documentation for this class was generated from the following file:

- `dcfg_api.H`

4.12 dcfg_pin_api::DCFG_PIN_MANAGER Class Reference

Connection between a Pin tool and a DCFG_DATA object.

```
#include <dcfg_pin_api.H>
```

Public Member Functions

- virtual bool `dcfg_enable_knob` () const
Whether the '-dcfg' knob was used on the command-line.
- virtual void `activate` ()=0
Initialize and add Pin instrumentation.
- virtual void `activate` (void *pinplay_engine)=0
Initialize and add PinPlay instrumentation.
- virtual `dcfg_api::DCFG_DATA_CPTR get_dcfg_data` () const =0
Get access to DCFG data being constructed by the Pin tool.
- virtual void `set_cfg_collection` (bool enable)=0
Explicitly set CFG-data collection.

Static Public Member Functions

- static `DCFG_PIN_MANAGER * new_manager` ()
Create a new DCFG_PIN_MANAGER.

4.12.1 Detailed Description

Connection between a Pin tool and a DCFG_DATA object.

This is an interface; use `DCFG_PIN_MANAGER::new_manager()` to create an object that implements the interface.

4.12.2 Member Function Documentation

4.12.2.1 `virtual void dcfg_pin_api::DCFG_PIN_MANAGER::activate () [pure virtual]`

Initialize and add Pin instrumentation.

Default behavior depends on settings of dcfg knobs.

4.12.2.2 `virtual void dcfg_pin_api::DCFG_PIN_MANAGER::activate (void * pinplay_engine) [pure virtual]`

Initialize and add PinPlay instrumentation.

Default behavior depends on settings of dcfg knobs and whether logger, replayer or both are activated.

Parameters

in	<i>pinplay_engine</i>	pointer to existing pinplay engine or NULL if none.
----	-----------------------	---

4.12.2.3 `virtual bool dcfg_pin_api::DCFG_PIN_MANAGER::dcfg_enable_knob () const [virtual]`

Whether the '-dcfg' knob was used on the command-line.

Returns

true if '-dcfg' knob was used, false otherwise.

4.12.2.4 `virtual dcfg_api::DCFG_DATA_CPTR dcfg_pin_api::DCFG_PIN_MANAGER::get_dcfg_data () const` [pure virtual]

Get access to DCFG data being constructed by the Pin tool.

The returned DCFG will only be valid at the end of a region or program. There will not be a DCFG if an [activate\(\)](#) method has not been called.

Returns

Pointer to associated DCFG data or `NULL` if none.

4.12.2.5 `static DCFG_PIN_MANAGER* dcfg_pin_api::DCFG_PIN_MANAGER::new_manager ()` [static]

Create a new [DCFG_PIN_MANAGER](#).

This is a factory method to create a new object that implements the [DCFG_PIN_MANAGER](#) interface.

Returns

Pointer to new object. It can be freed with `delete`.

4.12.2.6 `virtual void dcfg_pin_api::DCFG_PIN_MANAGER::set_cfg_collection (bool enable)` [pure virtual]

Explicitly set CFG-data collection.

This controls whether control-flow instructions are instrumented to build a CFG. This is independent of whether a DCFG file is written. If a DCFG file is written with CFG collection disabled, it will have no CFG data in it.

Parameters

<code>in</code>	<code>enable</code>	turn CFG collection on or off.
-----------------	---------------------	--------------------------------

The documentation for this class was generated from the following file:

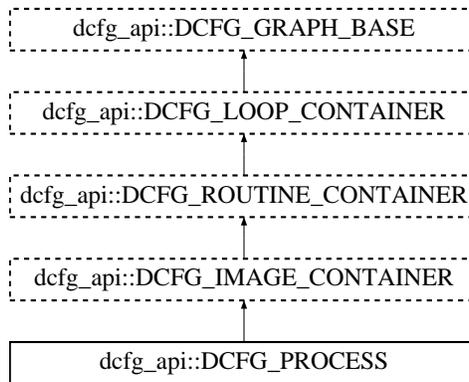
- `dcfg_pin_api.H`

4.13 dcfg_api::DCFG_PROCESS Class Reference

Interface to information about an O/S process.

```
#include <dcfg_api.H>
```

Inheritance diagram for `dcfg_api::DCFG_PROCESS`:



