



# Coverity Extend SDK 2020.12 Checker Development Guide

Coverity Extend Software Development Kit (Coverity Extend SDK) is part of Coverity Analysis.  
Copyright 2020 Synopsys, Inc. All rights reserved worldwide.

---

## Table of Contents

1. Coverity Extend SDK Usage .....	1
1.1. Overview .....	3
1.1.1. Introduction .....	3
1.1.2. Coverity Extend SDK directory structure .....	3
1.1.3. Compiling Coverity Extend SDK checkers .....	4
1.2. Creating your first checker: Hello .....	5
1.2.1. The Hello checker source code .....	5
1.2.2. Compiling the Hello checker .....	6
1.2.3. Running the Hello checker .....	7
1.2.4. Committing the issues to Coverity Connect .....	11
1.2.5. Dissecting the hello checker .....	12
1.2.6. Creating a makefile for convenience .....	13
1.3. The Abstract Syntax Tree .....	15
1.3.1. What is the AST? .....	15
1.3.2. Examining nodes in the AST with print_tree .....	16
1.3.3. Patterns .....	18
1.3.4. Accessors .....	20
1.4. State machine paradigm .....	22
1.4.1. Simple checker vs. checker with store .....	22
1.4.2. Abstract interpretation .....	22
1.4.3. Visit order .....	23
1.4.4. Manipulating the store .....	24
1.4.5. Example: tracking the sign of expressions .....	24
1.5. Output .....	27
1.5.1. OUTPUT_ERROR .....	27
1.5.2. ADD_EVENT .....	27
1.5.3. COMMIT_ERROR .....	28
1.5.4. ADD_INPUTFILE_ONLY_EVENT .....	28
1.5.5. COMMIT_INPUTFILE_ONLY_ERROR .....	28
1.5.6. ADD_INPUTFILE_EVENT .....	28
1.5.7. COMMIT_INPUTFILE_ERROR .....	28
1.5.8. Example: sign2 .....	29
1.6. Conditionals .....	30
1.6.1. ANALYZE_CONDITION .....	30
1.6.2. MATCH_COND .....	30
1.6.3. force_backtrack .....	31
1.6.4. Example: sign3 .....	31
1.6.5. Abstract comparison .....	31
1.6.6. Comparison evaluation .....	32
1.7. Paths .....	34
1.7.1. Many paths per function .....	34
1.7.2. False path pruning (FPP) .....	34
1.7.3. Two-pass checking .....	35
1.7.4. Termination .....	35
1.8. Examining the class hierarchy .....	36
1.8.1. Introduction .....	36

1.8.2. Mapping from variables to their class/type .....	36
1.8.3. Tree structure of types .....	37
1.8.4. Classes .....	37
1.8.5. type_t iterators .....	38
1.8.6. When are type_t objects resident in memory? .....	38
1.8.7. Example: print type information .....	39
1.8.8. Example: switch default .....	39
1.9. Reporting events and defects on input files .....	40
1.9.1. Additional steps for building Coverity Extend SDK checkers for Android applications .....	40
1.9.2. Input file class extend_inputfile_t .....	40
1.9.3. Input file macros .....	40
1.9.4. Input file checker examples .....	42
1.10. Troubleshooting .....	43
1.10.1. A Coverity Extend SDK checker aborts execution with a Tree used with no match error. ....	43
2. Coverity Extend SDK Reference .....	46
2.1. Introduction .....	48
2.2. Handler functions .....	49
2.2.1. Handler function overview .....	49
2.3. Patterns .....	57
2.3.1. Patterns for C# and Java checkers .....	58
2.3.2. Functions common to all patterns .....	58
2.3.3. ASTNodePattern superclass .....	59
2.3.4. ExpressionPattern superclass .....	59
2.3.5. TypePattern superclass .....	60
2.3.6. SymbolPattern superclass .....	60
2.3.7. Predefined pattern objects .....	60
2.3.8. Expression patterns .....	60
2.3.9. Statement patterns .....	67
2.3.10. Other patterns .....	68
2.4. Accessors .....	71
2.4.1. Additional AST query functions .....	71
2.4.2. Queries on the current function .....	71
2.4.3. Queries on the current file .....	72
2.4.4. Queries on trees .....	72
2.5. The store .....	73
2.5.1. Store overview .....	73
2.5.2. void SET_STATE(tree t, int v) .....	73
2.5.3. void CLEAR_STATE(tree t) .....	74
2.5.4. bool GET_STATE(tree t, int &v) .....	74
2.5.5. bool MATCH_STATE(tree t, int v) .....	74
2.5.6. bool COPY_STATE(tree dst, tree src) .....	74
2.5.7. FOREACH_IN_STORE(tree &t, int &v) { body } .....	74
2.5.8. bool ADD_EVENT(tree t, char const *tag, desc) .....	74
2.5.9. bool COMMIT_ERROR(tree t, char const *tag, desc) .....	74
2.5.10. ADD_INPUTFILE_EVENT .....	75
2.5.11. COMMIT_INPUTFILE_ERROR .....	75

2.6. Adding events .....	76
2.7. Types .....	77
2.7.1. Introduction .....	77
2.7.2. type_t .....	77
2.7.3. any_type_t .....	78
2.7.4. scalar_type_t .....	78
2.7.5. pointer_type_t .....	78
2.7.6. array_type_t .....	79
2.7.7. cv_wrapper_type_t .....	79
2.7.8. function_type_t .....	79
2.7.9. scoped_type_t .....	79
2.7.10. scope_t .....	79
2.7.11. defined_type_t .....	80
2.7.12. typedef_type_t .....	80
2.7.13. forward_declarable_type_t .....	80
2.7.14. enum_type_t .....	80
2.7.15. tag_t .....	80
2.7.16. union_type_t .....	80
2.7.17. field_t .....	81
2.7.18. class_type_t .....	81
2.7.19. parent_t .....	81
2.7.20. function_t .....	82
2.7.21. member_type_t .....	82
2.7.22. extend_inputfile_t .....	82
2.8. Reference information .....	83
2.8.1. Header files .....	83
2.8.2. Name mangling .....	83
3. Checker Examples .....	89
3.1. Checker source files .....	90
3.1.1. sign checker .....	90
3.1.2. sign2 checker .....	94
3.1.3. sign3 checker .....	98
3.1.4. print_types.cpp .....	107
3.1.5. switch_default.cpp .....	109
3.1.6. javascript_match_local.cpp .....	110
4. Coverity Runtime Library Development Guide .....	111
4.1. Overview .....	112
4.2. Directory Structure .....	113
4.3. Building the Runtime Library for Daemon and Linux .....	114
4.4. Testing the Runtime Library for Linux .....	115
4.5. Deploying a Runtime Library and Daemon for Linux .....	116
4.5.1. Dynamic Library Deployment .....	116
4.5.2. Static Library Deployment .....	116
4.5.3. Daemon Deployment .....	116
4.6. Building the Runtime Library and Daemon for Windows .....	118
4.7. Testing the Runtime Library for Windows .....	119
4.8. Deploying a Runtime Library and Daemon for Windows .....	120
4.9. Configuring the Runtime Library Build for Linux and Windows .....	121

4.9.1. Common Environment Variables .....	121
4.9.2. Linux-specific Environment Variables .....	122
4.9.3. Windows-specific Environment Variables .....	122
4.10. Instrumentation Predicate Language .....	124
4.10.1. Non-CIT compiler support .....	124
4.10.2. CIT predicate language extensions .....	130

---

# Part 1. Coverity Extend SDK Usage

## Table of Contents

1.1. Overview .....	3
1.1.1. Introduction .....	3
1.1.2. Coverity Extend SDK directory structure .....	3
1.1.3. Compiling Coverity Extend SDK checkers .....	4
1.2. Creating your first checker: Hello .....	5
1.2.1. The Hello checker source code .....	5
1.2.2. Compiling the Hello checker .....	6
1.2.3. Running the Hello checker .....	7
1.2.4. Committing the issues to Coverity Connect .....	11
1.2.5. Dissecting the hello checker .....	12
1.2.6. Creating a makefile for convenience .....	13
1.3. The Abstract Syntax Tree .....	15
1.3.1. What is the AST? .....	15
1.3.2. Examining nodes in the AST with <code>print_tree</code> .....	16
1.3.3. Patterns .....	18
1.3.4. Accessors .....	20
1.4. State machine paradigm .....	22
1.4.1. Simple checker vs. checker with store .....	22
1.4.2. Abstract interpretation .....	22
1.4.3. Visit order .....	23
1.4.4. Manipulating the store .....	24
1.4.5. Example: tracking the sign of expressions .....	24
1.5. Output .....	27
1.5.1. <code>OUTPUT_ERROR</code> .....	27
1.5.2. <code>ADD_EVENT</code> .....	27
1.5.3. <code>COMMIT_ERROR</code> .....	28
1.5.4. <code>ADD_INPUTFILE_ONLY_EVENT</code> .....	28
1.5.5. <code>COMMIT_INPUTFILE_ONLY_ERROR</code> .....	28
1.5.6. <code>ADD_INPUTFILE_EVENT</code> .....	28
1.5.7. <code>COMMIT_INPUTFILE_ERROR</code> .....	28
1.5.8. Example: <code>sign2</code> .....	29
1.6. Conditionals .....	30
1.6.1. <code>ANALYZE_CONDITION</code> .....	30
1.6.2. <code>MATCH_COND</code> .....	30
1.6.3. <code>force_backtrack</code> .....	31
1.6.4. Example: <code>sign3</code> .....	31
1.6.5. Abstract comparison .....	31
1.6.6. Comparison evaluation .....	32
1.7. Paths .....	34
1.7.1. Many paths per function .....	34
1.7.2. False path pruning (FPP) .....	34
1.7.3. Two-pass checking .....	35

---

---

1.7.4. Termination .....	35
1.8. Examining the class hierarchy .....	36
1.8.1. Introduction .....	36
1.8.2. Mapping from variables to their class/type .....	36
1.8.3. Tree structure of types .....	37
1.8.4. Classes .....	37
1.8.5. type_t iterators .....	38
1.8.6. When are type_t objects resident in memory? .....	38
1.8.7. Example: print type information .....	39
1.8.8. Example: switch default .....	39
1.9. Reporting events and defects on input files .....	40
1.9.1. Additional steps for building Coverity Extend SDK checkers for Android applications .....	40
1.9.2. Input file class extend_inputfile_t .....	40
1.9.3. Input file macros .....	40
1.9.4. Input file checker examples .....	42
1.10. Troubleshooting .....	43
1.10.1. A Coverity Extend SDK checker aborts execution with a Tree used with no match error. ....	43

---

---

# Chapter 1.1. Overview

## Table of Contents

1.1.1. Introduction .....	3
1.1.2. Coverity Extend SDK directory structure .....	3
1.1.3. Compiling Coverity Extend SDK checkers .....	4

### 1.1.1. Introduction

Coverity Extend SDK is a framework for writing program analyzers (that is, *checkers*) in C++ that support analyses of C/C++, Java, and C# applications. Much of this framework is the same as that used by the checkers in Coverity Analysis. The framework provides the following services:

- Basic front-end features: parsing, type checking and elaboration, abstract syntax construction, template instantiation, and linking across translation units.
- Facilities to inspect abstract syntax, using pattern matching.
- Mechanisms to traverse paths in the abstract syntax in execution order, prune false paths, and merge similar states to ensure termination in loops.
- Flexible checker state management for derivation of flow-sensitive properties.
- Output routines that work with the false path pruning (FPP) mechanism to ensure that defects are only reported in feasible paths.

What you must write is a description of a *state machine*, also known as an abstract interpreter. This description specifies how the state transitions occur and which states constitute errors. The Coverity Extend SDK framework then runs this state machine over each function in the code that is undergoing analysis, collects the defect (issue) reports (those produced in error states), and allows you to commit the reports to Coverity Connect, where developers can learn about and triage the issues.

 **Note**

CodeXM is a language specifically designed for writing new checkers. If you have not already invested in the Extend SDK, we strongly recommend you use CodeXM rather than the mechanisms described in this manual. See *Learning to Write CodeXM Checkers* [🔗](#).

### 1.1.2. Coverity Extend SDK directory structure

The installer puts Coverity Extend SDK resources into the `<install_dir>/sdk` directory. It contains the following subdirectories:

- `compiler` — Coverity Extend SDK compiler for checker source code.
- `doc` — Coverity Extend SDK documentation in HTML and PDF formats.
- `headers` — Coverity Analysis specific C++ header files that are needed to compile a Coverity Extend SDK checker. See Section 2.8.1, “Header files” for a description of each header.

- `libs` — Binary files that are needed to link to a Coverity Extend SDK checker.
- `samples` — Sample checkers that illustrate different Coverity Extend SDK features.

### 1.1.3. Compiling Coverity Extend SDK checkers

The Coverity Extend SDK provides a few tools for compiling checkers:

- On Unix and Windows: The `build-checker` or, on Windows, `build-checker.bat` command. You will use this command to build the Hello checker sample in Chapter 1.2, *Creating your first checker: Hello*.

```
<install_dir>/sdk/build-checker <checker-name>
```

This command compiles a Coverity Extend SDK checker. It looks in the current directory for `<checker_name>.c` or `<checker_name>.cpp`, compiles it, and places the executable in the current directory. You can then run it as you would run `cov-analyze`. (For information about `cov-analyze`, see the *Coverity Command Reference*.)

The `build-checker` command needs to be able to find the Coverity Extend SDK installation directory. If you copy the command from `<install_dir>/sdk` to another location, you must set environment variable `PREVENT_ROOT` to the root directory of the Coverity Extend SDK installation. For example:

For example, on Unix:

```
export PREVENT_ROOT=<install_dir>/sdk
```

For example, on Windows:

```
set PREVENT_ROOT=<install_dir>/sdk
```

- Makefile in `<install_dir>/sdk/samples` compiles all the sample checkers located in `<install_dir>/sdk/samples`. You need to run `Makefile` from this directory. For example:

```
> cd <install_dir>/sdk/samples
> make
```

#### ⚠ Caution

When compiling on a Windows system, you must use the `make` command provided by GNU, and your system must be configured to use a POSIX shell. The Cygwin environment is one possible solution.

The console output consists of a number of compile and link command lines. After successful completion, the console will print a message similar to the following:

```
SUCCESS! Your checker has been compiled to ./whileloopassign
```

- On Unix only: Each sample checker subdirectory (for example, `<install_dir>/sdk/samples/hello`) includes a `Makefile` that is designed to compile just that checker. See Section 1.2.6, “Creating a makefile for convenience”.

---

## Chapter 1.2. Creating your first checker: Hello

### Table of Contents

1.2.1. The Hello checker source code .....	5
1.2.2. Compiling the Hello checker .....	6
1.2.3. Running the Hello checker .....	7
1.2.4. Committing the issues to Coverity Connect .....	11
1.2.5. Dissecting the hello checker .....	12
1.2.6. Creating a makefile for convenience .....	13

In this section, you will build and run the Hello checker, then commit issues it finds to a stream in Coverity Connect. In addition to using Coverity Connect and the Coverity Extend SDK, you will also need to run Coverity Analysis commands on sample code and use Coverity Analysis to run your Hello checker.

#### Requirements

- You must have an installation of Coverity with a valid license. The Coverity Analysis installer can install Coverity Analysis along with the Coverity Extend SDK component and other Coverity products.

For an introduction to Coverity products, see *Coverity Analysis 2020.12 User and Administrator Guide* [↗](#).

- You must have access to an installation of Coverity Connect. As a best practice, you should use a test instance of Coverity Connect, rather than using a production instance. At minimum, you (or a Coverity Connect administrator) should set up a separate project in Coverity Connect with a test stream into which you can commit issues found by the Hello checker. You will need the stream name and a Coverity Connect role that gives you permission to commit issues to that stream and to view issues in that stream.

For Coverity Connect installation and configuration details, see *Coverity 2020.12 Installation and Deployment Guide* [↗](#) and *Coverity Platform 2020.12 User and Administrator Guide* [↗](#).

### 1.2.1. The Hello checker source code

In this section, you create a simple Coverity Extend SDK checker (`hello.cpp`) designed to print every abstract syntax tree that is passed into the `ANALYZE_TREE` function. In subsequent sections, you will compile and run this checker, then commit the issues that it finds in a code sample to Coverity Connect.

#### To create the Hello checker:

1. Type or copy the following source code into a text editor:

```
/*  
(c) 2017, Synopsys, Inc. All rights reserved worldwide.
```

```
The information contained in this file is the proprietary and confidential
information of Synopsys, Inc. and its licensors, and is supplied subject to,
and may be used only by Synopsys customers in accordance with the terms and
conditions of a previously executed license agreement between Synopsys and that
customer.
*/

// hello.c
// trivial Extend checker

#include "extend-lang.hpp"      // Extend API

START_EXTEND_CHECKER( hello, simple );

ANALYZE_TREE()
{
    cout << "ANALYZE_TREE: " << CURRENT_TREE << endl;
    OUTPUT_ERROR("ANALYZE_TREE: " << CURRENT_TREE);

    ReturnPat ret;
    if( MATCH(ret) ) print_tree(CURRENT_TREE);
}

END_EXTEND_CHECKER();

MAKE_MAIN( hello )

// EOF
```

The source code and makefile for this checker are located in the `<install_dir>/sdk/samples/hello` directory.

Unlike the `print_tree` checker (see Section 1.3.2, “Examining nodes in the AST with `print_tree`”), this checker restricts the level of information that is returned by using an `if` statement before calling `print_tree`.

2. Save this file as `hello.cpp` in a directory that is *outside* of the Coverity Extend SDK installation directory.

For example: `<HELLO>` so that your checker source file is now called `HELLO/hello.cpp`

### **Note**

Saving the file inside of the installation directory can make the upgrade process for Coverity Extend SDK more difficult.

## 1.2.2. Compiling the Hello checker

In this section, you compile the Hello checker that you built in Section 1.2.1, “The Hello checker source code”.

### To compile the Hello checker:

1. Go to your <HELLO> directory, and invoke `build-checker`:

```
> cd <HELLO>
> <install_dir>/sdk/build-checker hello
```

Note that the argument is `hello`, not `hello.cpp`.

<install\_dir> is the Coverity Analysis root directory.

After printing the compilation and linking command line, this build command prints the following:

```
SUCCESS! Your checker has been compiled to ./hello
```

If an error occurs, set your `PATH`. See `build-checker` in Section 1.1.3, “Compiling Coverity Extend SDK checkers”.

#### Note

On Unix, you can run the makefile for the sample checker instead of running `build-checker`. This file is located in the `<install_dir>/sdk/samples/hello` directory.

On Windows, if you see the following error when compiling your checker, you must either restart your console as an administrator or make a copy of the `samples` directory:

```
C:/Program Files/Coverity/Coverity Static Analysis/sdk/compiler/bin/ld.exe:
cannot open output file hello.exe: Permission denied
collect2: ld returned 1 exit status
ERROR: Checker "hello" did not successfully compile.
```

2. Locate the following output in your <HELLO> directory:

- `hello` (on Unix)
- `hello.exe` (on Windows)

This output is the checker program, which supports a command-line interface similar to `cov-analyze`, except that it can only run one checker. (For information about `cov-analyze`, see the *Coverity 2020.12 Command Reference* [🔗](#).)

## 1.2.3. Running the Hello checker

In this section, you run the Hello checker that you compiled in Section 1.2.2, “Compiling the Hello checker”. See Requirements before attempting to complete the steps in this section.

### To run the Hello checker:

1. Create some sample input for the checker.

You can save the following code as a file called `HELLO/test1/hello.test.c`:

```
        /*
(c) 2017, Synopsys, Inc. All rights reserved worldwide.
The information contained in this file is the proprietary and confidential
information of Synopsys, Inc. and its licensors, and is supplied subject to,
and may be used only by Synopsys customers in accordance with the terms and
conditions of a previously executed license agreement between Synopsys and that
customer.
*/

// test1/hello.test.c
// test input for 'hello' checker

int foo()
{
    int x = 1;
    x += 5;
    return x;
}

// EOF
```

2. Use Coverity to configure a compiler.

To configure gcc and g++ compilers with Coverity Analysis:

```
> cd <install_dir>/bin
> ./cov-configure --gcc
```

To configure the Microsoft C/C++ compiler `cl.exe` with Coverity Analysis:

```
> cd <install_dir>\bin
> cov-configure --msvc
```

 **Note**

The remaining steps in this section assume that you are using the Unix-based gcc compiler. If you are using a different compiler, configure it instead, and adjust the command lines to use the appropriate command-line syntax for that compiler and operating system. For guidance with the configuration of such compilers, see the *Coverity Analysis 2020.12 User and Administrator Guide*. For more complete information about compiler configuration, you can also refer to the *Coverity Analysis 2020.12 User and Administrator Guide* and the *Coverity 2020.12 Command Reference* documentation on the `cov-configure`. All of this documentation is available from `<install_dir>/docs/<en|ja>/index.html` (where `en` contains the English-language documentation set for Coverity Analysis, and `ja` contains the Japanese-language documentation).

3. Use `cov-build` to intercept calls to the compiler and save its abstract syntax in the intermediate directory.

On Unix:

```
> cd <HELLO>
> <install_dir>/bin/cov-build --dir int_dir gcc -c test1/hello.test.c
```

On Windows:

```
> <install_dir>\bin\cov-build --dir int_dir cl test1\hello.test.c
```

Upon successful completion, this command prints the following output:

```
[...]
1 C/C++ compilation units (100%) are ready for analysis
The cov-build utility completed successfully.
```

This command creates a `c/emit` subdirectory (an emit directory) in your intermediate directory that contains the `cov-build` output: `<HELLO>/int_dir/c/emit`.