Public Member Functions

- virtual DCFG_ID [get_process_id](#) () const =0
Get the process ID.
- virtual UINT32 [get_highest_thread_id](#) () const =0
Get the highest thread ID.
- virtual UINT32 [get_basic_block_ids_by_addr](#) (UINT64 addr, [DCFG_ID_CONTAINER](#) &node_ids) const =0
Get basic block ID(s) containing given address in this process.
- virtual UINT32 [get_start_node_id](#) () const =0
Get ID of start node.
- virtual UINT32 [get_end_node_id](#) () const =0
Get ID of end node.
- virtual UINT32 [get_unknown_node_id](#) () const =0
Get ID of unknown node.
- virtual DCFG_ID [get_edge_id](#) (DCFG_ID source_node_id, DCFG_ID target_node_id) const =0
Get the ID of an edge given its source and target nodes.
- virtual UINT32 [get_successor_node_ids](#) (DCFG_ID source_node_id, [DCFG_ID_CONTAINER](#) &node_ids) const =0
Get the set of target nodes that have an edge from the given source.
- virtual UINT32 [get_predecessor_node_ids](#) (DCFG_ID target_node_id, [DCFG_ID_CONTAINER](#) &node_ids) const =0
Get the set of source nodes that have an edge to the given target.
- virtual [DCFG_EDGE_CPTR](#) [get_edge_info](#) (DCFG_ID edge_id) const =0
Get access to data for an edge.
- virtual [DCFG_BASIC_BLOCK_CPTR](#) [get_basic_block_info](#) (DCFG_ID node_id) const =0
Get access to data for a basic block.
- virtual bool [is_special_node](#) (DCFG_ID node_id) const =0
Determine whether a node ID refers to any "special" (non-basic-block) node.
- virtual bool [is_start_node](#) (DCFG_ID node_id) const =0
Determine whether a node ID refers to the special non-basic-block start node.
- virtual bool [is_end_node](#) (DCFG_ID node_id) const =0
Determine whether a node ID refers to the special non-basic-block end node.
- virtual bool [is_unknown_node](#) (DCFG_ID node_id) const =0
Determine whether a node ID refers to the special non-basic-block "unknown" node.

4.13.1 Detailed Description

Interface to information about an O/S process.

4.13.2 Member Function Documentation

4.13.2.1 `virtual UINT32 dcfg_api::DCFG_PROCESS::get_basic_block_ids_by_addr (UINT64 addr, DCFG_ID_CONTAINER & node_ids) const` [pure virtual]

Get basic block ID(s) containing given address in this process.

It is possible to get zero or more IDs returned: zero if the address appears in no basic blocks, one if it appears in exactly one block in one image, and more than one if it is not unique. Basic blocks may not be unique if a dynamically-linked process unloads one image and loads another image in an overlapping address region.

Returns

Number of IDs that were added to `node_ids`.

Parameters

<code>in</code>	<code><i>addr</i></code>	Virtual address that can appear anywhere within a basic block.
<code>out</code>	<code><i>node_ids</i></code>	Container to which basic-block IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of <code>DCFG_ID_CONTAINER</code> : <code>DCFG_ID_VECTOR</code> , <code>DCFG_ID_SET</code> , etc.

4.13.2.2 `virtual DCFG_BASIC_BLOCK_CPTR dcfg_api::DCFG_PROCESS::get_basic_block_info (DCFG_ID node_id) const` [pure virtual]

Get access to data for a basic block.

Returns

Pointer to interface object for specified basic block or NULL if `node_id` refers to a "special" node or is invalid.

Parameters

<code>in</code>	<code><i>node_id</i></code>	ID of desired basic block.
-----------------	-----------------------------	----------------------------

4.13.2.3 `virtual DCFG_ID dcfg_api::DCFG_PROCESS::get_edge_id (DCFG_ID source_node_id, DCFG_ID target_node_id) const` [pure virtual]

Get the ID of an edge given its source and target nodes.

Returns

ID number of edge or zero (0) if there is no edge between the two nodes.

Parameters

in	<i>source_node_id</i>	ID number of node the edge is coming from.
in	<i>target_node_id</i>	ID number of node the edge is going to.

4.13.2.4 `virtual DCFG_EDGE_CPTR dcfg_api::DCFG_PROCESS::get_edge_info (DCFG.ID edge_id) const` [pure virtual]

Get access to data for an edge.

Returns

Pointer to interface object for specified edge or NULL if *edge_id* is invalid.

Parameters

in	<i>edge_id</i>	ID of desired edge.
----	----------------	---------------------

4.13.2.5 `virtual UINT32 dcfg_api::DCFG_PROCESS::get_end_node_id () const` [pure virtual]

Get ID of end node.

This is a "special" node that is not a basic block. It is the target node of the edge from the last basic block executed in each thread.

Returns

ID number of end node.

4.13.2.6 `virtual UINT32 dcfg_api::DCFG_PROCESS::get_highest_thread_id () const` [pure virtual]

Get the highest thread ID.

The lowest thread ID is zero (0). Typically, threads are consecutively numbered from zero to [DCFG_PROCESS::get_highest_thread_id\(\)](#).

Returns

Highest thread ID recorded when the DCFG was created.

4.13.2.7 `virtual UINT32 dcfg_api::DCFG_PROCESS::get_predecessor_node_ids (DCFG.ID target_node_id, DCFG.ID_CONTAINER & node_ids) const` [pure virtual]

Get the set of source nodes that have an edge to the given target.

Predecessor node sets are used in various graph algorithms.

Returns

Number of IDs that were added to *node_ids*.

Parameters

in	<i>target_node_id</i>	ID number of target node.
out	<i>node_ids</i>	Container to which source node IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.

4.13.2.8 virtual `DCFG_ID dcfg_api::DCFG_PROCESS::get_process_id () const` [pure virtual]

Get the process ID.

Returns

Process ID captured when the DCFG was created.

4.13.2.9 virtual `UINT32 dcfg_api::DCFG_PROCESS::get_start_node_id () const` [pure virtual]

Get ID of start node.

This is a "special" node that is not a basic block. It is the source node of the edge to the first basic block executed in each thread.

Returns

ID number of start node.

4.13.2.10 virtual `UINT32 dcfg_api::DCFG_PROCESS::get_successor_node_ids (DCFG_ID source_node_id, DCFG_ID_CONTAINER & node_ids) const` [pure virtual]

Get the set of target nodes that have an edge from the given source.

Successor node sets are used in various graph algorithms.

Returns

Number of IDs that were added to `node_ids`.

Parameters

in	<i>source_node_id</i>	ID number of source node.
out	<i>node_ids</i>	Container to which target node IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.

4.13.2.11 virtual `UINT32 dcfg_api::DCFG_PROCESS::get_unknown_node_id () const` [pure virtual]

Get ID of unknown node.

This is a "special" node that is not a basic block. It is a placeholder for any section of executable code for which

basic-block data cannot be obtained. An unknown node should not appear in a well-formed graph.

Returns

ID number of the unknown node.

4.13.2.12 `virtual bool dcfg_api::DCFG_PROCESS::is_end_node (DCFG.ID node_id) const` [pure virtual]

Determine whether a node ID refers to the special non-basic-block end node.

Returns

`true` if end node, `false` otherwise.

Parameters

<code>in</code>	<code><i>node_id</i></code>	ID of node in question.
-----------------	-----------------------------	-------------------------

4.13.2.13 `virtual bool dcfg_api::DCFG_PROCESS::is_special_node (DCFG.ID node_id) const` [pure virtual]

Determine whether a node ID refers to any "special" (non-basic-block) node.

This could be a start, end, or unknown node. If this returns `false` it does not necessarily mean that the node is a basic-block; it could be that the ID is invalid.

Returns

`true` if node is special, `false` otherwise.

Parameters

<code>in</code>	<code><i>node_id</i></code>	ID of node in question.
-----------------	-----------------------------	-------------------------

4.13.2.14 `virtual bool dcfg_api::DCFG_PROCESS::is_start_node (DCFG.ID node_id) const` [pure virtual]

Determine whether a node ID refers to the special non-basic-block start node.

Returns

`true` if start node, `false` otherwise.

Parameters

<code>in</code>	<code><i>node_id</i></code>	ID of node in question.
-----------------	-----------------------------	-------------------------

4.13.2.15 `virtual bool dcfg_api::DCFG_PROCESS::is_unknown_node (DCFG.ID node_id) const` [pure virtual]

Determine whether a node ID refers to the special non-basic-block "unknown" node.

A well-formed DCFG should not have any unknown nodes.

Returns

`true` if unknown node, `false` otherwise.

Parameters

<code>in</code>	<code>node_id</code>	ID of node in question.
-----------------	----------------------	-------------------------

The documentation for this class was generated from the following file:

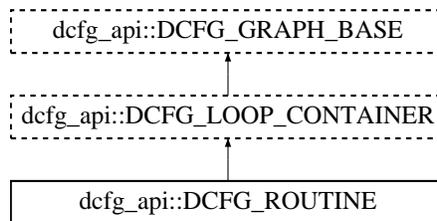
- `dcfg_api.H`

4.14 dcfg_api::DCFG_ROUTINE Class Reference

Interface to information about a routine in an image.

```
#include <dcfg_api.H>
```

Inheritance diagram for `dcfg_api::DCFG_ROUTINE`:

**Public Member Functions**

- virtual `DCFG_ID` `get_process_id ()` const =0
Get the process ID.
- virtual `DCFG_ID` `get_image_id ()` const =0
Get the image ID.
- virtual `DCFG_ID` `get_routine_id ()` const =0
Get routine ID, which equals the basic-block ID of the entry node.
- virtual const `std::string *` `get_symbol_name ()` const =0
Get symbol name of this routine.
- virtual `UINT32` `get_entry_edge_ids (DCFG_ID_CONTAINER &edge_ids)` const =0
Get set of entry edge IDs.
- virtual `UINT32` `get_exit_edge_ids (DCFG_ID_CONTAINER &edge_ids)` const =0
Get set of exit edge IDs.
- virtual `DCFG_ID` `get_idom_node_id (DCFG_ID node_id)` const =0
Get immediate dominator.
- virtual `UINT64` `get_entry_count ()` const =0
Get dynamic entry count.
- virtual `UINT64` `get_entry_count_for_thread (UINT32 thread_id)` const =0
Get dynamic entry count per thread.

4.14.1 Detailed Description

Interface to information about a routine in an image.

A routine is also known as a subroutine, function, or procedure.

4.14.2 Member Function Documentation

4.14.2.1 `virtual UINT64 dcfg_api::DCFG_ROUTINE::get_entry_count () const [pure virtual]`

Get dynamic entry count.

This is the number of times the routine was called or otherwise entered. By the DCFG definition, a routine can only be entered at its entry node. A call within a routine to its entry node is considered an entry (via recursion). If there is a branch within a routine to its entry node, it will also be considered an entry (this is unusual).

Returns

Number of times routine was entered, summed across all threads.

4.14.2.2 `virtual UINT64 dcfg_api::DCFG_ROUTINE::get_entry_count_for_thread (UINT32 thread_id) const [pure virtual]`

Get dynamic entry count per thread.

See [DCFG_ROUTINE::get_entry_count\(\)](#) for entry-count definition.

Returns

Number of times routine was entered in given thread.

Parameters

in	<i>thread_id</i>	Thread number. Typically, threads are consecutively numbered from zero to DCFG_PROCESS::get_highest_thread_id() .
----	------------------	---

4.14.2.3 `virtual UINT32 dcfg_api::DCFG_ROUTINE::get_entry_edge_ids (DCFG_ID_CONTAINER & edge_ids) const [pure virtual]`

Get set of entry edge IDs.

For routines, these are typically from "call" statements, but they can also include branches to routines for unstructured code. The source of an entry edge will be from another routine except in the case of direct recursion (call from routine to itself). The target of an entry edge will always be the entry node. This set does *not* include incoming return edges. If you also want return edges, use [DCFG_GRAPH_BASE::get_inbound_edge_ids\(\)](#).

Returns

Number of IDs that were added to `edge_ids`.

Parameters

out	<i>edge_ids</i>	Container to which exit edge IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
-----	-----------------	--

4.14.2.4 `virtual UINT32 dcfg_api::DCFG_ROUTINE::get_exit_edge_ids (DCFG_ID_CONTAINER & edge_ids) const [pure virtual]`

Get set of exit edge IDs.

For routines, these are typically from "return" statements, but they can also include branches out of routines for unstructured code. The target of an exit edge will be to another routine except for direct recursion. This set does *not* include outgoing call edges. If you also want call edges, use [DCFG_GRAPH_BASE::get_outbound_edge_ids\(\)](#).

Returns

Number of IDs that were added to *edge_ids*.

Parameters

out	<i>edge_ids</i>	Container to which exit edge IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
-----	-----------------	--

4.14.2.5 `virtual DCFG_ID dcfg_api::DCFG_ROUTINE::get_idom_node_id (DCFG_ID node_id) const [pure virtual]`

Get immediate dominator.

The immediate dominator (idom) is the last node before the given node that must be executed before the given node is executed. The idom of the entry node is itself. The idom must be within the routine, i.e., it does not consider edges between routines. Idoms relationships are used in many graph algorithms.

Returns

ID number of idom of *node_id* or zero (0) if *node_id* is not in this routine.

Parameters

in	<i>node_id</i>	ID number of <i>dominated</i> node.
----	----------------	-------------------------------------

4.14.2.6 `virtual DCFG_ID dcfg_api::DCFG_ROUTINE::get_image_id () const [pure virtual]`

Get the image ID.

Returns

Image ID of this routine.

4.14.2.7 `virtual DCFG_ID dcfg_api::DCFG_ROUTINE::get_process_id() const [pure virtual]`

Get the process ID.

Returns

Process ID of this routine.

4.14.2.8 `virtual DCFG_ID dcfg_api::DCFG_ROUTINE::get_routine_id() const [pure virtual]`

Get routine ID, which equals the basic-block ID of the entry node.