For information about the `cov-build` command, see the [Coverity 2020.12 Command Reference](#).

4. Use your Hello checker (`hello`) to analyze this intermediate directory:
  - a. Copy the `hello` checker program into the Coverity Analysis `bin` directory.

For example, on Unix:

```
> cp hello <install_dir>/bin
```

On Windows:

```
> copy hello.exe <install_dir>\bin
```

- b. Run `hello` from your `<HELLO>` directory:

```
> cd <HELLO>
> <install_dir>/bin/hello --dir int_dir --force
```

 **Note**

The options to the `hello` checker are the same as those for the `cov-analyze` command.

The output looks something like the following:

```
                                Looking for translation units
|0-----25-----50-----75-----100|
|*****|
[STATUS] Computing links for 1 translation unit
|0-----25-----50-----75-----100|
|*****|
[STATUS] Computing virtual overrides
```

## Creating your first checker: Hello

```
|0-----25-----50-----75-----100|
*****
[STATUS] Computing callgraph
|0-----25-----50-----75-----100|
*****
[STATUS] Topologically sorting 1 function
|0-----25-----50-----75-----100|
*****
[STATUS] Computing node costs
|0-----25-----50-----75-----100|
*****
[STATUS] Starting analysis run
ANALYZE_TREE: "{...}"
ANALYZE_TREE: "int x = 1"
ANALYZE_TREE: "x"
ANALYZE_TREE: "1"
ANALYZE_TREE: "x = 1"
ANALYZE_TREE: "x = 1;"
ANALYZE_TREE: "int x = 1;"
ANALYZE_TREE: "x"
ANALYZE_TREE: "x"
ANALYZE_TREE: "5"
ANALYZE_TREE: "x + 5"
ANALYZE_TREE: "x = x + 5"
ANALYZE_TREE: "x += 5;"
ANALYZE_TREE: "x"
ANALYZE_TREE: "return x;"
tree = S_return:
  loc = <file ID 0>:5:3-<file ID 0>:5:11
  expr = E_variable:
    type = int
    deepID = 2147483646
    var = x, type = int, dflags = {}

  isImplicit = 0
ANALYZE_TREE: "<destruction for x>"
|0-----25-----50-----75-----100|
*****
Analysis summary report:
-----
Files analyzed                : 1
Total LoC input to cov-analyze : 2990
Functions analyzed            : 1
Paths analyzed                 : 1
Time taken by Coverity analysis : 00:00:00
Defect occurrences found       : 15 hello
```

Aside from the usual `cov-analyze` text, the output consists of one line for each call to the `ANALYZE_TREE` function, showing the abstract syntax tree that was passed to it.

The checker also creates `<HELLO>/int_dir/<programming_language>/output/hello.errors.xml`, containing that output in a format that can be committed to Coverity Connect.

### 1.2.3.1. Running the Hello checker from another directory

You can run a Coverity Extend SDK checker from a directory other than `<install_dir>/bin` (as shown in Section 1.2.3, “Running the Hello checker”). You might do so if Coverity is installed to a read-only directory.

The Hello checker requires the following options:

- The installation directory: Specified by the `--prevent-root` option.
- The intermediate directory: Specified by the `--dir` option.

For example:

```
> cd <HELLO>
> <install_dir>/sdk/build-checker hello
> <install_dir>/bin/cov-build --dir int_dir_2 gcc -c test1/hello.test.c
> ./hello --dir int_dir_2 --prevent-root=<install_dir>
```

### 1.2.4. Committing the issues to Coverity Connect

Just as you can commit the output of the `cov-analyze` command to Coverity Connect, you can also commit the issues that the Hello checker finds.

**To commit issues found by Hello to Coverity Connect:**

1. Prepare Coverity Connect to receive the issues found by the Hello checker:
  - a. Start Coverity Connect.

The startup command is located in the Coverity Connect `/bin` directory:

```
> cd <install_dir_cc>/bin
> ./cov-start-im
```

- b. Log into Coverity Connect.
- c. Create a project that is configured with a Coverity Analysis stream for the C/C++ programming language.

For example: `hello_stream` in the project `extend_examples`

 **Note**

If you need help with any of these steps, contact your Coverity Connect administrator.

2. Use Coverity to commit (push) the issues to Coverity Connect:

```
> <install_dir>/bin/cov-commit-defects \
  --host <server_hostname> \
  --port <port_number> \
  --stream hello_stream \
```

```
--user admin --dir int_dir
```

This command produces output similar to the following on the console:

```
Connecting to server sduke-t61p:9090
2012-08-06 21:54:57 UTC - Committing 4 file descriptions...
|0-----25-----50-----75-----100|
*****
2012-08-06 21:54:57 UTC - Committing 4 source files...
|0-----25-----50-----75-----100|
*****
2012-08-06 21:54:56 UTC - Calculating 4 cross-references...
|0-----25-----50-----75-----100|
*****
2012-08-06 21:54:57 UTC - Committing 4 cross-references...
|0-----25-----50-----75-----100|
*****
2012-08-06 21:54:58 UTC - Committing 0 functions...
2012-08-06 21:54:58 UTC - Committing 15 defect occurrences...
|0-----25-----50-----75-----100|
*****
2012-08-06 21:54:59 UTC - Committing 3 output files...
|0-----25-----50-----75-----100|
*****
New snapshot ID 10004 added.
Elapsed time: 00:00:04
```

## 1.2.5. Dissecting the hello checker

This section describes each line of `hello.cpp`.

The following include is for the header file that contains declarations for the classes, functions, and macros that comprise the Coverity Extend SDK API:

```
#include "extend-lang.hpp" // Coverity Extend SDK API
```

The headers are explained in detail in Section 2.8.1, “Header files”.

The next line declares that the name of the checker is `hello` and that it is a *simple* checker:

```
START_EXTEND_CHECKER( hello, simple );
```

Here, `simple` means that the checker is flow-insensitive: the checker is stateless, and the Abstract Syntax Tree (AST) nodes are not visited in any particular order. An AST is a tree-shaped data structure that represents the phrase structure of the concrete input syntax (for more information, see Chapter 1.3, *The Abstract Syntax Tree*). >Note that subsequent sections will introduce you to examples of flow-sensitive checkers. The Coverity Extend SDK macros such as `START_EXTEND_CHECKER` and `ANALYZE_TREE` are explained in Chapter 2.2, *Handler functions*.

The next line starts the principal function of a checker:

```
ANALYZE_TREE( )
```

The body of `ANALYZE_TREE` is called for every AST in the program that is undergoing analysis. The ASTs are passed one at a time, and as they arrive, each one is then the `CURRENT_TREE`.

The next line prints the current AST to standard output:

```
{
  cout << "ANALYZE_TREE: " << CURRENT_TREE << endl;
}
```

The next line prints the current AST as an issue report:

```
OUTPUT_ERROR("ANALYZE_TREE: " << CURRENT_TREE);
```

This report is also stored in an intermediate directory output file, `output/hello.errors.xml`. Note that, as discussed in Chapter 1.5, *Output*, not every call to `OUTPUT_ERROR` results in a user-visible issue report. For example, if the path along which the issue is found is later determined to be infeasible, the report is suppressed. Consequently, printing to standard output is useful as a debugging aid, but `OUTPUT_ERROR` should be used for the actual issue reports.

The Extend SDK defines *pattern types* which can be used to determine if the current AST node meets certain criteria. As an example, here we create a pattern which matches return statements by declaring a variable `ret` of type `ReturnPat`. The `MATCH` predicate determines whether the current AST node being processed by the checker matches the pattern. See Section 1.3.3, “Patterns ” for more details.

The `print_tree` function displays detailed information about the AST node; for more information, see Section 1.3.2, “Examining nodes in the AST with `print_tree`”

```
ReturnPat ret;
if( MATCH(ret) ) print_tree(CURRENT_TREE);
}
```

The next line signals the end of the checker:

```
END_EXTEND_CHECKER();
```

The final line creates `main()`, the entry point to the checker executable. The name of the checker is passed as an argument.

```
MAKE_MAIN( hello )
```

## 1.2.6. Creating a makefile for convenience

When creating your own checkers, you can adapt the Hello makefile (`<install_dir>/sdk/samples/hello/Makefile`) and the `checker.mk` file that it includes: `checker.mk` and `include.mk`.

Makefile for the Hello sample checker:

```
        # hello/Makefile
# Makefile for the 'hello' Extend checker

# name of the checker
include checker.mk
```

## Creating your first checker: Hello

---

```
# default target
all: $(CHECKER)

# rules shared by the example checkers
include ../include.mk

# EOF
```

The checker.mk file for the Hello sample checker:

```
        # hello/checker.mk
# set CHECKER to 'hello'

# This fragment is separated into its own file so that it can
# be used by the Makefile in this directory, as well of those
# in the test subdirectories.

CHECKER := hello

# EOF
```

---

## Chapter 1.3. The Abstract Syntax Tree

### Table of Contents

1.3.1. What is the AST? .....	15
1.3.2. Examining nodes in the AST with <code>print_tree</code> .....	16
1.3.3. Patterns .....	18
1.3.4. Accessors .....	20

There are three main tasks that a checker typically performs:

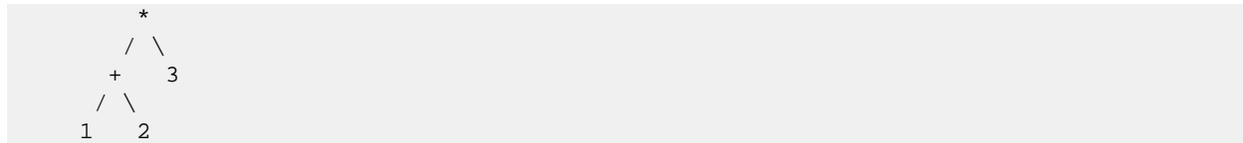
1. Inspect the AST to recognize syntax of importance.
2. Update the store to reflect the effect of that syntax (simple checkers do not do this).
3. Output errors when appropriate.

### 1.3.1. What is the AST?

An Abstract Syntax Tree (AST) is a tree-shaped data structure that represents the phrase of the concrete input syntax of the source code. For example, the input `1+2*3` has a corresponding AST that looks like this:



From this AST fragment, you can infer that the multiplication of 2 and 3 happens first (even though `*` occurs textually after the `+`), and the result is added to 1. If the input is instead `(1+2)*3`, then the AST is:



Note that the grouping parentheses has affected the AST by changing the order of operations. The parentheses are not explicitly present in the AST, since the tree structure is sufficient to represent their effect.

In Coverage Analysis, the AST is the output of the parser, and the input to the checker.

AST fragments can be grouped into several categories:

- Expressions, such as the examples shown previously.
- Statements, such as `x = 3;` or `while (true) { ... }.`
- Type identifiers, such as `int` or `class C { ... }.`
- Function definitions as a whole.

Although there are a few other categories, these are the main ones that most checkers use.

### 1.3.2. Examining nodes in the AST with `print_tree`

You can use the `print_tree` function to return detailed information about each node of the AST, as shown in the `print_tree` checker next:

```
// print_tree.c
// Coverity Extend SDK checker printing the AST tree at every node

#include "extend-lang.hpp"      // Coverity Extend SDK API

START_EXTEND_CHECKER( hello, simple );

ANALYZE_TREE()
{
    print_tree(CURRENT_TREE);
}

END_EXTEND_CHECKER();

MAKE_MAIN( hello )

// EOF
```

The source code and makefile for this checker are located in the `<install_dir>/sdk/samples/print_tree` directory.

Like the `hello` checker, this checker prints every abstract syntax tree that is passed into the `ANALYZE_TREE` function. However, the `print_tree` checker returns much more detailed information than the `hello` checker. You can use `print_tree(CURRENT_TREE)` to expose detailed information about an AST node. For example, the exact kind and many other details of each program element are exposed by the `element` structure, which you can see in the output below.

If you run this checker on the sample program at `<install_dir>/sdk/samples/print_tree/test1/print_tree_test.cpp` the output has the following format:

```
[STATUS] Reading call graph
[STATUS] Computing class hierarchy
|0-----25-----50-----75-----100|
*****
[STATUS] Computing call graph
[STATUS] Starting analysis run (analysis pass)
|0-----25-----50-----75-----100|
*****tree = S_compound:
loc = /nfs/foo2/extend/samples/print_tree/test1/print_tree.test.cpp:5
endLoc = /nfs/foo2/extend/samples/print_tree/test1/print_tree.test.cpp:9
stmts = {
    element = S_decl:
        loc = /nfs/foo2/extend/samples/print_tree/test1/print_tree.test.cpp:6
        decl = Declaration:
            var = "x"
            init = IN_expr:
```

```
expr = E_intLit:
  cached_hash = 0
  type = int
  i = 1
  original_expr = <null Expression>

destructionCode = <null Statement>

initCode = S_expr:
  loc = /nfs/foo2/extend/samples/print_tree/test1/print_tree.test.cpp:6
  expr = E_assign:
    cached_hash = 0
    type = int
    target = E_variable:
      cached_hash = 0
      type = int
      var = "x"

    op = 21
    src = E_intLit:
      cached_hash = 0
      type = int
      i = 1
      original_expr = <null Expression>

  destroyStmts = {
    tree = S_destroy
  }

element = S_expr:
  loc = /nfs/foo2/extend/samples/print_tree/test1/print_tree.test.cpp:7
  expr = E_assign:
    cached_hash = 0
    type = int
    target = E_variable:
      cached_hash = 0
      type = int
      var = "x"

    op = 9
    src = E_intLit:
      cached_hash = 0
      type = int
      i = 5
      original_expr = <null Expression>

element = S_return:
  loc = /nfs/foo2/extend/samples/print_tree/test1/print_tree.test.cpp:8
```

```

    expr = E_variable:
        cached_hash = 0
        type = int
        var = "x"

    isImplicit = 0
}

```

### 1.3.3. Patterns

The main method that checkers use to inspect the AST is through the use of *patterns*, which are fragments of syntax with pattern variables (*holes*) that can match arbitrary subtrees. The basic approach is to define a pattern with named holes, test to see if the pattern matches some input syntax, and then use the named holes to examine the parts of the patterns that match.

#### 1.3.3.1. Expression patterns

The expression patterns are, in most cases, constructed using C++ operator overloading. For an example, see `<install_dir>/sdk/samples/patterns/patterns.cpp`.

```

Expr a, b;
if (MATCH_TREE( a + b , CURRENT_TREE)) {
    cout << "  matched an addition; a=" << a << " , b=" << b << endl;
}

```

This checker fragment declares two pattern variables (*holes*) called `a` and `b`, with type `Expr`. The `Expr` pattern type matches any expression; there are other pattern types that are more restrictive. For convenience, there is a predefined pattern variable called `"_"` (underscore) that matches anything.

The previous code then constructs a pattern expression:

```
a + b
```

The `+` here is actually an overloaded operator that constructs a pattern. The pattern matches input ASTs that use the binary operator `+`.

It then uses `MATCH_TREE` to compare the constructed pattern to `CURRENT_TREE` (which is the AST passed to `ANALYZE_TREE`). This returns true when the input matches the pattern. For example, when given the following line of input syntax:

```
z = x + y;
```

the checker fragment prints out:

```
matched an addition; a="x", b="y"
```

because there is exactly one subexpression in the input file that matches the pattern.

When the line of code:

```
cout << "matched an addition; a=" << a << ", b=" << b << endl;
```

prints a pattern variable (such as `a`), it prints the AST fragment that the pattern variable matched. This AST fragment can also be obtained explicitly by calling the `Pattern::get_tree()` method.

Since it is common to use `MATCH_TREE` with `CURRENT_TREE`, the `MATCH` macro is available as a shortcut:

```
MATCH(<pattern>) is equivalent to MATCH_TREE(<pattern>, CURRENT_TREE)
```

To match a function call expression, create a pattern of type `CallSite`:

```
CallSite bar("bar"); // match call to bar()
if (MATCH( bar ))
    cout << "call to bar: " << CURRENT_TREE << endl;
```

When the `CallSite` pattern matches, you can inspect its arguments using the `nargs` and `get_arg` methods:

```
for (int i=0; i < bar.nargs(); i++) {
    cout << " arg " << (i+1) << ": " << bar.get_arg(i) << endl;
}
```

To match a call site with specific patterns for the arguments, simply pass the argument patterns to `CallSite::operator()`:

```
Const_int ci;
if (MATCH( bar(ci) ))
    cout << " single literal integer argument: " << ci.llval() << endl;
```

The previous fragment utilizes the `Const_int` pattern, which matches an expression that is an integer literal.

See Section 2.3.8, “Expression patterns” for more information.

### 1.3.3.1.1. Variable kind patterns

There are several pattern types that match uses of specific kinds of variables. Some of the more important are next.

**Table 1.3.1. Patterns for variables**

Parameter	Usage
<code>LocalVar</code>	use of local variable
<code>StaticVar</code>	use of a variable with static storage duration
<code>GlobalVar</code>	use of a variable with global scope

See Section 2.3.8.3, “Variable reference expression patterns” for more information.

### 1.3.3.1.2. Pattern combinators

Patterns can be combined using the following general-purpose combinators.

**Table 1.3.2. Pattern combinators**

<code>And(p1,p2)</code>	match if p1 and p2 match the given AST
<code>Or(p1,p2)</code>	match if p1 or p2 matches the given AST
<code>Within(p)</code>	match if some enclosing (parent or ancestor of given) AST matches p
<code>Contains(p)</code>	match if a subtree (descendant) of the given AST matches p

For example, given the declarations:

```
CallSite bar("bar");
Const_int ct;
LocalVar lv;
StaticVar sv;
GlobalVar gv;
```

the pattern:

```
bar(Or(ct,lv), Or(sv,gv))
```

matches any call to `bar` with two arguments, where the first argument is either an integer literal or a local variable, and the second is either a static variable or a global variable.

The following figure illustrates how `Within()` looks for parents/ancestors, and `Contains()` looks for descendants:

**Figure 1.3.1. Within() and Contains() combinators example**

Thus, the following statement

```
MATCH( Within( Contains (pat)))
```

matches the pattern from anywhere within the current function.

See Section 2.3.10.5, “Combinators (And, Or, etc.)” for more information.

## 1.3.4. Accessors

In addition to the patterns, there are a variety of functions that perform various queries on the AST. This section covers some of the more common functions. See Chapter 2.4, *Accessors* for more information.

Several accessors return information about the current function being analyzed.

**Table 1.3.3. Accessors for functions**

<code>current_function_get_name</code>	name of current function
<code>current_function_get_return_type</code>	return type of current function

## The Abstract Syntax Tree

---

<code>current_file_get_name</code>	file containing current function
<code>print_tree</code>	Returns detailed information on the node, including the type of pattern

Others return information about a specific AST.

**Table 1.3.4. Other accessors**

<code>get_type_of_tree</code>	get expression's type
<code>get_size_of_type</code>	representation size of a type

---

## Chapter 1.4. State machine paradigm

### Table of Contents

1.4.1. Simple checker vs. checker with store .....	22
1.4.2. Abstract interpretation .....	22
1.4.3. Visit order .....	23
1.4.4. Manipulating the store .....	24
1.4.5. Example: tracking the sign of expressions .....	24

After inspecting the AST of a particular segment of code to decide what it means, a checker must then respond to this meaning. Typically, this is done via the state machine interaction model.

### 1.4.1. Simple checker vs. checker with store

The `simple` checker type (where `simple` is the second argument to `START_EXTEND_CHECKER`) is stateless. It is not sensitive to the order in which it encounters AST fragments. In the program analysis literature this is known as a *flow-insensitive analysis*.

In contrast, the `int_store` checker type is stateful. It has a *store*, which is a map from ASTs to values. This map can be used to implement an abstract interpreter, a concept explained in the next section. This is called *flow-sensitive analysis* and is what makes the Extend SDK so powerful.

### 1.4.2. Abstract interpretation

Abstract interpretation is a general framework for doing program analysis. The core of the analysis is an abstract store, which is a map from program variables to abstract values. At one extreme, the abstract values could actually be concrete values, and then we would have a real (concrete) interpreter. However, by abstracting the value space, we enable the analysis of programs with loops or inputs. The choice of abstraction is dictated by the property being checked.

For example, in an analysis that looks for occurrences of a call to `fopen` and then a call to `fclose`, you might use the following rules:

- variable maps to 1 means `fopen` has been called but `fclose` has not.
- variable is unmapped (mapped to nothing) means either `fopen` has not been called, or else `fclose` has subsequently been called.

As another example, you might want to check that a negative value is never cast to `unsigned`, and use a store with these rules:

- expression maps to 0 if it is negative.
- expression maps to 1 if it is negative or zero.
- expression maps to 2 if it is zero.

- expression maps to 3 if it is positive or zero.
- expression maps to 4 if it is positive.
- expression is unmapped if its sign is unknown

Note that these two examples not only use different abstract values (store ranges), but they map *from* different constructs (variables vs. expressions, the store domains). The choice of store domain is usually determined by concerns such as soundness and completeness: a simple domain such as local variables will tend toward a sound analysis (where a program bug implies a defect report), while a complex domain such as expressions will tend towards a complete analysis (where a bug-free program implies no defect report), though this characterization oversimplifies things.

In practice, it takes some experimentation to select a good store domain and range for your particular properties of interest.

### 1.4.3. Visit order

In addition to the *abstraction*, which was the subject of the previous section, an abstract interpreter also *interprets*. This means that it simulates execution, modulo the chosen abstraction.