By the DCFG definition, a routine can only have one entry node. If there is a call into the "middle" of a routine, that entry point defines a separate routine in a DCFG.

Returns

ID number of entry node.

4.14.2.9 `virtual const std::string* dcfg_api::DCFG_ROUTINE::get_symbol_name() const [pure virtual]`

Get symbol name of this routine.

For more comprehensive symbol and source-code data, use [DCFG_BASIC_BLOCK::get_symbol_name\(\)](#), [DCFG_BASIC_BLOCK::get_source_filename\(\)](#), and [DCFG_BASIC_BLOCK::get_source_line_number\(\)](#) for one or more basic blocks in this routine.

Returns

Pointer to name of the symbol at the entry node of the routine if it exists, `NULL` otherwise.

The documentation for this class was generated from the following file:

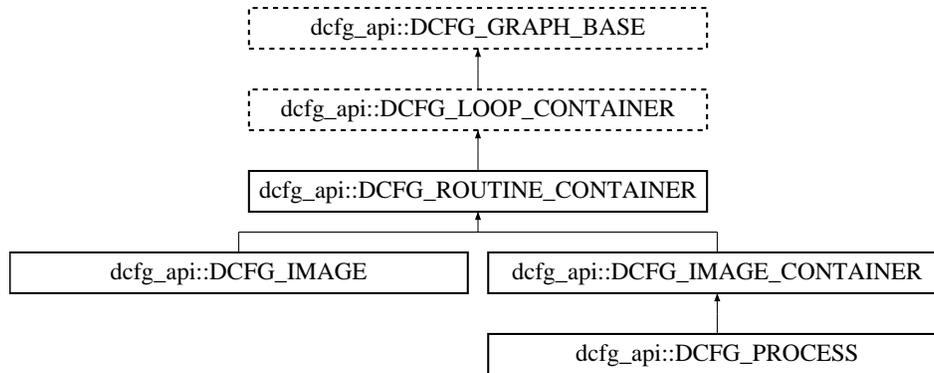
- `dcfg_api.H`

4.15 `dcfg_api::DCFG_ROUTINE_CONTAINER` Class Reference

Common interface to any structure containing routines, i.e., images and processes.

```
#include <dcfg_api.H>
```

Inheritance diagram for `dcfg_api::DCFG_ROUTINE_CONTAINER`:



Public Member Functions

- virtual UIN32 [get_routine_ids](#) (DCFG_ID_CONTAINER &node_ids) const =0
Get the set of routine IDs.
- virtual [DCFG_ROUTINE_CPTR](#) [get_routine_info](#) (DCFG_ID routine_id) const =0
Get access to data for a routine.

4.15.1 Detailed Description

Common interface to any structure containing routines, i.e., images and processes.

4.15.2 Member Function Documentation

4.15.2.1 virtual UIN32 `dcfg_api::DCFG_ROUTINE_CONTAINER::get_routine_ids (DCFG_ID_CONTAINER & node_ids) const`
[pure virtual]

Get the set of routine IDs.

Get IDs of all routines in the structure. A routine ID is the same as the ID of the basic block at its entry node.

Returns

Number of IDs that were added to `node_ids`.

Parameters

out	<i>node_ids</i>	Container to which IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER : DCFG_ID_VECTOR , DCFG_ID_SET , etc.
-----	-----------------	--

4.15.2.2 virtual [DCFG_ROUTINE_CPTR](#) `dcfg_api::DCFG_ROUTINE_CONTAINER::get_routine_info (DCFG_ID routine_id) const`
[pure virtual]

Get access to data for a routine.

Returns

Pointer to interface object for specified routine or `NULL` if `routine_id` is invalid.

Parameters

<code>in</code>	<code>routine_id</code>	ID of desired routine.
-----------------	-------------------------	------------------------

The documentation for this class was generated from the following file:

- `dcfg_api.H`

4.16 dcfg_trace_api::DCFG_TRACE_READER Class Reference

Interface to all data in a DCFG edge trace.

```
#include <dcfg_trace_api.H>
```

Public Member Functions

- virtual bool `open` (const std::string filename, UINT32 tid, std::string &errMsg)=0
Open a file for reading from the given thread.
- virtual bool `get_edge_ids` (dcfg_api::DCFG_ID_CONTAINER &edge_ids, bool &done, std::string &errMsg)=0
Read a chunk of edge IDs.

Static Public Member Functions

- static `DCFG_TRACE_READER * new_reader` (dcfg_api::DCFG_ID process_id)
Create a new DCFG edge-trace reader.