Consequently, like a real interpreter, an abstract interpreter visits the abstract syntax statements and expressions in execution order. For example, in the following fragment:

```
x = y + z;  
a = foo(b, c*2);
```

the visit order is:

```
y  
z  
y + z  
x = y + z  
x  
x = y + z;           // statement  
b  
c  
2  
c*2  
foo(b, c*2)  
a = foo(b, c*2)  
a  
a = foo(b, c*2);    // statement
```

This is called postorder traversal: the children of a given node are visited (recursively) in order, and then the node itself is visited.

Left-hand sides of assignments are evaluated *after* the assignment, since the left-hand side becomes the value of the entire assignment expression.

The previous visit order can be seen by running the `hello` checker that we created in the previously (with the source file at `<HELLO>/hello.c`) on `<install_dir>/sdk/hello/test2/hello.test.c`.

When the abstract interpreter reaches choice points (such as an `if` statement), it first follows one path, then later backtracks to follow the other. In this way, all paths in the function are explored. See Chapter 1.7, *Paths* for more detail about paths.

## 1.4.4. Manipulating the store

The store is a map from expressions to integer values:

```
store : expression -> integer
```

Some of the common functions that are used to manipulate the map are described next. For a full listing, see Chapter 2.5, *The store*.

### 1.4.4.1. SET\_STATE(*t*, *v*)

Map expression tree *t* to *v*. Any prior mapping is removed.

### 1.4.4.2. GET\_STATE(*t*, *v*)

Retrieve the mapping for *t*. If it exists, `GET_STATE` returns true and stores the value in *v*. Otherwise, `GET_STATE` returns false and *v* is undefined.

### 1.4.4.3. MATCH\_STATE(*t*, *v*)

Return true if *t* is mapped to *v*.

### 1.4.4.4. CLEAR\_STATE(*t*)

Remove any mapping for *t*.

### 1.4.4.5. COPY\_STATE(*dst*, *src*)

First, `CLEAR_STATE(dst)`. Then, if *src* is mapped, copy its mapping to *dst*.

### 1.4.4.6. FOREACH\_IN\_STORE(*t*, *v*)

Using a loop, bind *t* and *v* to all of the (`expression tree`, `value`) pairs in the store. The bindings are retrieved in an undefined order. The store should not be modified during the iteration.

## 1.4.5. Example: tracking the sign of expressions

The previous store functions (except for `MATCH_STATE`) are demonstrated in `sign.cpp`, a checker that tracks the sign of expressions. See `<install_dir>/sdk/samples/sign/sign.cpp` or Section 3.1.1, “sign checker”.

After some preliminary code to define the abstract domain, it uses the store to remember sign information for variables and expressions.

When it sees certain function calls, it prints out some of the information that it is tracking. Since we haven't yet covered the output routines, this checker just uses `cout`.

The `sign.test.c` file (see `<install_dir>/sdk/samples/sign/test1/sign.test.c`) provides some basic input to the `sign.cpp` checker:

```
// sign.test.c
// test input for 'sign' checker

void whatis(int);
void print_store();
unsigned something();

int foo(int x, int y)
{
    whatis(x);
    whatis(y);

    int n_one = -1;
    int zero = 0;
    int one = 1;

    whatis(n_one);
    whatis(zero);
    whatis(one);

    whatis(n_one + zero);
    whatis(n_one + one);
    whatis(n_one - one);
    whatis(one + one);

    x = 3;
    y = 3;
    whatis(x+y);

    x = -3;
    y = -3;
    whatis(x+y);

    x = 0;
    x = x+1;
    whatis(x);

    unsigned u = something();
    whatis(u);

    x = zero - u;
    whatis(x);
    whatis(x + y);

    print_store();

    return 0;
}
```

```
}  
// EOF
```

Complete the same steps that you followed for the hello checker as follows:

1. Run `build-checker` on `sign.cpp` (`build-checker sign`).
2. Copy `sign` to the Coverity Analysis bin directory.
3. Run `cov-build` on the `sign.test.c` file.
4. Run the `sign` checker on the intermediate directory.

A portion of the output is shown next:

```
sign.test.c:10: "x" has unknown value  
sign.test.c:11: "y" has unknown value  
sign.test.c:17: "n_one" has value AV_NEGATIVE  
sign.test.c:18: "zero" has value AV_ZERO  
sign.test.c:19: "one" has value AV_POSITIVE  
sign.test.c:21: "(n_one + zero)" has value AV_NEGATIVE  
sign.test.c:22: "(n_one + one)" has unknown value  
sign.test.c:23: "(n_one - one)" has value AV_NEGATIVE  
sign.test.c:24: "(one + one)" has value AV_POSITIVE  
sign.test.c:28: "(x + y)" has value AV_POSITIVE  
sign.test.c:32: "(x + y)" has value AV_NEGATIVE  
sign.test.c:36: "x" has value AV_POSITIVE  
sign.test.c:39: "u" has value AV_POS_ZERO  
sign.test.c:42: "x" has value AV_NEG_ZERO  
sign.test.c:43: "(x + y)" has value AV_NEGATIVE  
sign.test.c:45: print_store:  
  "y" has value AV_NEGATIVE  
  "x" has value AV_NEG_ZERO  
  "u" has value AV_POS_ZERO  
  "zero" has value AV_ZERO  
  "one" has value AV_POSITIVE  
  "n_one" has value AV_NEGATIVE  
  "-1" has value AV_NEGATIVE  
  "0" has value AV_ZERO  
  "1" has value AV_POSITIVE  
  "(n_one + zero)" has value AV_NEGATIVE  
  "(n_one - one)" has value AV_NEGATIVE  
  "(one + one)" has value AV_POSITIVE  
  "3" has value AV_POSITIVE  
  "(x + y)" has value AV_NEGATIVE  
  "-3" has value AV_NEGATIVE  
  "(x + 1)" has value AV_POSITIVE  
  "(zero - u)" has value AV_NEG_ZERO  
17 mappings
```

---

## Chapter 1.5. Output

### Table of Contents

1.5.1. OUTPUT_ERROR .....	27
1.5.2. ADD_EVENT .....	27
1.5.3. COMMIT_ERROR .....	28
1.5.4. ADD_INPUTFILE_ONLY_EVENT .....	28
1.5.5. COMMIT_INPUTFILE_ONLY_ERROR .....	28
1.5.6. ADD_INPUTFILE_EVENT .....	28
1.5.7. COMMIT_INPUTFILE_ERROR .....	28
1.5.8. Example: sign2 .....	29

**Outputting defects.** So far we have just been using `cout` to communicate information from a checker, but the Coverity Extend SDK has a more sophisticated defect reporting mechanism with several advantages:

- The resulting reports are suitable for display in the Coverity Connect, just like other defect reports.
- They properly take into account path feasibility, a topic covered in more detail in Section 1.7.2, “False path pruning (FPP)”.
- A series of reports can be associated with specific variables or expressions, allowing the checker to communicate a timeline of important events in the diagnosis of the defect. This can greatly improve the comprehensibility of the report for flow-sensitive checkers.

### 1.5.1. OUTPUT\_ERROR

The simplest reporting routine is `OUTPUT_ERROR(<message>)`, where `<message>` has an `ostream` operator `<<` to its left. For example:

```
OUTPUT_ERROR("Zounds! " << some_expr << " is " << some_value);
```

This method has properties (a) and (b) from previous, but not (c). It is suitable for flow-insensitive checkers.

### 1.5.2. ADD\_EVENT

In Section 1.4.4, “Manipulating the store”, we oversimplified things a bit. The store actually maps an expression to a value and a set of events:

```
expression -> integer, set of events
```

An event is created by calling `ADD_EVENT(t, tag, text)`:

- `tree t` — The associated expression tree. Among other things, this is used to obtain the line number to display the event.
- `string tag` — Something that identifies the general kind of event, for example `var_assign`. It provides a name for hyperlinks pointing at the event. Also, source code annotations can be used to

suppress errors with a given event tag. Finally, the event tag affects CID merging; reports with different sets of event tags are never merged.

- `string text` — A string that explains the event, for example "a is assigned to the value of b".

The set of events associated with an expression are (conventionally) a history of what has happened to give the expression the abstract value it currently has. A good rule of thumb is that any time you set or change an abstract value, you should add an event explaining why.

The store operations explained in Section 1.4.4, "Manipulating the store" operate on events as well as values, transparently. For example, `CLEAR_STATE` removes all events, and `COPY_STATE` copies events.

It is not possible to associate an event with an expression (with `ADD_EVENT`) without first giving that expression an abstract value (with `SET_STATE`). In some cases, you might need to invent a new abstract value (for example, `unknown`) so that you can assign a value in order to attach an event.

Also note that simply adding an event to an expression will not cause it to be output. Rather, calling `COMMIT_ERROR(t, ...)` or `COMMIT_INPUTFILE_ERROR(t, ...)` will output the event that you add this way.

### 1.5.3. COMMIT\_ERROR

You use `COMMIT_ERROR(t, tag, text)` to output a defect report that contains all the events in the store that were previously associated with `t` and one final, main event given by `tag` and `text`:

- `tree t`: The expression tree with which events were previously associated through the use of `ADD_EVENT(t, ...)` or `ADD_INPUTFILE_EVENT(t, ...)`. It is an error if `t` does not have any associated events (in the current implementation, `COMMIT_ERROR` does nothing in this case).
- `tag/text`: These are the same as described in `ADD_EVENT` and are used to create one final event for the defect report (but this final event is not added to the store). If you pass the empty string (" ") for both, no final event is created.

### 1.5.4. ADD\_INPUTFILE\_ONLY\_EVENT

For a description, see `ADD_INPUTFILE_ONLY_EVENT(/*extend_inputfile_t*/ f, line, tag, text_utf8)`.

### 1.5.5. COMMIT\_INPUTFILE\_ONLY\_ERROR

For a description, see `COMMIT_INPUTFILE_ONLY_ERROR(/*extend_inputfile_t*/ f, line, tag, text_utf8)`.

### 1.5.6. ADD\_INPUTFILE\_EVENT

For a description, see `ADD_INPUTFILE_EVENT (tree, /*extend_inputfile_t*/ f, line, tag, text_utf8)`.

### 1.5.7. COMMIT\_INPUTFILE\_ERROR

For a description, see `COMMIT_INPUTFILE_ERROR (tree, /*extend_inputfile_t*/ f, line, tag, text_utf8)`.

### 1.5.8. Example: sign2

The error output routines are demonstrated in the `sign2.cpp` checker (see `<install_dir>/sdk/samples/sign2/sign2.cpp` or Section 3.1.2, “sign2 checker”), which is an extension of the `sign.cpp` checker discussed previously (see Section 1.4.5, “Example: tracking the sign of expressions”).

The main difference between the `sign.cpp` and `sign2.cpp` checkers is that in `sign2.cpp` every call to `SET_STATE` is followed by a call to `ADD_EVENT`. There are also some calls to `CLEAR_STATE`, used to remove previously-associated events in cases where a new value is being stored. `COPY_STATE` is used to copy events from one expression to another, to reflect the history of each value.

The `whatis` query now uses `COMMIT_ERROR` to output the events associated with the expression being queried, or `OUTPUT_ERROR` if nothing is known about the expression.

The `print_store` query uses an `ostringstream` to construct a big string, which it then sends to `OUTPUT_ERROR`.

Finally, this checker has some true defect detection because it recognizes typecasts from signed to unsigned where the source might be negative. Though these potential defects include many false positives at first, the premise for this checker is to suppress them by adding assertions. However, in order to respond to assertions, the checker needs to handle conditionals, which is the subject of the next section.

---

## Chapter 1.6. Conditionals

### Table of Contents

1.6.1. ANALYZE_CONDITION .....	30
1.6.2. MATCH_COND .....	30
1.6.3. force_backtrack .....	31
1.6.4. Example: sign3 .....	31
1.6.5. Abstract comparison .....	31
1.6.6. Comparison evaluation .....	32

A checker has two fundamental sources of information about a path: assignments and conditionals. Assignments are handled by computing an abstraction of the right-hand side, and storing that abstraction in the left-hand side. That is, they simply correspond to updating the store.

Conditionals, on the other hand, act as *constraints*: the current abstract state (the store) must be refined in such a way as to be consistent with the branch of the conditional being taken. The refinement algorithm is dependent on the abstraction being used by the checker.

### 1.6.1. ANALYZE\_CONDITION

ANALYZE\_CONDITION handles conditional guards in the same way that ANALYZE\_TREE handles statements and expression side effects.

### 1.6.2. MATCH\_COND

Within ANALYZE\_CONDITION, you can use MATCH\_COND to inspect the guard expression. MATCH\_COND automatically takes account of whether the true or false branch is being taken. For example, if the code to analyze says:

```
if (x == y) {  
  // then-branch  
}  
else {  
  // else-branch  
}
```

then the checker fragment:

```
Expr a, b;  
MATCH_COND(a != b)
```

matches only when the `else` branch is followed.

#### Note

It is important to note that MATCH\_COND only works when matching comparisons. For instance, MATCH\_COND(`a == b`) will match a true `a == b` condition or a false `a != b` condition. Conditions are always comparisons *except when the condition is a non-comparison boolean*

*expression*. For instance, the condition in the expression `int x; if(x) { }` will be transformed into `x != 0`, but the condition in the expression `bool x; if(x) { }` will simply be `x`. In the latter case, `MATCH_COND` will not work, so you might instead check the `cov_polarity` variable, which indicates whether the condition being evaluated is true or false. For example, `MATCH_COND(a == b)` is equivalent to `cov_polarity ? MATCH(a == b) : MATCH(a != b)`.

### 1.6.3. force\_backtrack

Often, an analysis is able to determine that a path is *infeasible*, meaning that it cannot be executed at run time. This discovery happens when the engine attempts to traverse through a conditional that is inconsistent with the known facts in the store. In such cases, the checker can call `force_backtrack()`, which stops further exploration of this path.

### 1.6.4. Example: sign3

The `sign3.cpp` checker illustrates handling conditionals (see `<install_dir>/sdk/samples/sign3/sign3.cpp` or Section 3.1.3, “sign3 checker”).

### 1.6.5. Abstract comparison

The core of the checker is in the section titled *abstract comparison*, which defines what it means to compare two abstract values using operators like `<` and `==`. Whenever an operator `<op>` is used to compare abstract values `a` and `b`, and we follow the path where the comparison yields true, there are several possible consequences embodied in `AbstractComparisonResult`:

1. We might decide that the comparison could not possibly have yielded true, in which case its truth is *inconsistent* with the current abstract state. For example:

```
int x = 0;
if (x < 0) {
    // inconsistent, i.e., unreachable
}
```

2. We might discover new facts about `a` or `b`. For example:

```
int x = 0;
if (x == y) {
    // discover that 'y' equals 0 as well
}

if (y <= 0 && z >= 0) {
    // discover: y <= 0
    // discover: z >= 0
    if (z == y) {
        // discover: y == 0 and z == 0
    }
}
```

3. We might discover no new information. For example:

```

if (x < y) {
    // lacking any previous information about 'x' and 'y', the
    // constraint 'x < y' cannot be expressed in our abstraction
}

```

In the `sign3` checker, this computation is performed by the `abstractComparison` function. This function is probably more complicated than ones you write, but illustrates the general technique in a realistic setting.

Because there are six relational comparison operators, there are six precomputed abstract relation tables stored in the `relationalOperators` global variable. The `RelationalOperator` class effectively maps from an AST tree code to an abstract operator table.

### 1.6.6. Comparison evaluation

The `ANALYZE_CONDITION` function begins by attempting to match the input condition with each of the six relational operators:

```

for (int i=0; i < NUM_RELATIONAL_OPERATORS; i++) {
    RelationalOperator *relop = relationalOperators[i];
    if (MATCH_COND(Binop(relop->treeCode, a, b))) {

```

When it finds a match, it abstractly evaluates the arguments `a` and `b`; an unmapped expression corresponds to `AV_UNKNOWN`:

```

    if (!GET_STATE(a, va)) { va = AV_UNKNOWN; }
    if (!GET_STATE(b, vb)) { vb = AV_UNKNOWN; }

```

It then performs the abstract comparison by doing a table lookup:

```

    AbstractComparisonResult &res = relop->map[va][vb];

```

If the comparison is infeasible, that is, it could not possibly have evaluated to true, then we abort analysis of the current path:

```

    if (!res.consistent) {
        force_backtrack();
    }

```

Otherwise, if incorporation of the new constraint has led to a refinement of the abstract value of `a` or `b`, then the store is updated accordingly:

```

    if (res.newAValue != va) {
        SET_STATE(a, res.newAValue);
    }
    if (res.newBValue != vb) {
        SET_STATE(b, res.newBValue);
    }

```

With these refinements, the `sign3` checker is able to confirm that `<install_dir>/sdk/samples/sign3/test1/sign3.test.c` never converts a negative integer to unsigned even though there are three places that have implicit conversions from `int` to `unsigned`. A portion of the output is shown next:

## Conditionals

---

```
matched conditional "x >= 0"; "x" = AV_UNKNOWN, "0" = AV_ZERO
  refined "x" to AV_POS_ZERO
matched conditional "y > 0"; "y" = AV_NEG_ZERO, "0" = AV_ZERO
  backtracking due to inconsistency
matched conditional !"y > 0"; "y" = AV_NEG_ZERO, "0" = AV_ZERO
matched conditional "x == z"; "x" = AV_POS_ZERO, "z" = AV_UNKNOWN
  refined "z" to AV_POS_ZERO
matched conditional !"x == z"; "x" = AV_POS_ZERO, "z" = AV_UNKNOWN
matched conditional !"x >= 0"; "x" = AV_UNKNOWN, "0" = AV_ZERO
  refined "x" to AV_NEGATIVE
```

---

## Chapter 1.7. Paths

### Table of Contents

1.7.1. Many paths per function .....	34
1.7.2. False path pruning (FPP) .....	34
1.7.3. Two-pass checking .....	35
1.7.4. Termination .....	35

**Realities of paths.** The mechanics of path traversal are relevant when writing a checker. Among other things, this chapter explains why using `cout` often produces misleading results.

### 1.7.1. Many paths per function

The Coverity Extend SDK engine does not simply traverse one path through a function; instead it traverses many paths to achieve complete coverage of all relevant sequences of operations.

The paths are not executed in sequence, one after another. Instead, common prefixes of paths are analyzed once with the analysis branching at conditionals to investigate each path separately.

What this means is that your checker's handler functions such as `ANALYZE_TREE` are called for expression and statement trees in different paths at different times; from the checker's point of view, the engine seems to be jumping from path to path unpredictably. This is why using `cout` is misleading.

The way to keep track of separate paths is by using the store. The Coverity Extend SDK engine will always call a handler with the same store (contents) for the same path. Thus, all knowledge about the current path should be saved to the store.

Since the events are also in the store, using `ADD_EVENT` is the best way to produce coherent, path-dependent output.

### 1.7.2. False path pruning (FPP)

Among the reasons for what at first might seem to be unpredictable switching among paths is false path pruning (FPP). Your Coverity Extend SDK checker is actually running in conjunction with a number of FPP modules, each of which is looking for sequences of operations that are inconsistent. For example, in this code:

```
int x = 2;
if (x != 2) {
    // unreachable
}
```

there is an FPP module that detects that the *then* branch of this conditional cannot be taken. That FPP module calls `force_backtrack`, but won't otherwise inform your checker, so that the checker finds the next `ANALYZE_TREE` coming from a different path.

### 1.7.3. Two-pass checking

Because the FPP modules are somewhat computationally expensive, they are disabled until the checker calls either the `COMMIT_ERROR` or `OUTPUT_ERROR` functions. Once the checker tries to output an error, the analysis of that function is restarted with the FPP modules enabled. This saves time in the usual case where a checker does not find any problems on any path, but still filters out reports from infeasible paths.

This is another reason why `cout` produces misleading results: your checker appears to analyze some functions twice due to the activation of the second pass.

### 1.7.4. Termination

Any function that has a loop will have an infinite number of apparent paths. Even with FPP enabled, there might still be no bound on the number of paths. For example, consider the following loop:

```
void foo(int n)
{
    for (int i=0; i<n; i++) {
        // ...
    }
}
```

**How does the Coverity Extend SDK engine avoid running forever checking an example like this?** The Coverity Extend SDK engine notices that the second and subsequent paths through the loop are not significantly different from the first iteration, and stops analyzing the loop. This condition is called a fixpoint of the loop.

The concept of *significantly different* is difficult to describe, and is not necessary to understand fully to write a checker. You must be aware that the contents of the store is the key determiner of whether something is considered different by the Coverity Extend SDK; engine. As a first approximation: exploration of the loop terminates if and only if two different iterations produce the same store. Although the actual rule for this behavior is more complex, this abstraction is generally accurate.

What this means for a checker writer is that you must not try to track values too precisely, otherwise you risk putting the Coverity Extend SDK engine into an infinite loop. You must choose abstractions that are precise enough to check the property of interest, but sufficiently *imprecise* to allow termination.

A simple example of the imprecise concept is integers. If you track integer values exactly, then you have an infinite abstract domain and hence the analysis does not terminate. But by abstracting this down to just three values (negative, zero, and positive), you can ensure termination.

If you track values of arbitrarily complex expressions, then there is no guarantee of reaching a fixpoint in a loop, so the abstract interpretation could go on forever. As another example, when tracking values in the heap, you should avoid tracking information about arbitrarily deep nested pointer dereference expressions. For example, tracking `p->field` is usually fine, but tracking `p->field1->field2->field3` is not, since the latter has enough precision to take considerable (infinite) time exploring all of its variations.

---

## Chapter 1.8. Examining the class hierarchy

### Table of Contents

1.8.1. Introduction .....	36
1.8.2. Mapping from variables to their class/type .....	36
1.8.3. Tree structure of types .....	37
1.8.4. Classes .....	37
1.8.5. type_t iterators .....	38
1.8.6. When are type_t objects resident in memory? .....	38
1.8.7. Example: print type information .....	39
1.8.8. Example: switch default .....	39

### 1.8.1. Introduction

You can use the Coverity Extend SDK to examine the class hierarchy of C++, C#, or java programs. This capability consists of an in-memory representation of the hierarchy and an API to query that representation.

The representation and API are declared in `<install_dir>/sdk/headers/types/extend-types.hpp`. The comments in that header file are the definitive documentation for individual methods. This section describes how to use the API at a high level, and Chapter 2.7, *Types* explains it at an intermediate level of detail.

### 1.8.2. Mapping from variables to their class/type

To obtain the type of an expression or variable, match it with a pattern that inherits from `TypedExpr` (such as `Expr`), and then use the `TypedExpr::get_type_t()` method.

This returns a pointer to a `types::type_t` (hereafter referred to as `type_t`) object, which is the root of a C++ class hierarchy for representing types.

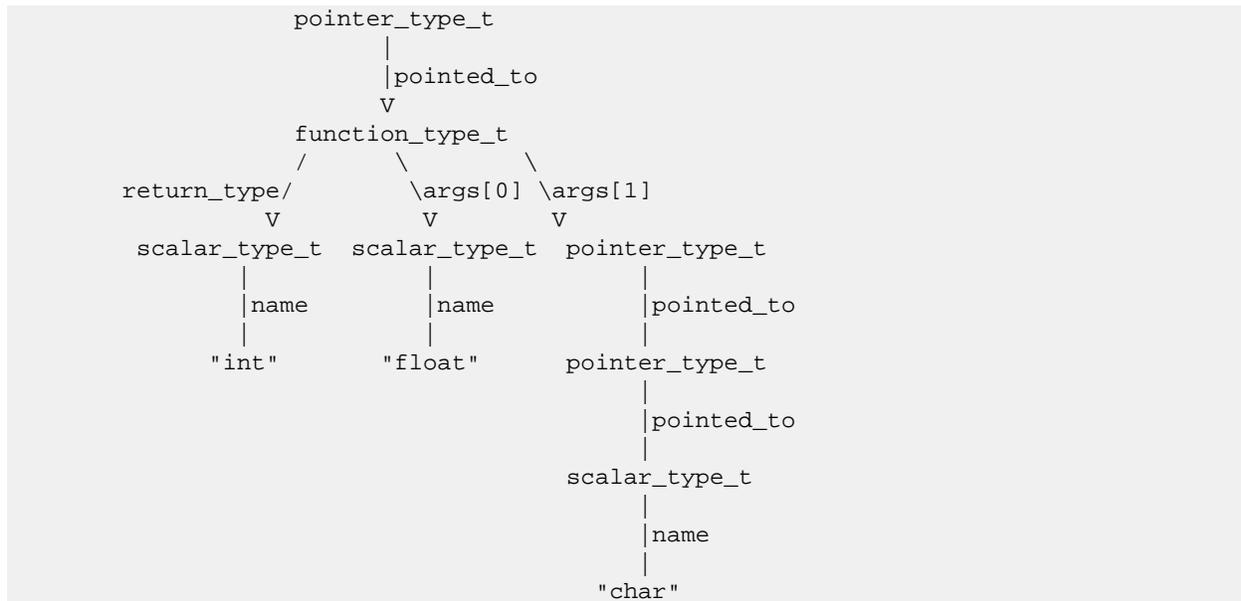
The `type_t` class has a number of methods for querying the actual type. For example, `type_t::as_class_p()` returns a pointer to a `class_type_t` if that `type_t` is a `class_type_t`, otherwise it returns `NULL`.

The following is an Coverity Extend SDK checker fragment that checks for the current expression being a pointer to a class:

```
Expr e;
if (MATCH(e)) {
    if (type_t *t = e.get_type_t()) {
        if (class_type_t *ct = t->as_class_p()) {
            // now 'ct' is the class_type_t representing the type
            // of the expression matched by 'e'
        }
    }
}
```

### 1.8.3. Tree structure of types

A given `type_t` object is actually a tree (or subtree). For example, the type `int (*)(float, char**)` looks like the following:



`scalar_type_t` is a leaf type.

The `class_type_t` and `union_type_t` classes are also leaf types of a sort. Since all recursive types (such as the type of a linked list) must go through a class or struct or union in order to recurse, you can think of them as leaves — so that types really are trees, rather than arbitrary graphs. However, this "leafness" is a property of the code that *uses* the `type_t` structure. It's merely a convenient convention. This convention is used by `type_recursive_visitor_t`, among others.

### 1.8.4. Classes

Each class in the program being analyzed is represented by a `class_type_t` object in the Extend SDK.

#### 1.8.4.1. Inheritance

Given a `class_type_t`, the inheritance hierarchy can be examined using, for example, the following:

```

class_type_t *c = ...;
defined_class_type_t d = c->load_definition();
if(d) {
    d->get_parents();
}

```

This returns a vector of the immediate ancestors.

To iterate over all parents, including those that are inherited, use:

```
class_type_t *c = ...;
defined_class_type_t d = c->load_definition();
if(d) {
    d->get_all_parents();
}
```

### 1.8.4.2. Virtual function overriding relationships

Given two `function_t` objects, the `function_t::overrides()` method can be used to determine whether one overrides the other. This function can only be called if the class hierarchy allows it. For example, if you call `f1->overrides(f2)`, then `f1->get_owner_class()->derives_from(f2->get_owner_class())` must be true.

### 1.8.5. `type_t` iterators

The `type_t` visitor classes allow types to be processed using the *visitor* pattern, which is sometimes more convenient than explicit recursive processing.

#### 1.8.5.1. `type_visitor_t`

The `type_visitor_t` is an interface that clients can implement. The handlers, such as `on_function` or `on_class`, react to the various kinds of `type_t` nodes. Invoking its `operator()` on a `type_t` object invokes the appropriate handler for the dynamic type of that object.

#### 1.8.5.2. `subtype_visitor_t`

The `subtype_visitor_t` is another interface. When a type object's `iter_subtypes` method is invoked, the type object invokes the passed `subtype_visitor_t::operator()` on each of its component or "sub" types. For example, the fields of a class are considered sub types, as is the `pointed_to` element of a `pointer_type_t`.

#### 1.8.5.3. `type_recursive_visitor_t`

The previous two interfaces are combined to form the `type_recursive_visitor_t`, which is both an interface and a tree traversal mechanism. To use this class, inherit from it, and then implement the appropriate `on_XXX` methods. By default, these methods recursively traverse their sub types (hence the *recursive* in the name of the visitor), except for classes and unions. That is, they visit the tree structure described in Section 1.8.3, "Tree structure of types". If you override one of the methods other than `on_class` or `on_union`, you must call the superclass method if you want recursive traversal to proceed below the overridden point.

### 1.8.6. When are `type_t` objects resident in memory?

In general, a checker should not save `type_t` pointers beyond the analysis of the function when they were obtained. The reason is that `type_t` objects get loaded and unloaded as the analysis runs. The

analysis guarantees to keep resident the set of types defined when the function being analyzed was compiled, but not beyond that.

### 1.8.7. Example: print type information

The PRINT\_TYPES checker example (see also, `<install_dir>/sdk/samples/print_types/print_types.cpp`) demonstrates some of the `type_t` API. The checker itself simply matches declarations and uses the declared types as a root set in a recursive exploration of the program's type hierarchy. It also prints out the name and type of those declarations. The bulk of the work is done by the `ClassTypeVisitor`, which inherits from `type_recursive_visitor_t`. Its `on_class` method prints out the base classes and (non-static) fields of each class that can be reached from the root set of types. It keeps a set of class names so it can avoid printing information about the same class twice. Note that, as explained in the previous section, you cannot keep a set of `class_type_t` pointers.

### 1.8.8. Example: switch default

The SWITCH\_DEFAULT checker example (see also, `<install_dir>/sdk/samples/switch_default.cpp`) demonstrates how to match the `case` and `default` substatements of a switch case statement. It reports switch statements that do not have an explicit `default` statement and shows how it is possible to extract the particular values in `case` statements.

---

## Chapter 1.9. Reporting events and defects on input files

### Table of Contents

1.9.1. Additional steps for building Coverity Extend SDK checkers for Android applications .....	40
1.9.2. Input file class <code>extend_inputfile_t</code> .....	40
1.9.3. Input file macros .....	40
1.9.4. Input file checker examples .....	42

In addition to analyzing ASTs and reporting defects in source code, Coverity Extend SDK checkers can also inspect the contents of files captured during the Coverity build and emit processes and report defects in them. These files can include source files, files packaged within a WAR file (and emitted with `cov-emit-java --webapp-archive` or similar), or an Android `AndroidManifest.xml` and its associated APK file (emitted with the `--android-apk` and `--input-file` options to `cov-emit-java`).

### 1.9.1. Additional steps for building Coverity Extend SDK checkers for Android applications

The general workflow uses `cov-emit-java` to emit the files to the intermediate directory, then runs the custom checkers on the emitted files. The checkers iterate over the input files to produce the reports.

1. Emit files, such as `AndroidManifest.xml`, that your checker requires to the intermediate directory. Such files must be associated with an APK, for example:

```
cov-emit-java --dir myIntDir --android-apk myAPK.apk --input-file
AndroidManifest.xml
```

The `--input-file` option is required and can be specified multiple times. The `--android-apk` option is required and can only be specified once. Descriptions of these command line options are in the *Coverity 2020.12 Command Reference*.

2. Run your custom checker to analyze the input files.

### 1.9.2. Input file class `extend_inputfile_t`

This methods in this class are for iterating over and reporting defects on files built or emitted by Coverity Analysis processes.

### 1.9.3. Input file macros

#### `FOREACH_MATCHING_INPUTFILE(f, suffix_utf8)`

- This macro iterates over the input files (including source and application archives) when the filesystem path matches the specified suffix.
  - `f` is of type `extend_inputfile_t` and will be set in each iteration.

- `suffix_utf8` is of type `const char *`, which is interpreted as a NUL-terminated UTF-8 sequence.

Arguments to the function declarations in the remaining macros (below) share the following characteristics:

- `f` is of type `extend_inputfile_t`.
- `line` is a positive integer line number on which to report the event.
- `tag` and `text_utf8` are of type `const char *`, which is interpreted as a NUL-terminated UTF-8 sequence.

#### **ADD\_INPUTFILE\_ONLY\_EVENT(*/\*extend\_inputfile\_t\*/ f, line, tag, text\_utf8*)**

- Queue an event (in an input file) to report at the next use of `COMMIT_INPUTFILE_ONLY_ERROR`. This macro is useful when reporting events that are entirely inside input files. It is most appropriate to use this macro inside `CHECKER_INIT` or `CHECKER_FINAL`.

#### **COMMIT\_INPUTFILE\_ONLY\_ERROR(*/\*extend\_inputfile\_t\*/ f, line, tag, text\_utf8*)**

- Report an error in an input file. This macro will include only events previously queued through `ADD_INPUTFILE_ONLY_EVENT`. This is useful when reporting events that are entirely inside input files.

It is most appropriate to use this macro inside `CHECKER_INIT` or `CHECKER_FINAL`.

#### **ADD\_INPUTFILE\_EVENT (tree, */\*extend\_inputfile\_t\*/ f, line, tag, text\_utf8*)**

- Create an event in input file `f`, and add it to the set of events in the store for `tree`. A subsequent use of `COMMIT_INPUTFILE_ERROR(tree, ...)` or `COMMIT_ERROR(tree, ...)` will include this event in the defect report it produces. This macro is for reporting events in input files in the context of a flow-sensitive checker; it allows mixing events in source code and input files.

Use this macro inside any `ANALYZE_*` or `FUNCTION_INIT` handler.

Note that subsequent use of `COMMIT_INPUTFILE_ERROR` or `COMMIT_ERROR` will include this event in the defect it creates.

For related information, see Chapter 1.9, *Reporting events and defects on input files*.

#### **COMMIT\_INPUTFILE\_ERROR (tree, */\*extend\_inputfile\_t\*/ f, line, tag, text\_utf8*)**

- Report the error in an input file using the events from `tree`, an AST node.

This macro is for use with a flow-sensitive checker and allows mixing events in source code and input files.

Use this macro inside any `ANALYZE_*` or `FUNCTION_INIT` handler.

For related information, see Chapter 1.9, *Reporting events and defects on input files*.

### 1.9.4. Input file checker examples

The sample checkers iterate over input files, querying encoding and parent archives, loading the contents to the emit (part of the intermediate directory), and reporting defects in both simple and stateful checkers.

Location: `<intermediate_directory>sdk/samples`

- `java_input_file_simple.cpp`
- `java_input_file_stateful.cpp`

---

## Chapter 1.10. Troubleshooting

### Table of Contents

1.10.1. A Coverity Extend SDK checker aborts execution with a Tree used with no match error. .... 43

#### 1.10.1. A Coverity Extend SDK checker aborts execution with a Tree used with no match error.

Example:

```
[STATUS] Reading call graph
[STATUS] Computing callgraph.
|0-----25-----50-----75-----100|
*****
[STATUS] Incremental analysis could not be used - this may take a while.
[STATUS] Starting analysis run (analysis pass)
|0-----25-----50-----75-----100|
*****|
extend-patterns.hpp:111:operator
P5_tree: Tree used with no match in pattern Var
SM NAME: a_never_follows_b
ANALYZING: a_never_follows_b_test.cpp:_Z5test6i
LINE 50: "B()"
0xb23c80 0xb23c68 call_expr, NAME: _Z1Bv TYPE: void
-0xb23c38 0xb23c20 addr_expr TYPE: void (void)*
-0xb23f08 0xb23bc0 function_decl NAME: _Z1Bv, public TYPE: void (void)
```

```
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
Returned with error code 0x3 at function 6.
```

Solution:

A pattern has been used before it matches anything.

For example, you can get this error if you run the following checker:

```
#include "extend-lang.hpp"
START_EXTEND_CHECKER( a_never_follows_b, int_store );
ANALYZE_TREE()
{
  CallSite a( "A" );
  CallSite b( "B" );
  Var v;
  tree t;
  int i;
  if( MATCH( b ) ) {
    SET_STATE( v, 1 );
    ADD_EVENT( v, "B", "B is called" << v );
  }
}
```

```
}
else if( MATCH( a ) ) {
FOREACH_IN_STORE( t, i ) {
ADD_EVENT( v, "A", "A is called" << v );
COMMIT_ERROR( t, "A", "A is follows b" << t );
}
}
// CLEAR_STATE( v );
}
END_EXTEND_CHECKER();
MAKE_MAIN( a_never_follows_b )
```

against the following test case:

```
extern void A();
extern void B();

// OK
void test1() {
    A();
}

// OK
void test2() {
    B();
}

// OK
void test3() {
    A();
    B();
}

// Defect
void test4() {
    B();
    A();
}

// OK
void test5(int x) {
    if (x) {
        A();
    } else {
        B();
    }
}
if (!x) {
    B();
} else {
    A();
}
```

```
}  
  
// Defect  
void test6(int x) {  
    if (x) {  
        A();  
    } else {  
        B();  
    }  
    if (!x) {  
        A();  
    } else {  
        B();  
    }  
}
```

---

## Part 2. Coverity Extend SDK Reference

### Table of Contents

2.1. Introduction .....	48
2.2. Handler functions .....	49
2.2.1. Handler function overview .....	49
2.3. Patterns .....	57
2.3.1. Patterns for C# and Java checkers .....	58
2.3.2. Functions common to all patterns .....	58
2.3.3. ASTNodePattern superclass .....	59
2.3.4. ExpressionPattern superclass .....	59
2.3.5. TypePattern superclass .....	60
2.3.6. SymbolPattern superclass .....	60
2.3.7. Predefined pattern objects .....	60
2.3.8. Expression patterns .....	60
2.3.9. Statement patterns .....	67
2.3.10. Other patterns .....	68
2.4. Accessors .....	71
2.4.1. Additional AST query functions .....	71
2.4.2. Queries on the current function .....	71
2.4.3. Queries on the current file .....	72
2.4.4. Queries on trees .....	72
2.5. The store .....	73
2.5.1. Store overview .....	73
2.5.2. void SET_STATE(tree t, int v) .....	73
2.5.3. void CLEAR_STATE(tree t) .....	74
2.5.4. bool GET_STATE(tree t, int &v) .....	74
2.5.5. bool MATCH_STATE(tree t, int v) .....	74
2.5.6. bool COPY_STATE(tree dst, tree src) .....	74
2.5.7. FOREACH_IN_STORE(tree &t, int &v) { body } .....	74
2.5.8. bool ADD_EVENT(tree t, char const *tag, desc) .....	74
2.5.9. bool COMMIT_ERROR(tree t, char const *tag, desc) .....	74
2.5.10. ADD_INPUTFILE_EVENT .....	75
2.5.11. COMMIT_INPUTFILE_ERROR .....	75
2.6. Adding events .....	76
2.7. Types .....	77
2.7.1. Introduction .....	77
2.7.2. type_t .....	77
2.7.3. any_type_t .....	78
2.7.4. scalar_type_t .....	78
2.7.5. pointer_type_t .....	78
2.7.6. array_type_t .....	79
2.7.7. cv_wrapper_type_t .....	79
2.7.8. function_type_t .....	79
2.7.9. scoped_type_t .....	79

---

---

2.7.10. scope_t .....	79
2.7.11. defined_type_t .....	80
2.7.12. typedef_type_t .....	80
2.7.13. forward_declarable_type_t .....	80
2.7.14. enum_type_t .....	80
2.7.15. tag_t .....	80
2.7.16. union_type_t .....	80
2.7.17. field_t .....	81
2.7.18. class_type_t .....	81
2.7.19. parent_t .....	81
2.7.20. function_t .....	82
2.7.21. member_type_t .....	82
2.7.22. extend_inputfile_t .....	82
2.8. Reference information .....	83
2.8.1. Header files .....	83
2.8.2. Name mangling .....	83

---

---

## Chapter 2.1. Introduction

The Coverity Extend Software Development Kit (Coverity Extend SDK) is a framework for writing program analyzers (that is, *checkers*) in C++ that support the analysis of C/C++, C#, and Java applications.

Features include:

- Basic frontend features: Parsing, type checking and elaboration, abstract syntax construction, template instantiation, and linking across translation units.
- Facilities to inspect abstract syntax using pattern matching.
- Mechanism to traverse paths in the abstract syntax in execution order, prune false paths, and merge similar states to ensure termination in loops.
- Flexible notion of checker *state* for checking flow-sensitive properties.
- Output routines that work with the false path pruning (FPP) mechanism to ensure that defects are only detected from feasible paths.

This guide provides detailed information about the Coverity Extend SDK API. See Part 1, “Coverity Extend SDK Usage” for basic concepts and an introduction to using the Coverity Extend SDK.

---

## Chapter 2.2. Handler functions

### Table of Contents

2.2.1. Handler function overview .....	49
--	----

### 2.2.1. Handler function overview

The core of an Coverity Extend SDK checker is a set of handler functions. These handlers are called by Coverity Analysis to inform the checker of key events.

#### 2.2.1.1. Coverity Extend SDK checker file structure

An Coverity Extend SDK checker source file is organized as follows:

```
// checker_name.c
// (comment about what the checker does)

#include "extend-lang.hpp"          // Coverity Extend SDK API

(1)

START_EXTEND_CHECKER( checker_name, checker_type );

(2)

END_EXTEND_CHECKER();

MAKE_MAIN( checker_name )
```

In section (1) you can define arbitrary C/C++ functions and data structures. Syntactically, it is in the global scope.

In section (2) you define the checker handler functions. Syntactically, section (2) is inside a class definition.

You can define member variables inside section (2); doing so is somewhat cleaner than defining them as global variables in section (1), but either method works. However, member variables cannot be initialized at the declaration site (see Section 2.2.1.5, "INIT\_OPTIONS").

#### 2.2.1.2. START\_EXTEND\_CHECKER

##### Synopsis

```
START_EXTEND_CHECKER( checker_name, checker_type );
```

##### Description

The `START_EXTEND_CHECKER` macro call ends section (1) and begins section (2). This macro begins a class declaration.

### Arguments

`checker_name` is the name of your checker. This is the same name as the name of the source file (without the `.c` extension). This name is used by Coverity Connect to identify defect reports that are created by your checker.

`checker_type` is one of the following:

- `simple`— Checker type used for flow-insensitive (stateless) checkers. It has no store. For more information about the store, see Chapter 2.5, *The store*.
- `int_store`— Checker type used for flow-sensitive (stateful) checkers. Its store maps from expressions to integers; the exact meaning of the integers is up to you to establish.
- `type`— A special kind of checker that has no store and does not analyze abstract syntax trees. It visits each class that has been defined. See Section 2.2.1.17, “ANALYZE\_CLASS”.

### 2.2.1.3. END\_EXTEND\_CHECKER

#### Synopsis

```
END_EXTEND_CHECKER( );
```

#### Description

This macro call ends section (2). It terminates the class declaration started by `START_EXTEND_CHECKER( )`.

### 2.2.1.4. MAKE\_MAIN

#### Synopsis

```
MAKE_MAIN( checker_name );
```

#### Description

This macro call defines the `main` function of the entire checker program. `checker_name` is the name of the checker and should be the same as the name of the source file (without the `.c` extension).