4.16.1 Detailed Description

Interface to all data in a DCFG edge trace.

This is an interface; use `DCFG_TRACE::new_trace()` to create an object that implements the interface.

4.16.2 Member Function Documentation

4.16.2.1 virtual bool `dcfg_trace_api::DCFG_TRACE_READER::get_edge_ids` (`dcfg_api::DCFG_ID_CONTAINER & edge_ids`, `bool & done`, `std::string & errMsg`) [pure virtual]

Read a chunk of edge IDs.

They will be added to `edge_ids` in the order in which they were recorded. This method will not typically read all the values at once. Call it repeatedly until `done` is set to `true`.

Returns

`true` on success, `false` otherwise (and sets `errMsg`).

Parameters

out	<i>edge_ids</i>	Container to which edge IDs are added. Previous contents of the container are <i>not</i> emptied by this call, so it should be emptied by the programmer before the call if desired. The programmer can use any implementation of DCFG_ID_CONTAINER that maintains insertion order: DCFG_ID_VECTOR, etc.
out	<i>done</i>	Set to <code>true</code> when end of sequence has been reached, <code>false</code> if there are more to read.
out	<i>errMsg</i>	Contains error message upon failure.

4.16.2.2 `static DCFG_TRACE_READER* dcfg_trace_api::DCFG_TRACE_READER::new_reader (dcfg_api::DCFG.ID process_id) [static]`

Create a new DCFG edge-trace reader.

This is a factory method to create a new object that implements the [DCFG_TRACE_READER](#) interface. A reader can access only one process. Create multiple readers to read multiple processes.

Returns

Pointer to new object. It can be freed with `delete`.

Parameters

in	<i>process_id</i>	ID of process to read. This can be determined from the DCFG data object corresponding to the trace.
----	-------------------	---

4.16.2.3 `virtual bool dcfg_trace_api::DCFG_TRACE_READER::open (const std::string filename, UINT32 tid, std::string & errMsg) [pure virtual]`

Open a file for reading from the given thread.

Returns

`true` on success, `false` otherwise (and sets `errMsg`).

Parameters

in	<i>filename</i>	Name of file to open, which must follow the DCFG-Trace JSON format.
in	<i>tid</i>	ID number of thread to read.
out	<i>errMsg</i>	Contains error message upon failure.

The documentation for this class was generated from the following file:

- `dcfg_trace_api.H`

Index

- activate
 - dcfg_pin_api::DCFG_PIN_MANAGER, 36
- add_id
 - dcfg_api::DCFG_ID_CONTAINER, 25
 - dcfg_api::DCFG_ID_SET, 26
 - dcfg_api::DCFG_ID_VECTOR, 27
- dcfg_api::DCFG_BASIC_BLOCK, 7
- dcfg_api::DCFG_DATA, 11
 - get_process_ids, 12
 - get_process_info, 12
 - new_dcfg, 12
 - read, 12, 13
 - write, 13
- dcfg_api::DCFG_EDGE, 14
 - get_edge_id, 15
 - get_edge_type, 15
 - get_exec_count, 16
 - get_source_node_id, 16
 - get_target_node_id, 16
 - is_any_branch_type, 16
 - is_any_bypass_type, 17
 - is_any_call_type, 17
 - is_any_return_type, 17
 - is_branch_edge_type, 17
 - is_call_edge_type, 18
 - is_context_edge_type, 18
 - is_entry_edge_type, 19
 - is_exit_edge_type, 19
 - is_rep_edge_type, 20
 - is_return_edge_type, 21
 - is_unknown_edge_type, 21
- dcfg_api::DCFG_GRAPH_BASE, 22
- dcfg_api::DCFG_ID_SET, 25
 - add_id, 26
- dcfg_api::DCFG_ID_VECTOR, 26
 - add_id, 27
- dcfg_api::DCFG_IMAGE, 27
 - get_base_address, 28
 - get_filename, 28
 - get_image_id, 29
 - get_process_id, 29
 - get_size, 29
- dcfg_api::DCFG_LOOP, 31
 - get_back_edge_ids, 32
 - get_image_id, 33
 - get_iteration_count, 33
 - get_loop_id, 33
 - get_parent_loop_id, 33
 - get_process_id, 34
 - get_routine_id, 34
- dcfg_api::DCFG_PROCESS, 37
 - get_edge_id, 39
 - get_edge_info, 40
 - get_process_id, 41
 - is_end_node, 42
 - is_special_node, 42
 - is_start_node, 42
 - is_unknown_node, 42
- dcfg_api::DCFG_ROUTINE, 43
 - get_entry_count, 44
 - get_image_id, 45
 - get_process_id, 45
 - get_routine_id, 46
 - get_symbol_name, 46
- dcfg_enable_knob
 - dcfg_pin_api::DCFG_PIN_MANAGER, 36
- get_back_edge_ids
 - dcfg_api::DCFG_LOOP, 31
- get_base_address
 - dcfg_api::DCFG_IMAGE, 28
- get_basic_block_id
 - dcfg_api::DCFG_BASIC_BLOCK, 8
- get_basic_block_ids
 - dcfg_api::DCFG_GRAPH_BASE, 23
- get_basic_block_ids_by_addr
 - dcfg_api::DCFG_IMAGE, 28
 - dcfg_api::DCFG_PROCESS, 39
- get_basic_block_info
 - dcfg_api::DCFG_PROCESS, 39
- get_dcfg_data
 - dcfg_pin_api::DCFG_PIN_MANAGER, 36
- get_edge_id
 - dcfg_api::DCFG_EDGE, 15
 - dcfg_api::DCFG_PROCESS, 39
- get_edge_ids
 - dcfg_trace_api::DCFG_TRACE_READER, 48
- get_edge_info
 - dcfg_api::DCFG_PROCESS, 40

get_edge_type
 dcfg_api::DCFG_EDGE, 15

get_end_node_id
 dcfg_api::DCFG_PROCESS, 40

get_entry_count
 dcfg_api::DCFG_ROUTINE, 44

get_entry_count_for_thread
 dcfg_api::DCFG_ROUTINE, 44

get_entry_edge_ids
 dcfg_api::DCFG_LOOP, 32
 dcfg_api::DCFG_ROUTINE, 44

get_exec_count
 dcfg_api::DCFG_BASIC_BLOCK, 8
 dcfg_api::DCFG_EDGE, 16

get_exec_count_for_thread
 dcfg_api::DCFG_BASIC_BLOCK, 8
 dcfg_api::DCFG_EDGE, 16

get_exit_edge_ids
 dcfg_api::DCFG_LOOP, 32
 dcfg_api::DCFG_ROUTINE, 45

get_filename
 dcfg_api::DCFG_IMAGE, 28

get_first_instr_addr
 dcfg_api::DCFG_BASIC_BLOCK, 9

get_highest_thread_id
 dcfg_api::DCFG_PROCESS, 40

get_idom_node_id
 dcfg_api::DCFG_ROUTINE, 45

get_image_id
 dcfg_api::DCFG_BASIC_BLOCK, 9
 dcfg_api::DCFG_IMAGE, 29
 dcfg_api::DCFG_LOOP, 33
 dcfg_api::DCFG_ROUTINE, 45

get_image_ids
 dcfg_api::DCFG_IMAGE_CONTAINER, 30

get_image_info
 dcfg_api::DCFG_IMAGE_CONTAINER, 30

get_inbound_edge_ids
 dcfg_api::DCFG_GRAPH_BASE, 23

get_inner_loop_id
 dcfg_api::DCFG_BASIC_BLOCK, 9

get_instr_count
 dcfg_api::DCFG_GRAPH_BASE, 23

get_instr_count_for_thread
 dcfg_api::DCFG_GRAPH_BASE, 23

get_internal_edge_ids
 dcfg_api::DCFG_GRAPH_BASE, 24

get_iteration_count
 dcfg_api::DCFG_LOOP, 33

get_iteration_count_for_thread
 dcfg_api::DCFG_LOOP, 33

get_last_instr_addr
 dcfg_api::DCFG_BASIC_BLOCK, 9

get_loop_id
 dcfg_api::DCFG_LOOP, 33

get_loop_ids
 dcfg_api::DCFG_LOOP_CONTAINER, 35

get_loop_info
 dcfg_api::DCFG_LOOP_CONTAINER, 35

get_num_instrs
 dcfg_api::DCFG_BASIC_BLOCK, 9

get_outbound_edge_ids
 dcfg_api::DCFG_GRAPH_BASE, 24

get_parent_loop_id
 dcfg_api::DCFG_LOOP, 33

get_predecessor_node_ids
 dcfg_api::DCFG_PROCESS, 40

get_process_id
 dcfg_api::DCFG_BASIC_BLOCK, 9
 dcfg_api::DCFG_IMAGE, 29
 dcfg_api::DCFG_LOOP, 34
 dcfg_api::DCFG_PROCESS, 41
 dcfg_api::DCFG_ROUTINE, 45

get_process_ids
 dcfg_api::DCFG_DATA, 12

get_process_info
 dcfg_api::DCFG_DATA, 12

get_routine_id
 dcfg_api::DCFG_BASIC_BLOCK, 10
 dcfg_api::DCFG_LOOP, 34
 dcfg_api::DCFG_ROUTINE, 46