The call to `MAKE_MAIN` should come directly after the call to `END_EXTEND_CHECKER`, and should be the last statement in the checker source file.

### 2.2.1.5. INIT\_OPTIONS

#### Synopsis

```
INIT_OPTIONS( ) { <code> }
```

#### Description

The code in `INIT_OPTIONS` is executed at program startup. Use this code to initialize member variables declared in section (2). This is the first handler function that is called when the program starts up (even before `CHECKER_INIT`).

### 2.2.1.6. HANDLE\_OPTION

#### Synopsis

```
HANDLE_OPTION() { <code> }
```

#### Description

This handler is called for every command-line argument to the `cov-analyze` command of the form:

```
(--checker_option|-co) checker_name:option_name[:option_value]
```

where `checker_name` is the same as the argument to `START_EXTEND_CHECKER`.

The following macros can be used in `HANDLE_OPTION` to determine the argument:

- `CHECK_OPTION(opt) { <code> }`— Executes code if the `option_name` on the command line equals `opt`.
- `OPTION_VALUE - <option_value>` (as a `char const *`) passed on the command line, or `NULL` if no value was passed.
- `OPTION_HANDLED()`— Tells Coverity Analysis that the command-line option has been recognized and processed. Calling this function causes a return from `HANDLE_OPTION`.
- `OPTION_NOT_HANDLED()`— Tells Coverity Analysis that the option has *not* been recognized. Calling this function causes a return from `HANDLE_OPTION`.

For example:

```
// The following member variables are located between
// the checkers START_EXTEND_CHECKER and END_EXTEND_CHECKER.
int mode;
bool use_mangled;

// called at program startup
INIT_OPTIONS()
{
    mode = 0;
    use_mangled = false;
}

// called for each --checker_option command-line argument
HANDLE_OPTION()
{
    CHECK_OPTION( "mode" ) {
        mode = atoi( OPTION_VALUE );
        OPTION_HANDLED();
    }
}
```

```

    }
    CHECK_OPTION( "use_mangled" ) {
        use_mangled = true;
        OPTION_HANDLED();
    }
    OPTION_NOT_HANDLED();
}
// You can now use mode and use_mangled in ANALYZE_TREE.

```

### 2.2.1.7. CHECKER\_INIT

#### Synopsis

```
CHECKER_INIT() { <code> }
```

#### Description

This function is called at program startup, after `INIT_OPTIONS` and `HANDLE_OPTION` have been called. Initialization code, particularly code that depends on the command-line options, can be placed in this function.

### 2.2.1.8. CHECKER\_FINAL

#### Synopsis

```
CHECKER_FINAL() { <code> }
```

#### Description

This function is called when the program is about to terminate.

### 2.2.1.9. FUNCTION\_INIT

#### Synopsis

```
FUNCTION_INIT() { <code> }
```

#### Description

This function is called when the checker is about to start analyzing a function. You can use functions such as `current_function_get_name` [p. 71 ] to get information about the function that will be analyzed.

### 2.2.1.10. FUNCTION\_FINAL

#### Synopsis

```
FUNCTION_FINAL() { <code> }
```

#### Description

This function is called when the checker is finished analyzing all of the paths in a function.

### 2.2.1.11. ANALYZE\_TREE

#### Synopsis

```
ANALYZE_TREE() { <code> }
```

#### Description

This is one of two central checker functions (`ANALYZE_CONDITION` is the other). It is called for each statement and expression, as described in Section 1.4.3, “Visit order”.

#### Options

Within the body of `ANALYZE_TREE`, you can use several macros to inspect the AST fragment that is undergoing analysis:

- `CURRENT_TREE`— The AST node that is undergoing analysis. It has type `tree`.
- `MATCH(pat)`— Matches `CURRENT_TREE` against pattern `pat`, returning true if it matches. See Chapter 2.3, *Patterns*.
- `MATCH_TREE(pat, t)`— Matches `t` against pattern `pat`.

### 2.2.1.12. ANALYZE\_CONDITION

#### Synopsis

```
ANALYZE_CONDITION() { <code> }
```

#### Description

This handler is called for every conditional expression through which the current path passes (see Chapter 1.7, *Paths* for more information). To inspect the condition, use `MATCH_COND(pat)`. This returns true if the current condition matches `pat`.

#### Note

Do not use `MATCH` or `MATCH_TREE` within `ANALYZE_CONDITION`. The latter has a *polarity* notion that only `MATCH_COND` can handle properly.

### 2.2.1.13. ANALYZE\_END\_OF\_PATH

#### Synopsis

```
ANALYZE_END_OF_PATH() { <code> }
```

#### Description

This function is called each time that the end of a particular path is reached. It is typically used in checkers that need to flag the absence of something along a path.

Since multiple paths are analyzed, this function can be called many times for a single function.

### 2.2.1.14. PREFER\_TO\_ANALYZE\_CSHARP

#### Synopsis

```
PREFER_TO_ANALYZE_CSHARP()
```

#### Description

This function makes the checker analyze C#-based output in the intermediate directory. By default, checkers otherwise analyze the C/C++ output.

Example:

```
START_EXTEND_CHECKER( cs1, simple );
PREFER_TO_ANALYZE_CSHARP();
ANALYZE_TREE()
{
    ...
}
```

The following error occurs if the intermediate directory does not contain C# output:

```
[ERROR] This program operates on Static C#
but specified intermediate directory <int-dir>
only contains data for C/C++.
```

#### Note

If you have both C# and C/C++ output in your intermediate directory and want to use your checker on the C/C++, you can use the following option on the command line to override PREFER\_TO\_ANALYZE\_CSHARP: `--cpp`

For example:

```
> <checker-name> --dir <intermediate_directory> --cpp
```

On the other hand, if you want to analyze C# output with a checker that does not call PREFER\_TO\_ANALYZE\_CSHARP, you can use the following option to override the default behavior of the checker: `--cs`

For example:

```
> <checker-name> --dir <intermediate_directory> --cs
```

### 2.2.1.15. PREFER\_TO\_ANALYZE\_JAVA

#### Synopsis

```
PREFER_TO_ANALYZE_JAVA()
```

## Description

This function makes the checker analyze Java-based output in the intermediate directory. By default, checkers otherwise analyze the C/C++ output.

Example:

```
START_EXTEND_CHECKER( javal, simple );

PREFER_TO_ANALYZE_JAVA();

ANALYZE_TREE()
{
    ...
}
```

The following error occurs if the intermediate directory does not contain Java output:

```
[ERROR] This program operates on Static Java
but specified intermediate directory <int-dir>
only contains data for C/C++.
```

### Note

If you have both Java and C/C++ output in your intermediate directory and want to use your checker on the C/C++, you can use the following option on the command line to override

```
PREFER_TO_ANALYZE_JAVA: --cpp
```

For example:

```
> <checker-name> --dir <intermediate_directory> --cpp
```

On the other hand, if you want to analyze Java output with a checker that does not call `PREFER_TO_ANALYZE_JAVA`, you can use the following option to override the default behavior of the checker: `--java`

For example:

```
> <checker-name> --dir <intermediate_directory> --java
```

## 2.2.1.16. PREFER\_TO\_ANALYZE\_JAVASCRIPT

### Synopsis

```
PREFER_TO_ANALYZE_JAVASCRIPT( )
```

### Description

This function makes the checker analyze JavaScript-based output in the intermediate directory. By default, checkers otherwise analyze the C/C++ output.

Example:

```
START_EXTEND_CHECKER( js1, simple );
```

```
PREFER_TO_ANALYZE_JAVASCRIPT();

ANALYZE_TREE()
{
    ...
}
```



**Note**

If you have both JavaScript and C/C++ output in your intermediate directory and want to use your checker on the C/C++, you can use the following option on the command line to override PREFER\_TO\_ANALYZE\_JAVASCRIPT: `--cpp`

For example:

```
> <checker-name> --dir <intermediate_directory> --cpp
```

On the other hand, if you want to analyze JavaScript output with a checker that does not call PREFER\_TO\_ANALYZE\_JAVASCRIPT, you can use the following option to override the default behavior of the checker: `--javascript`

For example:

```
> <checker-name> --dir <intermediate_directory> --javascript
```

### 2.2.1.17. ANALYZE\_CLASS

**Synopsis**

```
ANALYZE_CLASS() { <code> }
```

**Description**

This function can only be used for `type` [p. 50] checkers. It is called for each class that is defined in the source code. You can use the `CURRENT_CLASS` macro to get the class that is undergoing analysis. It has type `defined_class_type_t`.

---

## Chapter 2.3. Patterns

### Table of Contents

2.3.1. Patterns for C# and Java checkers .....	58
2.3.2. Functions common to all patterns .....	58
2.3.3. ASTNodePattern superclass .....	59
2.3.4. ExpressionPattern superclass .....	59
2.3.5. TypePattern superclass .....	60
2.3.6. SymbolPattern superclass .....	60
2.3.7. Predefined pattern objects .....	60
2.3.8. Expression patterns .....	60
2.3.9. Statement patterns .....	67
2.3.10. Other patterns .....	68

This introduction explains the pattern matching API. It does not replace the comments in the `><install_dir>/sdk/headers/patterns/*-patterns.hpp` header files, but provides a high level overview. You should refer to `*-patterns.hpp` as you read this section.

Refer to the sample pattern checker at `><install_dir>/sdk/samples/patterns/patterns.cpp`, which exercises all of the APIs described in this section.

Patterns are a mechanism for inspecting AST fragments (the AST, or Abstract Syntax Tree, is the internal representation of the code to analyze). The basic idea is to create a pattern object and then use its `match()` method to compare the pattern to a specific AST fragment. When `match()` returns true, each subpattern can be queried to obtain the AST fragment that it matched.

In addition to AST fragments (that represent code), patterns can match types and symbols. Types are C/C++ types such as structs, classes, or typedefs. Symbols are unique representatives for variables, functions, and class fields (as opposed to a specific appearance of them in the code). For instance, a variable expression (which is an AST fragment) references the symbol corresponding to the variable.

Correspondingly, there are three pattern hierarchies, with the corresponding C++ type hierarchy that they are used to match (argument type to `match()` is a rough equivalent). Note that only an `ASTNodePattern` (or a subclass) can be used directly with `MATCH` or `MATCH_TREE`. For `MATCH_COND`, you should use `ExpressionPattern`.

- `ASTNodePattern` (class `ASTNode`)
- `TypePattern` (class `type_t`)
- `SymbolPattern` (class `symbol_t`)

`ASTNodePattern` has two important sub-hierarchies for matching statements (such as a `for` loop) or expressions (such as an addition). Correspondingly, class `ASTNodePattern` has these subclasses:

- `StatementPattern` (class `Statement`)
- `ExpressionPattern` (class `Expression`)

Class `ASTNode` has a third subclass, `Declaration` (variable declaration), that can be matched by the `Decl` pattern. Since `Declaration` has no further subclasses, there is no hierarchy.

Next are diagrams of each of those hierarchies. In these diagrams triangles represent inheritance. Filled triangles indicate that the superclass is abstract (not instantiable), while the superclasses for the open triangles are concrete.

**Figure 2.3.1. ASTNodePattern class hierarchy**

**Figure 2.3.2. ExpressionPattern class hierarchy**

**Figure 2.3.3. StatementPattern class hierarchy**

**Figure 2.3.4. SymbolPattern class hierarchy**

## 2.3.1. Patterns for C# and Java checkers

The Coverity Extend SDK was originally used only to analyze C/C++ source code. Now that the Coverity Extend SDK also supports the creation of checkers that analyze C# and Java source code, you should note the following conventions when writing such checkers:

- To match the use of a C# struct or class or Java class, use the pattern `StructType`. See Section 2.3.5, “TypePattern superclass”.
- The terms *function* and *method* are sometimes used interchangeably in the Coverity Extend SDK header files and documentation. To match a C# or Java method call, use the pattern `CallSite`. See Section 2.3.8.2, “Function call site expression patterns”.
- To match the use of a C# or Java static field, use the pattern `StaticVar`. See Section 2.3.8.3, “Variable reference expression patterns”.
- To match the use of a C# or Java instance field, use the pattern `Component`. Keep in mind that matching a non-static class instance field involves an implicit dereference. In other words, the code `obj.field` involves a dereference of the object reference `obj`. For a code example, see `<install_directory>/sdk/samples/java_match_field`. For more information, see Section 2.3.8.1, “Basic expression patterns”.
- To match C# or Java references in an Coverity Extend SDK checker, you must use the `Pointer` pattern. The `Reference` pattern is only used for analyzing C/C++ code. Using the `Reference` pattern to analyze C# or Java code will not result in a match. See Section 2.3.8.4, “Type-filtered expression patterns”.

## 2.3.2. Functions common to all patterns

Every pattern superclass exposes a number of methods suitable for use in the Extend SDK. In the following, `T` is used to represent either `ASTNode`, `type_t`, or `symbol_t` (as appropriate).

- `bool match(const T *t)` — The primary matching function. Call it to attempt to match a data structure `t` with the pattern. If the match is a success, `match` returns `true`.
- `T last_<XXX>()` — Returns the last `T` that matched the pattern. Use only if the match succeeded. The value for `<XXX>` depends on which hierarchy you're using, and which level you're at in this hierarchy, and can be `astnode`, `type`, `symbol`, `expr`, or `stmt`. For compatibility with previous versions of the Extend SDK, `get_tree` is equivalent to `last_astnode`.
- `void print(ostream &out) const` — Prints a textual representation of this pattern. The text does not depend on whether the pattern has been used to match anything; it simply describes the structure of the pattern itself.
- `operator const T *()` — Same as `last_XXX()`; provided as a syntactic convenience.

Patterns can also be passed to `operator<<(ostream)`, in which case it will print the last matched `T`.

### 2.3.3. ASTNodePattern superclass

The `ASTNodePattern` is primarily used as a superclass for `StatementPattern` and `ExpressionPattern`. It is also used to inspect the tree hierarchy formed by all `ASTNodes` (for instance, an expression can be contained within a `for` loop). It also has a function, `recursive_match`, that returns a list of all the `ASTNodes` underneath (and including) the given one that matched the pattern.

### 2.3.4. ExpressionPattern superclass

`ExpressionPattern` matches expressions. By default, most `ExpressionPatterns` implicitly strip casts off of the expressions they match (exceptions to this are noted), since casts are often just noise from the point of view of program analysis; likewise for the return value of `last_expr()` and like functions. `ExpressionPattern` also has a few extra functions that are not available in other pattern hierarchies:

- `bool match(const Expression *e, bool polarity)` — Matches an expression that is negated if and only if `polarity` is `false`. For instance, `(a == b).match(e, false)` will match `a != b`. Typically used in `ANALYZE_CONDITION` and is used in the implementation of `MATCH_COND`.
- `match_with_casts` — Matches an expressions without first stripping casts. This does not affect cast stripping in subpatterns.
- `field()` — Returns a pattern that matches a field taken off an expression matched by this pattern. For instance, if `A` matches `foo`, then `A.field()` matches `foo.bar`. The method can also take a `SymbolPattern` argument to restrict the specific fields to match.
- `method()` — Same as `field()`, except that it matches a non-static method call.
- `get_type()` — Returns the type of the last matched expression.

Most C++ operators are also overloaded for class `ExpressionPattern`, that allows for the construction of patterns that match equivalent syntax. For instance, pattern `A + B` matches an addition.

### 2.3.5. TypePattern superclass

`TypePattern` matches types. They cannot be used directly in `MATCH` or `MATCH_TREE` (because these functions match AST nodes) but they can be used as parameters to other patterns, or used directly on the result of `get_type_of_tree`. By default, most `TypePatterns` remove typedefs and qualifiers before attempting to match (exceptions are noted). This means for instance, that a pattern `x_p` defined like this:

```
StructType x("X");
PointerType x_p(x);
```

matches all these types:

```
typedef struct X * X_p; X_p
struct X *
struct X const *
```

`TypePattern` also has a `match_with_typedefs_and_qualifiers` that is analogous to `ExpressionPattern::match_with_casts`.

### 2.3.6. SymbolPattern superclass

`SymbolPattern` matches symbols. Like the `TypePattern` class, `SymbolPattern` cannot be used directly in a `MATCH` or `MATCH_TREE`. Instead, use `SymbolPattern` as a parameter to other patterns. The most common `SymbolPattern` is `NamedSymbol`, which can be used to match a specific function or variable.

### 2.3.7. Predefined pattern objects

The Extend SDK has several pattern objects predefined for convenience.

- `anyASTNode`, `anyExpr`, `anyStmt`, `anyType`, `anySymbol` — Patterns that matches anything in the relevant pattern hierarchy.

 **Note**

Do not pass `any<XXX>` as a pattern and then try to extract the data that it last matched; they are global and used internally, and the last matched data could change at any time.

- `_` (underscore) — A pattern that matches anything in any pattern hierarchy. Because of potential ambiguity, it's preferable to use `any<XXX>` described previously.

### 2.3.8. Expression patterns

The most commonly used pattern classes are `ExpressionPatterns`.

`AnyExpression` (also `Expr`) is the most general pattern and it matches any expression. It doesn't strip casts.

### 2.3.8.1. Basic expression patterns

The basic expression patterns are those that correspond directly to primitive syntactic expression constructors. Here's a list of some of them.

- `BinOp (BinaryOp op, ExpressionPattern &a, ExpressionPattern &b)` — Match a binary operator, for example `a + b`. Typically you do not need to explicitly create a `BinOp`, because there are overloaded operators on `ExpressionPatterns` that will do so automatically. However, there are occasions where direct use is convenient. The possible values for `op` are listed in `><install_dir>/sdk/headers/ast/cc_flags.hpp`
- `SymBinOp(BinaryOp op, ExpressionPattern &a, ExpressionPattern &b)` — Match an operator that is symmetric, for example `a + b`. Any of the `BinOp` operators can be used, but only the symmetric ones make sense. This is useful when the `a` and `b` patterns are different, as `SymBinOp` matches both `(a,b)` and `(b,a)` orderings.
- `ASymBinOp(BinaryOp op, ExpressionPattern &, ExpressionPattern &b)` — Match an anti-symmetric operator or its dual. For example, `a < b` or `b > a`. Only anti-symmetric binops (inequalities) can be used.
- `Unop(UnaryOp op, ExpressionPattern &)` — Match a unary operator, for example `-a`. As with `BinOp`, direct use of `Unop` is only occasionally useful, since overloaded operators are provided that cover most of the common uses. The possible values for `op` are listed in `><install_dir>/sdk/headers/ast/cc_flags.hpp`
- `MapAccess(ExpressionPattern &map, ExpressionPattern &key)`

This expression pattern applies only to JavaScript.

Properties of other expressions will be represented by this. Also, global variables are represented by a `MapAccess` off a `GlobalVar` map. For example, the following will match all global variable expressions:

```
GlobalVar global;
MapAccess access(global, _);

if (MATCH(access)) { ... }
```

- `Star` — Match a dereference, for example `*p`. This will *not* match if `p` is an array (see below).
- `ArrayIndex` — Match an array element reference, i.e. `a[i]` where `a` is an array. This will *not* match if `a` is a pointer.
- `Assign` — Match an assignment or compound assignment, for example `a = b` or `a *= 2`.
- `Effect` — Match an increment or decrement, for example `++a` or `b--`.
- `CondPattern` — Match a use of the `?:` operator.
- `Const_int` — Match an integer literal.
- `Const_float` — Match a floating point literal.

- `Const_string` — Match a string literal.
- `Component` — Match a use of a field of an object, for example `a.b`.
- `Cast` — Match a cast. The flags control whether automatic (implicit) casts, manual (explicit) casts, or both are matched. This includes C++-style casts such as `reinterpret_cast` but not `dynamic_cast`.
- `StmtExpr` — Match a GNU statement expression, for example `( { x; y; z; } )`.
- `This` — Match a use, either explicit or implicit, of the `this` expression.
- `Addr` — Match an address-of expression, for example `&e`.
- `Throw` — Matches a `throw` expression.
- `NewPattern` — Matches a `new` expression.
- `DeletePattern` — Matches a `delete` expression.
- `DynamicCast` — Matches a `dynamic_cast` expression.

The two remaining primitive constructs are variable reference and function call, which are addressed in subsequent sections.

### 2.3.8.2. Function call site expression patterns

There are a variety of patterns to match function call sites:

- `CallSite` — Match function calls. This pattern is used to implement all the other ones below, which are only provided for backwards compatibility and convenience. A `CallSite` pattern can be set up to match function pointers, direct function calls or both (using `setCalledExpression`). It can also include or exclude method/non-method calls (using `setReceiverObject`). It also has an `operator()` method that allows specifying call arguments.

 **Note**

See bullet items below for uses of `CallSite` that replace deprecated `Fun` patterns.

- `Fun()` — Match a call to any function, including calls through function pointers. Equivalent to:

```
CallSite()
```

For backwards compatibility, note that using `operator()` with no arguments has no effect (but using `CallSite` with no arguments matches a call with no arguments).

 **Note**

This function is deprecated as of version 2020.12.

Use `CallSite` instead. For example:

- You can replace the following:

```
Fun f;
```

With the following:

```
CallSite f;
```

- `Fun(char const *name, fun_options flags = NONE)` — Match a call to a function, restricted as follows:
  - If the call is through a function pointer, match if and only if `flags` includes `FUNCTION_POINTERS_ALLOWED`.
  - If the call is to a named function (including class methods):
    - If `name` is `NULL`, then match.
    - Otherwise, if `flags` includes `UNMANGLE_NAME`, match if the unmangled name of the called function equals `name`.