get_routine_ids
 dcfg_api::DCFG_ROUTINE_CONTAINER, 47

get_routine_info
 dcfg_api::DCFG_ROUTINE_CONTAINER, 47

get_size
 dcfg_api::DCFG_BASIC_BLOCK, 10
 dcfg_api::DCFG_IMAGE, 29

get_source_filename
 dcfg_api::DCFG_BASIC_BLOCK, 10

get_source_line_number
 dcfg_api::DCFG_BASIC_BLOCK, 10

get_source_node_id
 dcfg_api::DCFG_EDGE, 16

get_start_node_id
 dcfg_api::DCFG_PROCESS, 41

get_successor_node_ids
 dcfg_api::DCFG_PROCESS, 41

get_symbol_name
 dcfg_api::DCFG_BASIC_BLOCK, 10
 dcfg_api::DCFG_ROUTINE, 46

get_symbol_offset
 dcfg_api::DCFG_BASIC_BLOCK, 10

get_target_node_id
 dcfg_api::DCFG_EDGE, 16

get_unknown_node_id
 dcfg_api::DCFG_PROCESS, 41

is_any_branch_type
 dcfg_api::DCFG_EDGE, 16

is_any_bypass_type
 dcfg_api::DCFG_EDGE, 17

is_any_call_type
 dcfg_api::DCFG_EDGE, 17

is_any_inter_routine_type
 dcfg_api::DCFG_EDGE, 17

is_any_return_type
 dcfg_api::DCFG_EDGE, 17

is_branch_edge_type
 dcfg_api::DCFG_EDGE, 17

is_call_bypass_edge_type
 dcfg_api::DCFG_EDGE, 17

is_call_edge_type
 dcfg_api::DCFG_EDGE, 18

is_conditional_branch_edge_type
 dcfg_api::DCFG_EDGE, 18

is_context_bypass_edge_type
 dcfg_api::DCFG_EDGE, 18

is_context_edge_type
 dcfg_api::DCFG_EDGE, 18

is_context_return_edge_type
 dcfg_api::DCFG_EDGE, 18

is_direct_branch_edge_type
 dcfg_api::DCFG_EDGE, 18

is_direct_call_edge_type
 dcfg_api::DCFG_EDGE, 19

is_direct_conditional_branch_edge_type
 dcfg_api::DCFG_EDGE, 19

is_direct_unconditional_branch_edge_type
 dcfg_api::DCFG_EDGE, 19

is_end_node
 dcfg_api::DCFG_PROCESS, 42

is_entry_edge_type
 dcfg_api::DCFG_EDGE, 19

is_excluded_bypass_edge_type
 dcfg_api::DCFG_EDGE, 19

is_exit_edge_type
 dcfg_api::DCFG_EDGE, 19

is_fall_thru_edge_type
 dcfg_api::DCFG_EDGE, 20

is_indirect_branch_edge_type
 dcfg_api::DCFG_EDGE, 20

is_indirect_call_edge_type
 dcfg_api::DCFG_EDGE, 20

is_indirect_conditional_branch_edge_type
 dcfg_api::DCFG_EDGE, 20

is_indirect_unconditional_branch_edge_type
 dcfg_api::DCFG_EDGE, 20

is_rep_edge_type
 dcfg_api::DCFG_EDGE, 20

is_return_edge_type
 dcfg_api::DCFG_EDGE, 21

is_special_node
 dcfg_api::DCFG_PROCESS, 42

is_start_node
 dcfg_api::DCFG_PROCESS, 42

is_sys_call_bypass_edge_type
 dcfg_api::DCFG_EDGE, 21

is_sys_call_edge_type
 dcfg_api::DCFG_EDGE, 21

is_sys_return_edge_type
 dcfg_api::DCFG_EDGE, 21

is_unconditional_branch_edge_type
 dcfg_api::DCFG_EDGE, 21

is_unknown_edge_type
 dcfg_api::DCFG_EDGE, 21

is_unknown_node
 dcfg_api::DCFG_PROCESS, 42

new_dcfg
 dcfg_api::DCFG_DATA, 12

new_manager
 dcfg_pin_api::DCFG_PIN_MANAGER, 37

new_reader
 dcfg_trace_api::DCFG_TRACE_READER, 49

open
 dcfg_trace_api::DCFG_TRACE_READER, 49

read
 dcfg_api::DCFG_DATA, 12, 13

set_cfg_collection
 dcfg_pin_api::DCFG_PIN_MANAGER, 37

write
 dcfg_api::DCFG_DATA, 13