Name mangling is a technique used by compilers to encode the type of an entity in its linker symbol name. For more information, see Section 2.8.2, “Name mangling”.

- Otherwise, match if the mangled name of the called function equals `name`.

 **Note**

This function is deprecated as of version 2020.12.

Use `CallSite` instead. For example:

- You can replace the following:

```
Fun f(name);
```

With the following:

```
CallSite f(name, /*unmangle*/false);
```

Note that `MATCH(f())` is equivalent to `MATCH(f)` when `f` is a `Fun`. However, if `f` is a `CallSite`, `MATCH(f())` will only match a call with no arguments, so Coverity recommends using `MATCH(f)` in this case.

- You can replace the following:

```
Fun f(NULL, Fun::FUNCTION_POINTERS_ALLOWED);
```

With the following:

```
CallSite f;
```

- You can replace the following:

```
Fun f(name, Fun::FUNCTION_POINTERS_ALLOWED);
```

With the following:

```
FunctionDecl fnDecl(name, /*unmangle*/false);
```

```
CallSite f(Or(fnDecl, *_));
```

- You can replace the following:

```
Fun f(NULL, Fun::FUNCTION_POINTERS_REQUIRED);
```

With the following:

```
CallSite f(*_);
```

- You can replace the following:

```
Fun f(NULL, Fun::FUNCTION_POINTERS_DISALLOWED);
```

With the following:

```
FunctionDecl fnDecl;
CallSite f(fnDecl);
```

- `Fun(MultipleNamesTag mntag, const char **fnames, fun_options flags = NONE)` — Similar to the previous pattern, except for a call to a named function, match if the name equals any of the strings in the NULL-terminated `fnames` array. If `fnames` begins with NULL, match regardless of the called function's name.

 **Note**

This function is deprecated as of version 2020.12.

Use `CallSite` instead. For example:

- You can replace the following:

```
Fun f(Fun::matchMultipleNames, names);
```

With the following:

```
CallSite f(namedSymbols(names));
```

- `MemberFun` — Match a call to a nonstatic member function, and provide patterns for the arguments. At a minimum, a pattern must be provided for the receiver object (the *instance*). Optionally, a sequence of argument patterns may be specified using `operator()`.

 **Note**

This function is deprecated as of version 2020.12.

Use `CallSite` instead. For example:

- You can replace the following:

```
MemberFun f(receiver);
```

With the following:

```
CallSite f;
f.setReceiverObject(receiver);
```

- `Constructor` — Match a call to a constructor.
- `CopyConstructor` — Match a call to a copy constructor, which is a special case of what `Constructor` matches.
- `Destructor` — Match a call to a destructor.
- `Anyfun` — Match a call to any function (there is no filtering based on function name). If you specify an argument pattern, the pattern matches if *any* argument at the call site matches the argument pattern. For example, `Anyfun()(Const_int)` matches any call where a literal int is among the arguments.

 **Note**

This function is deprecated as of version 2020.12.

Use `CallSite` instead. For example:

- You can replace the following:

```
Anyfun f;
MATCH(f(pat));
```

With the following:

```
CallSite f;
f.setAnyArg(pat);
```

### 2.3.8.3. Variable reference expression patterns

There are several patterns that match variable references, possibly taking into account scope and linkage:

- `Var` — Match a variable used in an expression.
- `TempVar` — Match a use of a temporary variable inserted by the parser.
- `LocalVar` — Match a use of a (nonstatic) local variable.
- `StaticVar` — Match a use of a static variable.
- `GlobalVar` — Match a use of a global variable.
- `Parm` — Match a use of a formal parameter.
- `FunctionDecl` — Match a use of a function as an expression (either the called expression in a function call, or taking the address of a function).
- `FunLocal` — Match a use of a local variable, formal parameter, or a field of (recursively) a class/struct/union-valued local or parameter. This corresponds to stack-allocated storage (except arrays).

### 2.3.8.4. Type-filtered expression patterns

Several patterns match expressions with certain types:

- `Array` — Match an expression with array type.
- `Const_obj` — Match an expression whose type has the `const` qualifier.
- `ExprWithType (TypePattern &)` — Matches an expression whose type is matched by the given `TypePattern`. For example,

```
Integer integer
```

is equivalent to:

```
IntegralType itype;
ExprWithType integer(itype);
```

- `Float` — Matches an expression with a floating point type (such as `float` or `double`). Same as `ExprWithType (FloatType)`.
- `FloatType` — Matches a floating point type (`float`, `double`, or `long double`). For example, you can pass this as a parameter to `ExprWithType`'s constructor.e.
- `FunctionPointer` — Match an expression with pointer to function type.
- `Integer` — Matches an expression with an integral type. Same as `ExprWithType (IntegralType)`.
- `IntegralType` — Matches an integral type, for instance `int`, `char` or `bool` but not `float` or `double`.
- `NonConstAddr` — Like `Addr` [p. 62] (match a use of the address-of operator), except it does not match if the object whose address is taken has the `const` type qualifier (for instance, when using a `const` reference function argument).
- `Pointer` — Match an expression with pointer or reference type.
- `Reference` — Match an expression with a C++ reference type.
- `Scalar` — Matches an expression with a scalar type. Same as `ExprWithType (ScalarType)`.
- `ScalarType` — Matches any scalar type (integral or floating point).
- `Struct` — Match an expression with `struct` or `class` type.
- `Union` — Match an expression with union type.

### 2.3.8.5. Complex expression patterns

Program analyses often need to detect certain kinds of more complex expression patterns. Several Extend SDK patterns do just this:

- `Arg` — Match an expression used as an argument to a function call. Accessors are provided to navigate to the call itself, and to see where in the call list the matched expression appeared. Will not strip casts.
- `ConditionPattern` — Match an expression used as a guard for a control flow statement or short-circuit operator.
- `Offset` — Match an expression that is the same as or an offset off of a given expression (pattern). For example, if you have an AST `p` that denotes a pointer, then `Offset(Same(p))` matches an expression like `&p->foo` which denotes a pointer to the same object that `p` does, but displaced by the offset of field `foo`. It will also recursively handle e.g. `&(p + 10)->foo`.
- `AnyField` — Given an expression (pattern), match an expression that is formed by appending field access operators. For example, given `a`, `AnyField(a)` matches `a.b` and `a.b.c`.
- `AnySubpart` — Like `AnyField`, except also allow array accesses, and (optionally) pointer dereferences.
- `Lval` — Match an *lvalue*, which is an expression that can appear on the left-hand side of an assignment operator. For example, `x` and `*p` are lvalues, whereas `3` and `a+b` are not (assuming those expressions use the built-in operators). A function call `f()` that returns a reference is translated by the parser into an explicit dereference `*f()` and the latter is matched by `Lval`.

### 2.3.8.6. Evaluation patterns

The evaluation patterns find the subexpression `s` of a (potentially) larger expression `e` that specifies the value yielded by `e`. These patterns are most easily explained in terms of the `Evals_to` function.

- `EvalPattern (Pattern &pat)` — Use `Evals_to` repeatedly to dig down into the matched expression, stopping as soon as `pat` matches a subexpression returned by `Evals_to`.
- `EvalsToPattern` — Use `Evals_to` to dig down to the smallest subexpression, then attempt to match `pat` against that subexpression.

### 2.3.9. Statement patterns

Each kind of statement has its own pattern to match it:

- `AnyStatement (also StmtPat)` — Match any statement.
- `DoWhilePat` — Match a `do` statement.
- `ExprStmt` — Match a statement containing a single expression, for example `a++;`.
- `IfPat` — Match an `if` statement.

Note that the guard expression for `if`, `for`, and `while` is normalized to a Boolean, and the match expression must agree with this normalized form. For example:

```
int f();
```

```
if (f()) { ... }
```

The test is normalized by the parser to:

```
if (f() != 0) { ... }
```

Hence, to match this, use:

```
CallSite call;
MATCH(IfPat(call != 0))
```

There is an example of this in `><install_dir>/sdk/samples/patterns/patterns.cpp`.

- `ForPat` — Match a `for` statement.
- `WhilePat` — Match a `while` statement. See `><install_dir>/sdk/samples/whileloopassign/whileloopassign.cpp` for some examples.
- `SwitchPat` — Match a `switch` statement.
- `ReturnPat` — Match a `return` statement. You can specify whether a value is returned. While there is always a `return` statement even when falling through the end of a function, you should use the `ANALYZE_END_OF_PATH` handler to respond to control flow that exits the function.
- `Try` — Match a `try/catch` statement.
- `AsmPat` — Match an `asm` statement.
- `LoopPat` — Matches `for`, `while` and `do` statements. This pattern does not match loops with a condition of constant 0, for example it won't match:

```
do {} while (0);
```

because this isn't a loop. You can use `DoWhilePat` to match this case.

Use:

```
ExprPat loop_cond;
LoopPat loop(loop_cond);
```

to match a loop and pull out the conditional expression. You can then check the conditional (for example is `a <= b`, or `a > b`) and act accordingly.

## 2.3.10. Other patterns

### 2.3.10.1. Binding patterns

The following classes are part of the implementation of `Same` and `PtrSame`, but they are not intended to be used directly.

- `Binding`

- `PatternBinding`
- `PtrBinding`
- `PtrBindingPattern`

### 2.3.10.2. Syntactic context filters

- `ContainsPat (Pattern &subpattern)` — Match a tree that has a subtree that matches `subpattern`.
- `InContextPat (Pattern &context)` — Match a tree that has a parent tree that matches `context`.
- `InStatementContextPat` — Similar to `InContextPat`, but only searches within the closest enclosing statement.
- `SubTreePat (tree t)` — Match any subtree of `t`.

### 2.3.10.3. STL construct patterns

- `STL_container` — Match an expression that denotes an instance of an STL container. Member functions are provided that match invocations of corresponding member functions on the container..
- `STL_iterator` — Match an expression that denotes an instance of an STL iterator. Member operators are provided that match applications of operators to the iterator.

### 2.3.10.4. Miscellaneous patterns

- `ExitScope` — Match a local variable going out of scope. This matches when exiting using a `return` statement, or any other way the flow goes out of a block (for example, `break` or `goto` statements).
- `DeadVariable` — Match a local variable becoming *dead*, which means that its value is not used again on the current path. The main purpose of this pattern is to optimize a checker's performance by removing useless mappings from the store.

### 2.3.10.5. Combinators (And, Or, etc.)

- `constant(int i)` — Equivalent to `Const_int(i)`.
- `assign(ExpressionPattern &a, ExpressionPattern &b)` — Equivalent to `Assign(a, b)`.
- `cast(ExpressionPattern &e)` — Equivalent to `Cast(e)`.
- `opt_cast(ExpressionPattern &e):` — Equivalent to `Or(e, Cast(e))`.
- `Const` — Build a `Const_obj` pattern.
- `And(Pattern &a, Pattern &b, ...)` — Match if *all* of the argument patterns match. Available for all pattern hierarchies.
- `Or(Pattern &b, Pattern &b, ...)` — Match if *any* of the argument patterns match. Available for all pattern hierarchies.
- `Not(Pattern &a)` — Match if *a* does *not* match.
- `Contains` — Build a `ContainsPat` pattern.

- `Within` — Build an `InContextPat` pattern.
- `WithinStatement` — Build an `InStatementContextPat` pattern.
- `Evals_to` — Look in an expression `e` to find the subexpression `s` that determines the value yielded by `e`. For example, `Evals_to(a = b)` returns `b`. The exact forms that are recognized are documented in the comments above the declaration in `<install_dir>/sdk/headers/patterns/extend-patterns.hpp`.

---

## Chapter 2.4. Accessors

### Table of Contents

2.4.1. Additional AST query functions .....	71
2.4.2. Queries on the current function .....	71
2.4.3. Queries on the current file .....	72
2.4.4. Queries on trees .....	72

### 2.4.1. Additional AST query functions

A number of additional AST query functions are declared in `<install_dir>/sdk/headers/extend/extend-lang.hpp`, which are described next.

The type safe Abstract Syntax Tree can be accessed from Coverity Extend SDK checkers. Details about the interface are located in `<install_dir>/sdk/headers/ast/cc.ast`.

### 2.4.2. Queries on the current function

Since the analysis works one function at a time, several functions return information about the current function being analyzed.

- `current_function_get_mangled_name` — Returns the full, mangled name of the current function.
- `current_function_get_name` — Returns the identifier for the current function, for example `foo`. This name is never mangled.
- `current_function_get_signature` — If the function's name is mangled (that is, a C++ function, but not mixed code using `extern "C"`), returns the result of demangling. This includes scope and parameter type information, for example `N::foo(int)`. Otherwise, this returns the identifier of the function, for example, `printf`.
- `current_function_get_class_name` — Returns the scope in which the current function is defined, for example `N`. For a function in the global scope, returns the empty string (`" "`).
- `current_function_is_ctor` — Returns true if the current function is a constructor.
- `current_function_is_dtor` — Returns true if the current function is a destructor.
- `current_function_is_pure_virtual` — Returns true if the current function is a pure virtual function. Note that a pure virtual function is defined as follows:

```
class MyClass {
  ...
  virtual void pureDefined() = 0; // combining pure and inline NOT allowed
};

void MyClass::pureDefined() { /* but this IS allowed */ }
```

- `current_function_is_virtual` — Returns true if the current function is a virtual function.

### 2.4.3. Queries on the current file

Additional queries return information about the file in which the current function is defined.

- `current_file_get_name` — Returns the name of the file in which the current function is defined.
- `current_file_lineno` — Returns the line number where the current AST fragment appears.

### 2.4.4. Queries on trees

These queries return information about the current tree.

- `is_tree_in_macro` — If the current tree is inside a macro, returns the macro's name. Otherwise, it returns NULL.
- `get_type_of_tree(tree t)` — Returns the `type_t` representing the type of `t`. If `t` does not have a type, returns NULL.
- `get_size_of_type(type_t *t)` — Returns the representation size in bytes for objects of type `t`.

---

## Chapter 2.5. The store

### Table of Contents

2.5.1. Store overview .....	73
2.5.2. void SET_STATE(tree t, int v) .....	73
2.5.3. void CLEAR_STATE(tree t) .....	74
2.5.4. bool GET_STATE(tree t, int &v) .....	74
2.5.5. bool MATCH_STATE(tree t, int v) .....	74
2.5.6. bool COPY_STATE(tree dst, tree src) .....	74
2.5.7. FOREACH_IN_STORE(tree &t, int &v) { body } .....	74
2.5.8. bool ADD_EVENT(tree t, char const *tag, desc) .....	74
2.5.9. bool COMMIT_ERROR(tree t, char const *tag, desc) .....	74
2.5.10. ADD_INPUTFILE_EVENT .....	75
2.5.11. COMMIT_INPUTFILE_ERROR .....	75

### 2.5.1. Store overview

The store is the primary data structure for a flow-sensitive checker (a flow-insensitive checker has no store). It is an approximation of a set of states that a real, running program might be in. As the checker walks over the program's abstract syntax tree (one function at a time), it simulates the program's behavior by changing the store in response to program operations such as assignments and function calls.

The store is a map from abstract syntax trees denoting expressions (*AST nodes*) to a pair consisting of an integer and an event sequence:

```
store : tree -> (int, event[])
```

The integer part of a mapping value is an *abstract value*. Whereas a real program (typically) has an infinite state space of *concrete values*, a checker reduces these down to a finite number of abstract values so that the checker can terminate. Much of the art of checker design is in choosing the set of abstract values. Although the examples in this manual are meant to suggest basic approaches, there are no hard and fast rules about abstract domain design, so some experimentation is required.

The event sequence part of the mapping value is for defect reporting purposes. It summarizes the sequence of abstract state transitions that have occurred for the mapped expression, so that a user viewing the report can understand what the checker did when it arrived at some conclusion. Note that an expression must be mapped to some abstract value before events can be attached to it.

The Extend SDK API provides several macros for querying and manipulating the store, as documented in the following sections.

For a detailed example of using the store, see Section 1.4.5, "Example: tracking the sign of expressions".

### 2.5.2. void SET\_STATE(tree t, int v)

If there is no mapping for  $t$ , creates a mapping, setting the integer component to  $v$  and the event sequence component to the empty sequence.

### 2.5.3. void CLEAR\_STATE(tree t)

Removes a mapping for `t` if one exists.

### 2.5.4. bool GET\_STATE(tree t, int &v)

If there is no mapping for `t`, returns false.

If there is a mapping for `t`, returns true, and sets `v` to equal the integer component of that mapping.

### 2.5.5. bool MATCH\_STATE(tree t, int v)

If there is a mapping for `t`, and the integer component of that mapping is equal to `v`, then returns true.

Otherwise, returns false.

### 2.5.6. bool COPY\_STATE(tree dst, tree src)

First, calls `CLEAR_STATE(dst)`.

Next, if there is no mapping for `src`, returns false.

Otherwise, creates a mapping for `dst`, sets its integer value and event sequence to equal those of `src`, and returns true.

### 2.5.7. FOREACH\_IN\_STORE(tree &t, int &v) { body }

For each mapping in the store, bind `t` to the tree and `v` to the integer component, then execute `body`. The body is syntactically in a `for` loop, so `break` and `continue` can be used to control the iteration.

The loop iterates over the mappings in an undefined order.

It is an error to modify the store during the iteration.

### 2.5.8. bool ADD\_EVENT(tree t, char const \*tag, desc)

If there is no mapping for `t`, returns false.

Otherwise, appends a new event to the event sequence to which `t` is mapped, and returns true.

The new event is constructed using `tag` and `desc`. The latter is evaluated as the right-hand argument to operator `<<(ostream&)` so you can construct complicated event strings, for example:

```
ADD_EVENT(t, "my_tag", "One plus " << 1 << " is " << (1+1));
```

See Chapter 2.6, *Adding events* for more information on tags.

### 2.5.9. bool COMMIT\_ERROR(tree t, char const \*tag, desc)

If there is no mapping for `t`, or `t` has no events, returns false.

If `tag` is neither `NULL` nor the empty string (" "), outputs the event sequence associated with `t`, *plus* one more event, constructed from `tag` and `event` as described in `ADD_EVENT`.

 **Note**

Due to these rules, you *must* create a mapping for a tree before you can add or output events. Further, you need to add at least one event (by using `ADD_EVENT`) before calling `COMMIT_ERROR`. In some circumstances, you need to invent a dummy value and/or event for this purpose.

## 2.5.10. `ADD_INPUTFILE_EVENT`

For a full description of this macro, see `ADD_INPUTFILE_EVENT` (`tree, /*extend_inputfile_t*/ f, line, tag, text_utf8`).

## 2.5.11. `COMMIT_INPUTFILE_ERROR`

For a full description of this macro, see `COMMIT_INPUTFILE_ERROR` (`tree, /*extend_inputfile_t*/ f, line, tag, text_utf8`).

---

## Chapter 2.6. Adding events

A defect report (error) from a checker consists of a nonempty sequence of events. An event is a pair consisting of two strings: a tag, and a description. A typical checker creates an event each time it updates its store, and a final event when it outputs a defect report. Event tags are short, single-word strings. Typically, a checker has one tag for each major kind of event that it creates.

You can suppress defect reports by adding source code annotations as described in the *Coverity 2020.12 Checker Reference* [☞](#). The event tag named is in the annotation.

Event descriptions are arbitrary strings that describe what is happening and what the defect is to the user.

The main way to create a defect report is to use `ADD_EVENT` and `COMMIT_ERROR`. However, using `OUTPUT_ERROR(desc)` outputs a single-event defect report immediately, bypassing the store entirely. Input file macros are also used for producing events and errors (see Chapter 1.9, *Reporting events and defects on input files*).

Outputting a defect does not necessarily mean it will ultimately be put into the final list of defects, due to two pass checking, which is explained in Section 1.7.3, “Two-pass checking”.

---

## Chapter 2.7. Types

### Table of Contents

2.7.1. Introduction .....	77
2.7.2. <code>type_t</code> .....	77
2.7.3. <code>any_type_t</code> .....	78
2.7.4. <code>scalar_type_t</code> .....	78
2.7.5. <code>pointer_type_t</code> .....	78
2.7.6. <code>array_type_t</code> .....	79
2.7.7. <code>cv_wrapper_type_t</code> .....	79
2.7.8. <code>function_type_t</code> .....	79
2.7.9. <code>scoped_type_t</code> .....	79
2.7.10. <code>scope_t</code> .....	79
2.7.11. <code>defined_type_t</code> .....	80
2.7.12. <code>typedef_type_t</code> .....	80
2.7.13. <code>forward_declarable_type_t</code> .....	80
2.7.14. <code>enum_type_t</code> .....	80
2.7.15. <code>tag_t</code> .....	80
2.7.16. <code>union_type_t</code> .....	80
2.7.17. <code>field_t</code> .....	81
2.7.18. <code>class_type_t</code> .....	81
2.7.19. <code>parent_t</code> .....	81
2.7.20. <code>function_t</code> .....	82
2.7.21. <code>member_type_t</code> .....	82
2.7.22. <code>extend_inputfile_t</code> .....	82

### 2.7.1. Introduction

As explained in Chapter 1.8, *Examining the class hierarchy*, types are represented using objects drawn from a class hierarchy rooted at `type_t`. Each of the classes and its fields is described in the following sections.

### 2.7.2. `type_t`

This class is the superclass of all type representation classes. It does not have any data members.

It has a number of virtual functions that can be used to determine which `type_t` subclass that an object is. This information is also available using standard run-time type identification, but these methods are sometimes more convenient. For each subclass `SUB`, there are methods:

- `const SUB &as_SUB() const` — If this object has dynamic type `SUB`, returns a reference to it as such. Otherwise, throws `invalid_type_exception`.
- `SUB &as_SUB()` — Same as previous, but accepts/returns a non-constant reference.
- `const SUB *as_SUB_p() const` — If this object has dynamic type `SUB`, returns a pointer to it as such. Otherwise, returns `NULL`.

- SUB `*as_SUB_p()` — Same as previous, but accepts/returns a non-constant pointer.

### 2.7.3. `any_type_t`

This is a placeholder type for JavaScript, since it is a dynamic language and detailed type information is not available. Most types in JavaScript will be treated as pointers to `any_type_t`. The exceptions to this are scalar types (e.g. integer literals) and pointers to the "char" scalar type (e.g. string literals). As such, the matching described for most types in this chapter is not applicable to JavaScript.

### 2.7.4. `scalar_type_t`

This class represents a fundamental type such as `int`. It has the following fields and methods.

- `string name` — Name of the type. Possible values are:
  - "void"
  - "bool"
  - "char"
  - "signed char"
  - "unsigned char"
  - "short"
  - "unsigned short"
  - "int"
  - "unsigned int"
  - "long"
  - "unsigned long"
  - "long long"
  - "unsigned long long"
  - "float"
  - "double"
  - "long double"
- `int size` — Size in bytes of objects of this type.
- `bool m_is_float` — True if this type is a floating-point type (one of `float`, `double` or `long double`).
- `bool m_is_signed` — True if this type is any of the floating-point types, or one of the signed integer types.

### 2.7.5. `pointer_type_t`

This class represents a pointer, reference, or pointer to member type.

- `type_t pointed_to` — The referent type. If it is a `member_type_t`, then this type represents a pointer to member.
- `bool m_is_ref` — True if this type is a reference, false if it is a pointer or pointer to member.

### 2.7.6. array\_type\_t

This class represents an array type.

- `type_t element_type` — The type of the elements.
- `int element_count` — Number of elements in the array. A value of -1 indicates that it is a variable length array.

### 2.7.7. cv\_wrapper\_type\_t

This class represents a variant of an underlying type but with `const` or `volatile` (or both) applied.

- `type_t target` — The type that is being wrapped with cv-qualifiers. Cannot be `cv_wrapper_type_t`.
- `v_flag_t flags` — Bitmap of cv-qualifiers being applied. Never 0.

### 2.7.8. function\_type\_t

This class represents a function type.

- `type_t args[]` — The sequence of argument types. For nonstatic methods, the receiver object (`this`) type is the first argument.
- `type_t return_type` — The return type of the function.
- `bool has_varargs` — If true, the function accepts a variable number of arguments. The `args` sequence has the required parameter types (those that precede "...").

### 2.7.9. scoped\_type\_t

This class is a superclass of types that the user can define.

- `string name` — The name of the type.
- `string unmangled_name` — The name of the type, mangled as defined by the IA64 C++ ABI.
- `scope_t scope` — The scope in which this type appears.

### 2.7.10. scope\_t

This class represents a named scope in the program being analyzed.

- `string name` — Name of the scope mangled as defined by the IA64 C++ ABI.
- `string unmangled_name` — Unmangled name of the scope.
- `scope_t parent` — Parent scope, or NULL.

### 2.7.11. `defined_type_t`

This class is the superclass of `typedef_type_t`.

- `string file` — The file in which the definition appears. If it is only declared, this is the file in which the declaration appears.
- `int line_no` — The line number of the previous declaration or definition.

### 2.7.12. `typedef_type_t`

This class represents a `typedef`.

- `type_t target` — The type for which this `typedef` is an alias.

### 2.7.13. `forward_declarable_type_t`

This class is a superclass of the types that can be forward-declared: class/union or enum only.

- `bool is_defined` — Indicates that the object is an `internal_defined_class_type_t` or `internal_defined_enum_type_t`, and the `as_defined` function can be called.
- `bool is_forward_declared` — Indicates that the type was forward-declared and the definition is not contained in the `forward_declarable_type_t` object itself. `is_forward_declared` is the negation of `is_defined`. To find out if a definition is available, use the `has_definition` function.
- `bool is_unnamed` — Indicates if the type was originally unnamed. The `get_name` function returns a generated name.
- `bool has_definition` — Indicates if there is a definition for this type.

### 2.7.14. `enum_type_t`

This class represents an enumeration type.

- `int size` — The size in bytes of an object of this enumeration's type.
- `tag_t values[]` — The sequence of enumerators in this enumeration.

### 2.7.15. `tag_t`

This class represents a single enumerator in an enumeration.

- `string name` — The name of the enumerator.
- `int value` — The value to which the enumerator is defined.

### 2.7.16. `union_type_t`

This class represents a union type.

- `int size` — The size in bytes of this union.
- `field_t fields[]` — The set of members of this union.

### 2.7.17. `field_t`

This class represents a member of a union.

- `string name` — The name of the member.
- `type_t type` — The type of the member.
- `bool is_bit_field` — Determine if the field is a bit field.
- `bool is_anonymous_bit_field` — Determine if the field is an anonymous bit field.
- `defined_class_type_t get_owner_class` — Retrieve the owner class as `defined_class_type_t`.
- `class_type_t *owner_class` — Retrieve the type as `member_type_t`.
- `unsigned get_offset` — Retrieve the offset, in bytes, from the beginning of the object.
- `unsigned char get_bit_offset` — If a bit field, this is the non-byte offset (0 otherwise). The bit field's bit offset is therefore `get_offset() * 8 + get_bit_offset()`.
- `bool is_signed_bitfield` — True if the type of a bitfield is explicitly signed, For example, `signed int`.
- `bool is_mutable` — Determine if this is a mutable field (mutable keyword).

### 2.7.18. `class_type_t`

This class represents a class or struct type.

- `bool m_is_struct` — True if this object represents a struct, false if it represents a class.
- `int size` — The size in bytes of this class.
- `parent_t parents[]` — The sequence of base classes.

### 2.7.19. `parent_t`

This class represents an immediate parent class of another class.

- `AccessKeyword access` — The accessibility of the base class.
- `bool is_virtual` — True if this is a virtual base class.
- `class_type_t get_class()` — A member function.

### 2.7.20. `function_t`

This class represents a function.

- `type_t *get_return_type` — Get the return type of this function.
- `function_type_t *get_ftype` — Get the function type.
- `bool is_virtual` — Determine if the method is virtual.
- `bool is_pure` — Determine if the method is pure virtual.
- `bool is_nonstatic_method` — Determine if the method is non-static.
- `bool is_static_method` — Determine if the method is static.
- `bool is_ctor` — Determine if the method is a constructor.
- `bool is_dtor` — Determine if the method is a destructor.

### 2.7.21. `member_type_t`

This class represents the type of a member of a class.

- `type_t member_type` — The type of this class member.
- `class_type_t c` — The class of this member.

### 2.7.22. `extend_inputfile_t`

For a description, see Chapter 1.9, *Reporting events and defects on input files*.

---

## Chapter 2.8. Reference information

### Table of Contents

2.8.1. Header files .....	83
2.8.2. Name mangling .....	83

The following sections contain header file and name mangling reference information.

### 2.8.1. Header files

The many header files in the `<install_dir>/sdk/headers` directory are necessary to compile an Coverity Extend SDK checker. However, only the following subset of these files declare or define functionality that can be used in an Coverity Extend SDK checker.

- `ast/astnode.hpp` - Defines the `ASTNode` class, the base class of abstract syntax tree (AST) node types, such as expressions and statements.
- `ast/cc.ast` - Defines the AST node types, representing statements, expressions, and so on. It serves as the source for the generated header `ast/cc.ast.hpp`.
- `ast/cc_flags.hpp` - Defines enumerations used by AST node types.
- `extend/extend-lang.hpp` - Defines the primary set of macros that an Coverity Extend SDK checker uses.

Note that `extend-checker-types` is not useful for end users.

- `symbols/field.hpp` - Defines the `field_t` type, a `symbol_t` representing a data member.
- `symbols/function.hpp` - Defines the `function_t` type, a `symbol_t` representing a function.
- `symbols/symbol.hpp` - Defines the `symbol_t` type, which is the base type for all the symbols (such as variables and functions).
- `symbols/variable.hpp` - Defines the `variable_t` type, a `symbol_t` representing a variable (global or local, including static data members).
- `types/extend-types.hpp` - Defines the `type_t` hierarchy, representing types.
- `types/scalar-types.hpp` - Defines enums and macros to go with the `scalar_type_t` type.

You can also use standard C/C++ headers.

### 2.8.2. Name mangling

Name mangling is a technique used by compilers to encode the type of an entity in its linker symbol name. This is necessary because of function overloading: `f(int)` and `f(int, int)` are distinct entities, so their linker symbol names must be distinct.

### 2.8.2.1. Mangled naming scheme: C++

For C++ name mangling, Coverity uses the IA64 C++ ABI [name mangling scheme](#), regardless of whether it is running on the IA64 platform or not. Table 2.8.1, “C++ Mangled names” provides the mangled names of two functions, as defined by that scheme.

**Table 2.8.1. C++ Mangled names**

Unmangled name	Mangled name
<code>f(int)</code>	<code>_Z1fi</code>
<code>f(int, int)</code>	<code>_Z1fii</code>

### 2.8.2.2. Mangled naming scheme: C#

Coverity uses a specialized grammar to represent the names and signatures of C# types, fields, methods, and so on. The rules for C# name mangling are as follows:

- Nested namespaces are separated by a period: `System.Data`
- Types directly contained by a namespace are separated by a period: `System.Math`
- Types represented by special keywords, such as `int` or `void`, are represented by their equivalent fully-qualified system types: `System.Int32`
- Array types are represented as in C#, by appending brackets containing zero or more commas: `System.Char[, ]`
- Nested types are separated by a solidus: `OuterClass/InnerClass`
- Unconstructed generic types and methods are represented by appending a backtick and the generic arity (the number of type parameters) to the mangled type or method name: `System.Collections.Generic.Dictionary`2`
- Constructed generic types are represented by appending to the mangled type name a comma-separated list of mangled type arguments in angle brackets: `System.Collections.Generic.Dictionary`2<System.Int32, System.String>`
- Type members other than nested types are separated from their containing type by a double colon: `MyClass::myField`
- Method names are followed by first, a parenthesized, comma-separated list of mangled formal parameter types (ote that the comma is not followed by a space), and second, the mangled return type: `System.Math::Sin(System.Double)System.Double`

## Reference information

---

- Class constructors, instance constructors and destructors are treated as void-returning methods named `.ctor`, `.ctor`, or `.dctor` respectively: `System.Object::ctor()System.Void`
- Property accessors are represented as though they were methods; the property name is prepended with `get_` or `set_` as appropriate: `System.String::get_Chars()System.Char[]`

Similarly, event accessors are represented as though they were methods; the property name is prepended with `add_` and `remove_` as appropriate.

- Indexers are treated as though they were methods; the indexer name is `get_Item` or `set_Item` as appropriate: `System.String::get_Item(System.Int32)System.Char`
- User-defined operators are treated as though they were methods named `op_Addition`, `op_LogicalNot`. The full list is as follows:

<code>!=</code>	<code>op_Inequality</code>
<code>&lt;</code>	<code>op_LessThan</code>
<code>&gt;</code>	<code>op_GreaterThan</code>
<code>&lt;=</code>	<code>op_LessThanOrEqual</code>
<code>&gt;=</code>	<code>op_GreaterThanOrEqual</code>
<code>*</code>	<code>op_Multiply</code>
<code>/</code>	<code>op_Division</code>
<code>%</code>	<code>op_Modulus</code>
<code>+</code>	<code>op_Addition / op_UnaryPlus</code>
<code>-</code>	<code>op_Subtraction / op_UnaryNegation</code>
<code>&lt;&lt;</code>	<code>op_LeftShift</code>
<code>&gt;&gt;</code>	<code>op_RightShift</code>
<code>&amp;</code>	<code>op_BitwiseAnd</code>
<code> </code>	<code>op_BitwiseOr</code>
<code>^</code>	<code>op_ExclusiveOr</code>
<code>!</code>	<code>op_LogicalNot</code>
<code>~</code>	<code>op_OnesComplement</code>
<code>++</code>	<code>op_Increment</code>
<code>--</code>	<code>op_Decrement</code>
<code>operator true</code>	<code>op_True</code>
<code>operator false</code>	<code>op_False</code>
<code>implicit operator &lt;type&gt;</code>	<code>op_Implicit</code>
<code>explicit operator &lt;type&gt;</code>	<code>op_Explicit</code>

- Formal type parameters are represented using the name of the parameter.

- Unsafe pointer types have a \* appended to the mangled name.

For example: the mangled name of the method in:

```
namespace N
{
  class O<T>
  {
    class I<U, V>
    {
      void M<W>(L<W> w, int[] i) {}
    }
  }
  class L<X> {}
}
```

Would be:

```
N.O`1/I`2::M`1(N.L`1<W>,System.Int32[])System.Void
```

### 2.8.2.3. Mangled naming scheme: Java

Coverity uses a specialized grammar to mangle the names of Java identifiers (*Identifier*), which include the names of packages, classes, fields, methods, and so on. Table 2.8.2, “Java Mangled name grammar” provides synopses of that grammar, and the code sample [p. 87 ] that follows it provides the mangled name for several identifiers in the sample.

**Table 2.8.2. Java Mangled name grammar**

Identifier type	Synopsis <sup>a</sup>
ClassName	{ Identifier "\$" } Identifier
MethodName	Identifier
FieldName	Identifier
PackageName	Identifier { "." Identifier }
MangledClassName	[ PackageName "." ] ClassName { "[" ] }
MangledFieldName	MangledClassName "." FieldName
PrimitiveTypeName	"boolean"   "char"   "float"   "double"   "byte"   "short"   "int"   "long"
TypeName	MangledClassName   PrimitiveTypeName { "[" ] }
ReturnTypeName	TypeName   "void"
ArgList	TypeName { ", " TypeName }
MangledMethodName	MangledClassName "." MethodName "(" [ ArgList ] ")" ReturnTypeName

<sup>a</sup>The synopses use the following syntax:

- Brackets for a sequence that is optional: [ ]

## Reference information

---

- Curly braces for a sequence that can be omitted or repeated: {}
- Pipes for a sequence that must match exactly one of the options: "x" | "y" | "z"

The comments in the following code sample list the mangled names of classes, fields, methods, and other items in a sample Java class.

```
// (c) 2017, Synopsys Inc. All rights reserved worldwide.
package p;

class Outer {                                // p.Outer1

    int i = 24;                               // p.Outer.i2
    String str = "s";                         // p.Outer.str

    // The following "virtual" function is created to
    // initialize all non-static member fields:
    //   p.Outer.<instance_field_initializations>()void3

    static final int a = 1; // p.Outer.a
    static int b = 2;      // p.Outer.b
    static {
        b = 777;
    }

    // The following "virtual" function is created to do
    // all the static initializations for this class:
    //   p.Outer.<clinit>()void

    Outer() {                                // p.Outer.<init>()void
        int localVar = 4;                    // localVar4
    }

    int foo(int i, int j) {                  // p.Outer.foo(int, int)int
        return 21;
    }

    void printme(String s) {                // p.Outer.printme(java.lang.String)void5
        System.out.println(s);             // java.io.PrintStream.println(java.lang.String)void
    }
}
```

<sup>1</sup>If a class is declared within a package, then its mangled class name is always prepended with its package name.

Examples: `java.lang.String`, `Package.ClassA`

<sup>2</sup>Class field names are always prepended with the full class name and the package name, if any.

Example: `Package.ClassA.field`

<sup>3</sup>A few special methods are created automatically (if applicable) for each class:

- `<init>` - A class constructor.
- `<clinit>` - A method containing all the static initializers for a class.
- `<instance_field_initializations>` - A method containing the initializers for non-static member fields.

<sup>4</sup>Local variable names do not require any mangling. The package and class names are not prepended for local variables.

<sup>5</sup>Unless a type is one of the primitive types (such as `boolean`, `char`, or `int`), its mangled type name includes the package it belongs to. Example: `java.lang.String`

## Reference information

```
class Inner { // p.Outer$Inner6
    int field = 1; // p.Outer$Inner.field
    int bar(int i, int j) { // p.Outer$Inner.bar(int, int)int78
        return 22;
    }
}

void testAnonClass(String s) { // p.Outer.testAnonClass(java.lang.String)void
    final String capturedLocal = s; // capturedLocal

    Inner anonymousInstance =
        new Inner() { // p.Outer$19

            // The constructor that is generated for an anonymous class
            // takes the containing class instance as an argument:
            // p.Outer$1.<init>(p.Outer)void
            //
            // In addition, a synthetic field called "this$0" is
            // created to refer to the containing class instance.
            // And synthetic fields called val$<var_name> are created
            // for any captured local variables from the containing class.

            public String getS() { // p.Outer$1.getS()java.lang.String
                Outer myParent = // myParent
                    Outer.this; // this$0
                return capturedLocal; // val$apturedLocal10
            }
        };
}
```

### Note

Also note that Java class constructors have different mangled names than C++ constructors. For example, compare the C++ name `CClass::CClass()` to the Java name `JavaClass.<init>`.

<sup>6</sup>Nested class names are denoted with \$. Example: `Package.Outer$Inner1$Inner2`

<sup>7</sup>The return type for a method is located at the end of the mangled name, rather than at the beginning. Example: `Package.ClassA.printString(java.lang.String)void`. Otherwise, mangled method names are very readable (especially compared to mangled C functions).

<sup>8</sup>When matching the mangled name of a method, be careful to notice the space between arguments. For example, attempting to match the name `foo(int, int)void` will not result in a match. You must use the following, instead: `foo(int, int)void`

<sup>9</sup>Anonymous classes have no real name, so they are assigned numbers. The numbers are assigned arbitrarily and you should not depend on their appearance in a particular order. For the purpose of name mangling, anonymous classes are treated as nested classes. Example: `Package.Outer$1`

<sup>10</sup>There are two kinds of synthetically-created variables, and they are both related to the way that an anonymous class captures values from its enclosing class or method.

- `this$0` - Pointer to the enclosing class instance.
- `val$<original_variable_name>` - Captured local variables. When a final local variable is used inside an anonymous class, this name is used to refer to it.

---

## Part 3. Checker Examples

### Table of Contents

3.1. Checker source files .....	90
3.1.1. sign checker .....	90
3.1.2. sign2 checker .....	94
3.1.3. sign3 checker .....	98
3.1.4. print_types.cpp .....	107
3.1.5. switch_default.cpp .....	109
3.1.6. javascript_match_local.cpp .....	110

---

## Chapter 3.1. Checker source files

### Table of Contents

3.1.1. sign checker .....	90
3.1.2. sign2 checker .....	94
3.1.3. sign3 checker .....	98
3.1.4. print_types.cpp .....	107
3.1.5. switch_default.cpp .....	109
3.1.6. javascript_match_local.cpp .....	110

Some of the sample checkers that were too long to include in the previous sections are reproduced here.



#### Note

The complete set of source code and makefiles are available in the `<install_dir>/sdk/samples` directory.

### 3.1.1. sign checker

```
// keep track of the sign of each expression

#include "extend-lang.hpp"      // Extend SDK API

// ----- utilities -----
// skip past pathname component of a file name
char const *strip_path(char const *fname)
{
    // find last slash; don't want to rely on strrchr being present
    for (char const *p = fname; *p; p++) {
        if (*p == '/') {
            fname = p+1;      // go one past this (maybe last) slash
        }
    }
    return fname;
}

// print out the current file/line (stripping the path of the file),
// and return an ostream for additional printing
ostream &cout_loc()
{
    return cout << strip_path(current_file_get_name()) << ":"
        << current_file_lineno() << ": ";
}

// ----- AbsValue -----
// abstract value domain
enum AbsValue {
    AV_NEGATIVE,      // < 0
    AV_NEG_ZERO,      // <= 0
    AV_ZERO,          // 0
}
```

## Checker source files

```
AV_POS_ZERO,      // >= 0
AV_POSITIVE,     // > 0
AV_UNKNOWN       // unknown; only for return value from abstract
                // arithmetic, not to be put into store
};

// confirm the int is in the right range for an AbsValue
void bcAbsValue(int i)
{
    assert((unsigned)i < AV_UNKNOWN);
}

// map from int to AbsValue; this is necessary because the store
// stores ints, not AbsValues, as its declared type
AbsValue toAbsValue(int i)
{
    bcAbsValue(i);
    return (AbsValue)i;
}

// print an abstract value
ostream& operator<< (ostream& os, AbsValue v)
{
    switch (v) {
        default: assert(!"bad AbsValue code");
        case AV_NEGATIVE: return os << "AV_NEGATIVE";
        case AV_NEG_ZERO: return os << "AV_NEG_ZERO";
        case AV_ZERO:     return os << "AV_ZERO";
        case AV_POS_ZERO: return os << "AV_POS_ZERO";
        case AV_POSITIVE: return os << "AV_POSITIVE";
    }
}

// ----- abstract operations -----
// abstract addition; assumes overflow can't happen
AbsValue abstractAdd(AbsValue a, AbsValue b)
{
    static AbsValue const map[5][5] = {
        // b:   a:  <0      <=0      0          >=0      >0
        /* <0 */ { AV_NEGATIVE, AV_NEGATIVE, AV_NEGATIVE, AV_UNKNOWN, AV_UNKNOWN },
        /* <=0 */ { AV_NEGATIVE, AV_NEG_ZERO, AV_NEG_ZERO, AV_UNKNOWN, AV_UNKNOWN },
        /* 0 */ { AV_NEGATIVE, AV_NEG_ZERO, AV_ZERO, AV_POS_ZERO, AV_POSITIVE },
        /* >=0 */ { AV_UNKNOWN, AV_UNKNOWN, AV_POS_ZERO, AV_POS_ZERO, AV_POSITIVE },
        /* >0 */ { AV_UNKNOWN, AV_UNKNOWN, AV_POSITIVE, AV_POSITIVE, AV_POSITIVE },
    };

    bcAbsValue(a);
    bcAbsValue(b);
    return map[a][b];
}

AbsValue abstractSub(AbsValue a, AbsValue b)
{
```

```

// just invert the sign of 'b' and add
bcAbsValue(b);
return abstractAdd(a, toAbsValue(AV_POSITIVE - b));
}

// ----- the checker -----
// This store maps expressions to AbsValue; unmapped expressions
// have unknown sign.
START_EXTEND_CHECKER( sign, int_store );

ANALYZE_TREE()
{
// integer literal?
Const_int ci;
if (MATCH(ci)) {
    if (ci.llval() < 0) {
        SET_STATE(CURRENT_TREE, AV_NEGATIVE);
    }
    else if (ci.llval() == 0) {
        SET_STATE(CURRENT_TREE, AV_ZERO);
    }
    else {
        SET_STATE(CURRENT_TREE, AV_POSITIVE);
    }
    return;
}

// unsigned variable?
Scalar scal;
Var var;
if (MATCH(var) && MATCH(scal) && scal.get_type()->is_unsigned()) {
    int v;
    if (GET_STATE(CURRENT_TREE, v) && v == AV_POSITIVE) {
        // 'var' is already known to be positive, so leave it alone
    }
    else {
        // set it to >= 0
        SET_STATE(CURRENT_TREE, AV_POS_ZERO);
    }
}

// arithmetic?
Expr a,b;
if (MATCH(a+b)) {
    int va, vb;
    if (GET_STATE(a, va) && GET_STATE(b, vb)) {
        AbsValue v = abstractAdd(toAbsValue(va), toAbsValue(vb));
        if (v != AV_UNKNOWN) {
            SET_STATE(CURRENT_TREE, v);
        }
    }
}
return;
}

```

```

}
if (MATCH(a-b)) {
  int va, vb;
  if (GET_STATE(a, va) && GET_STATE(b, vb)) {
    AbsValue v = abstractSub(toAbsValue(va), toAbsValue(vb));
    if (v != AV_UNKNOWN) {
      SET_STATE(CURRENT_TREE, v);
    }
  }
  return;
}

// assignment?
if (MATCH(a = b)) {
  COPY_STATE(a, b);
  return;
}

// query for abstract value?
if (MATCH(CallSite("whatis")(a))) {
  int val;
  if (GET_STATE(a, val)) {
    cout_loc() << a << " has value " << toAbsValue(val) << endl;
  }
  else {
    cout_loc() << a << " has unknown value" << endl;
  }
  return;
}

// print entire store?
if (MATCH(CallSite("print_store"))) {
  cout_loc() << "print_store:\n";

  int mappings = 0;
  const ASTNode* t;
  int v;
  FOREACH_IN_STORE(t, v) {
    cout << " " << t << " has value " << toAbsValue(v) << endl;
    mappings++;
  }

  cout << " " << mappings << " mappings" << endl;
  return;
}
}

END_EXTEND_CHECKER();

MAKE_MAIN( sign )

```

### 3.1.2. sign2 checker

```

// keep track of the sign of each expression

// Extended from sign.c: Use Extend SDK API output routines.

#include "extend-lang.hpp"      // Extend SDK API

using std::ostringstream;

// ----- utilities -----
// skip past pathname component of a file name
char const *strip_path(char const *fname)
{
    // find last slash; don't want to rely on strrchr being present
    for (char const *p = fname; *p; p++) {
        if (p[0] == '/' && p[1] != '\0') {
            fname = p+1;    // go one past this (maybe last) slash
        }
    }
    return fname;
}

// print out the current file/line (stripping the path of the file),
// and return an ostream for additional printing
ostream &cout_loc()
{
    return cout << strip_path(current_file_get_name()) << ":"
        << current_file_lineno() << ": ";
}

// ----- AbsValue -----
// abstract value domain
enum AbsValue {
    AV_NEGATIVE,      // < 0
    AV_NEG_ZERO,     // <= 0
    AV_ZERO,         // 0
    AV_POS_ZERO,     // >= 0
    AV_POSITIVE,     // > 0
    AV_UNKNOWN       // unknown; only for return value from abstract
                    // arithmetic, not to be put into store
};

// confirm (bounds check) that the int is in the right range
void bcAbsValue(int i)
{
    assert((unsigned)i < AV_UNKNOWN);
}

// map from int to AbsValue; this is necessary because the store
// stores ints, not AbsValues, as its declared type
AbsValue toAbsValue(int i)
{

```

```

bcAbsValue(i);
return (AbsValue)i;
}

// print an abstract value
ostream& operator<< (ostream &os, AbsValue v)
{
    switch (v) {
        default: assert(!"bad AbsValue code");
        case AV_NEGATIVE: return os << "AV_NEGATIVE";
        case AV_NEG_ZERO: return os << "AV_NEG_ZERO";
        case AV_ZERO:      return os << "AV_ZERO";
        case AV_POS_ZERO: return os << "AV_POS_ZERO";
        case AV_POSITIVE: return os << "AV_POSITIVE";
    }
}

// ----- abstract operations -----
// abstract addition; assumes overflow can't happen
AbsValue abstractAdd(AbsValue a, AbsValue b)
{
    static AbsValue const map[5][5] = {
        // b:   a:  <0      <=0      0      >=0      >0
        /* <0 */ { AV_NEGATIVE, AV_NEGATIVE, AV_NEGATIVE, AV_UNKNOWN, AV_UNKNOWN },
        /* <=0 */ { AV_NEGATIVE, AV_NEG_ZERO, AV_NEG_ZERO, AV_UNKNOWN, AV_UNKNOWN },
        /* 0 */ { AV_NEGATIVE, AV_NEG_ZERO, AV_ZERO, AV_POS_ZERO, AV_POSITIVE },
        /* >=0 */ { AV_UNKNOWN, AV_UNKNOWN, AV_POS_ZERO, AV_POS_ZERO, AV_POSITIVE },
        /* >0 */ { AV_UNKNOWN, AV_UNKNOWN, AV_POSITIVE, AV_POSITIVE, AV_POSITIVE },
    };

    bcAbsValue(a);
    bcAbsValue(b);
    return map[a][b];
}

// abstract subtraction
AbsValue abstractSub(AbsValue a, AbsValue b)
{
    // just invert the sign of 'b' and add
    bcAbsValue(b);
    return abstractAdd(a, toAbsValue(AV_POSITIVE - b));
}

// ----- the checker -----
// This store maps expressions to AbsValue; unmapped expressions
// have unknown sign.
START_EXTEND_CHECKER( sign2, int_store );

ANALYZE_TREE()
{
    // integer literal?
    Const_int ci;
    if (MATCH(ci)) {

```

```

if (ci.llval() < 0) {
    SET_STATE(CURRENT_TREE, AV_NEGATIVE);
}
else if (ci.llval() == 0) {
    SET_STATE(CURRENT_TREE, AV_ZERO);
}
else {
    SET_STATE(CURRENT_TREE, AV_POSITIVE);
}
ADD_EVENT(CURRENT_TREE, "literal", "Saw literal value: " << ci.llval());
return;
}

// unsigned variable?
Scalar scal;
Var var;
if (MATCH(var) && MATCH(scal) && scal.get_type()->is_unsigned()) {
    int v;
    if (GET_STATE(CURRENT_TREE, v) && v == AV_POSITIVE) {
        // 'var' is already known to be positive, so leave it alone
    }
    else {
        // set it to >= 0
        CLEAR_STATE(CURRENT_TREE); // avoid lots of 'unsigned' events
        SET_STATE(CURRENT_TREE, AV_POS_ZERO);
        ADD_EVENT(CURRENT_TREE, "unsigned", "Variable is unsigned");
    }
}

// arithmetic?
Scalar a,b;
if (MATCH(a+b)) {
    // any prior info we might have had regarding "a+b" is irrelevant
    CLEAR_STATE(CURRENT_TREE);

    int va, vb;
    if (GET_STATE(a, va) && GET_STATE(b, vb)) {
        AbsValue v = abstractAdd(toAbsValue(va), toAbsValue(vb));
        if (v != AV_UNKNOWN) {
            // at this time, there is no way to copy the events from two
            // different sources, so just get what I can ... bug 3439
            COPY_STATE(CURRENT_TREE, a);

            SET_STATE(CURRENT_TREE, v);
            ADD_EVENT(CURRENT_TREE, "addition",
                "Addition: " << a << " (" << toAbsValue(va) <<
                ") plus " << b << " (" << toAbsValue(vb) <<
                ") yields " << v);
        }
    }
}
return;
}

```

```

if (MATCH(a-b)) {
    CLEAR_STATE(CURRENT_TREE);

    int va, vb;
    if (GET_STATE(a, va) && GET_STATE(b, vb)) {
        AbsValue v = abstractSub(toAbsValue(va), toAbsValue(vb));
        if (v != AV_UNKNOWN) {
            COPY_STATE(CURRENT_TREE, a);
            SET_STATE(CURRENT_TREE, v);
            ADD_EVENT(CURRENT_TREE, "subtraction",
                "Subtraction: " << a << " (" << toAbsValue(va) <<
                ") minus " << b << " (" << toAbsValue(vb) <<
                ") yields " << v);
        }
    }
    return;
}

// assignment?
if (MATCH(a = b)) {
    COPY_STATE(a, b);
    ADD_EVENT(a, "var_assign",
        "Assigning " << a << " to value of " << b);
    return;
}

// possible conversion error?
IntegralType destType;
Cast cast(a, destType); // cast from expression 'a' to type 'destType'
if (MATCH(cast) && !a.get_type()->is_unsigned() && destType.is_unsigned()) {
    int v;
    if (GET_STATE(a, v)) {
        if (v == AV_NEGATIVE) {
            COMMIT_ERROR(a, "conversion_error",
                a << " is converted to 'unsigned' but is known to be negative");
        }
        else if (v == AV_NEG_ZERO) {
            COMMIT_ERROR(a, "conversion_error",
                a << " is converted to 'unsigned' but may be negative");
        }
        else {
            // we know it is *not* negative, so the cast is safe
        }
    }
    else {
        OUTPUT_ERROR(a << " is converted to 'unsigned' but may be negative");
    }
}

// query for abstract value?
if (MATCH(CallSite("whatis")(a))) {
    int val;
    if (GET_STATE(a, val)) {

```

```

    COMMIT_ERROR(a, "whatis", a << " has value " << toAbsValue(val));
}
else {
    // here, COMMIT_ERROR would do nothing
    OUTPUT_ERROR("whatis: " << a << " has unknown value");
}
return;
}

// print entire store?
if (MATCH(CallSite("print_store"))) {
    ostringstream os;
    os << "print_store: ";

    int mappings = 0;
    const ASTNode* t;
    int v;
    FOREACH_IN_STORE(t, v) {
        if (mappings > 0) {
            os << ", ";
        }
        os << t << " has value " << toAbsValue(v);
        mappings++;
    }

    os << "; " << mappings << " mappings";

    OUTPUT_ERROR(os.str());
    return;
}
}

END_EXTEND_CHECKER();

MAKE_MAIN( sign2 )

```

### 3.1.3. sign3 checker

```

// keep track of the sign of each expression

// Extended from sign.c: Use Extend SDK API output routines.

#include "extend-lang.hpp" // Extend SDK API

#if 1
# define DIAGNOSTIC(stuff) cout << stuff << endl /* user ; */
#else
# define DIAGNOSTIC(stuff) ((void)0) /* user ; */
#endif

// ----- AbsValue -----
// abstract value domain
enum AbsValue {

```

```

AV_NEGATIVE,      // < 0
AV_NEG_ZERO,     // <= 0
AV_ZERO,         // 0
AV_POS_ZERO,     // >= 0
AV_POSITIVE,     // > 0
AV_UNKNOWN,      // unknown; only for return value from abstract
                // arithmetic, not to be put into store
NUM_ABSVALS
};

#define FOREACH_ABSVAL(var) \
    for(AbsValue var = AV_NEGATIVE; var < NUM_ABSVALS; \
        var = (AbsValue)(var+1))

// confirm (bounds check) that the int is in the right range
void bcAbsValue(int i)
{
    assert((unsigned)i < AV_UNKNOWN);
}

// this one allows the AV_UNKNOWN value
void bcAbsValueU(int i)
{
    assert((unsigned)i < NUM_ABSVALS);
}

// map from integer code to AbsValue; this is necessary because the
// store stores ints, not AbsValues, as its declared type
AbsValue toAbsValue(int i)
{
    bcAbsValueU(i);
    return (AbsValue)i;
}

// map a integer value to abstract value
AbsValue abstractSingleValue(long long v)
{
    if (v < 0) {
        return AV_NEGATIVE;
    }
    else if (v == 0) {
        return AV_ZERO;
    }
    else {
        return AV_POSITIVE;
    }
}

// print an abstract value
ostream& operator<< (ostream &os, AbsValue v)
{
    switch (v) {
        default: assert(!"bad AbsValue code");
    }
}

```

## Checker source files

```
case AV_NEGATIVE: return os << "AV_NEGATIVE";
case AV_NEG_ZERO: return os << "AV_NEG_ZERO";
case AV_ZERO:     return os << "AV_ZERO";
case AV_POS_ZERO: return os << "AV_POS_ZERO";
case AV_POSITIVE: return os << "AV_POSITIVE";
case AV_UNKNOWN:  return os << "AV_UNKNOWN";
}
}

// ----- abstract operations -----
// abstract addition; assumes overflow can't happen
AbsValue abstractAdd(AbsValue a, AbsValue b)
{
    static AbsValue const map[AV_UNKNOWN][AV_UNKNOWN] = {
        // b:   a:   <0      <=0      0      >=0      >0
        /* <0 */ { AV_NEGATIVE, AV_NEGATIVE, AV_NEGATIVE, AV_UNKNOWN, AV_UNKNOWN },
        /* <=0 */ { AV_NEGATIVE, AV_NEG_ZERO, AV_NEG_ZERO, AV_UNKNOWN, AV_UNKNOWN },
        /* 0 */   { AV_NEGATIVE, AV_NEG_ZERO, AV_ZERO,     AV_POS_ZERO, AV_POSITIVE },
        /* >=0 */ { AV_UNKNOWN,  AV_UNKNOWN,  AV_POS_ZERO, AV_POS_ZERO, AV_POSITIVE },
        /* >0 */ { AV_UNKNOWN,  AV_UNKNOWN,  AV_POSITIVE, AV_POSITIVE, AV_POSITIVE },
    };

    bcAbsValue(a);
    bcAbsValue(b);
    return map[a][b];
}

// abstract subtraction
AbsValue abstractSub(AbsValue a, AbsValue b)
{
    // just invert the sign of 'b' and add
    bcAbsValue(b);
    return abstractAdd(a, toAbsValue(AV_POSITIVE - b));
}

// ----- abstract comparison -----
// Record the consequences of learning that the comparison 'a' op 'b'
// is true.
struct AbstractComparisonResult {
    // if this is false, we know the comparison could *not* be true,
    // so we will abort the path
    bool consistent;

    // what is the new approximation for 'a' and 'b'?
    AbsValue newAValue;
    AbsValue newBValue;
};

// the type of a precomputed relation map; maps from the abstract
// values of its arguments to the abstract comparison result
typedef AbstractComparisonResult RelationMap[NUM_ABSVALS][NUM_ABSVALS];

// concrete relational operator
```

## Checker source files

```
typedef bool (*ConcreteOperator)(int a, int b);

// compute best approximation of union of concrete values
// represented by 'a' and 'b'
AbsValue greatestLowerBound(AbsValue a, AbsValue b)
{
    if (a == AV_UNKNOWN || b == AV_UNKNOWN) {
        return AV_UNKNOWN;
    }

    static AbsValue const map[AV_UNKNOWN][AV_UNKNOWN] = {
        // b:   a:  <0      <=0      0      >=0      >0
        /* <0 */ { AV_NEGATIVE, AV_NEG_ZERO, AV_NEG_ZERO, AV_UNKNOWN, AV_UNKNOWN },
        /* <=0 */ { AV_NEG_ZERO, AV_NEG_ZERO, AV_NEG_ZERO, AV_UNKNOWN, AV_UNKNOWN },
        /* 0 */ { AV_NEG_ZERO, AV_NEG_ZERO, AV_ZERO, AV_POS_ZERO, AV_POS_ZERO },
        /* >=0 */ { AV_UNKNOWN, AV_UNKNOWN, AV_POS_ZERO, AV_POS_ZERO, AV_POS_ZERO },
        /* >0 */ { AV_UNKNOWN, AV_UNKNOWN, AV_POS_ZERO, AV_POS_ZERO, AV_POSITIVE },
    };

    bcAbsValue(a);
    bcAbsValue(b);
    return map[a][b];
}

// Test if a given concrete value is a member of the set represented
// by the given abstract value.
bool elementOf(int concrete, AbsValue abstract)
{
    switch (abstract) {
        default: assert(!"bad AbsValue");
        case AV_NEGATIVE: return concrete < 0;
        case AV_NEG_ZERO: return concrete <= 0;
        case AV_ZERO: return concrete == 0;
        case AV_POS_ZERO: return concrete >= 0;
        case AV_POSITIVE: return concrete > 0;
        case AV_UNKNOWN: return true;
    }
}

// Concretize 'a' and 'b', filter for pairs satisfying 'op', then
// re-abstract.
AbstractComparisonResult abstractComparison(AbsValue a, AbsValue b,
                                           ConcreteOperator op)
{
    AbstractComparisonResult ret;
    ret.consistent = false;
    ret.newAValue = AV_UNKNOWN;
    ret.newBValue = AV_UNKNOWN;

    // The algorithm here is to just compare all pairs of concrete
    // values drawn from [-2,2], since that is sufficient precision
    // to distinguish all our abstract comparisons.
}
```

```

//
// This is pretty stupid (inefficient), but it works, and we'll only
// do it once at the beginning.
for (int aa=-2; aa<=2; aa++) {
  if (!elementOf(aa, a)) { continue; }
  for (int bb=-2; bb<=2; bb++) {
    if (!elementOf(bb, b)) { continue; }

    // Now 'aa' is a concrete element of 'a', and 'bb' is a
    // concrete element of 'b'.

    // Filter on 'op'.
    if (!op(aa, bb)) {
      continue;
    }

    // Abstract the ('aa', 'bb') pair.
    AbsValue aaa = abstractSingleValue(aa);
    AbsValue bbb = abstractSingleValue(bb);

    // Fold this into our current approximation.
    if (!ret.consistent) {
      ret.consistent = true;
      ret.newAValue = aaa;
      ret.newBValue = bbb;
    }
    else {
      ret.newAValue = greatestLowerBound(ret.newAValue, aaa);
      ret.newBValue = greatestLowerBound(ret.newBValue, bbb);
    }
  }
}

return ret;
}

// ----- relational operators -----
// information about a single relational operator ("<", "=", etc.)
class RelationalOperator {
public:      // data
  // Code to denote it
  BinaryOp binaryOp;

  // concrete comparison function; this is used to compute 'map'
  ConcreteOperator concreteOp;

  // abstract comparison table
  RelationMap map;

public:      // funcs
  RelationalOperator(BinaryOp bop, ConcreteOperator concrete);
};

```

## Checker source files

```
RelationalOperator::RelationalOperator(BinaryOp bop, ConcreteOperator concrete)
: binaryOp(bop),
  concreteOp(concrete)
{
  // compute the abstract operation table
  FOREACH_ABSVAL(a) {
    FOREACH_ABSVAL(b) {
      map[a][b] = abstractComparison(a, b, concreteOp);
    }
  }
}

// concrete comparisons
bool compareLess(int a, int b)      { return a < b; }
bool compareLessEq(int a, int b)   { return a <= b; }
bool compareGreater(int a, int b)  { return a > b; }
bool compareGreaterEq(int a, int b){ return a >= b; }
bool compareEqual(int a, int b)    { return a == b; }
bool compareNotEqual(int a, int b) { return a != b; }

enum { NUM_RELATIONAL_OPERATORS = 6 };
RelationalOperator *relationalOperators[NUM_RELATIONAL_OPERATORS];

// ----- the checker -----
// This store maps expressions to AbsValue; unmapped expressions
// have unknown sign.
START_EXTEND_CHECKER( sign3, int_store );

// Called at program startup.
CHECKER_INIT()
{
  relationalOperators[0] = new RelationalOperator(BIN_LESS, compareLess);
  relationalOperators[1] = new RelationalOperator(BIN_LESSEQ, compareLessEq);
  relationalOperators[2] = new RelationalOperator(BIN_GREATER, compareGreater);
  relationalOperators[3] = new RelationalOperator(BIN_GREATEREQ, compareGreaterEq);
  relationalOperators[4] = new RelationalOperator(BIN_EQUAL, compareEqual);
  relationalOperators[5] = new RelationalOperator(BIN_NOTEQUAL, compareNotEqual);
}

ANALYZE_TREE()
{
  // integer literal?
  Const_int ci;
  if (MATCH(ci)) {
    SET_STATE(CURRENT_TREE, abstractSingleValue(ci.llval()));
    ADD_EVENT(CURRENT_TREE, "literal", "Saw literal value: " << ci.llval());
    return;
  }

  // unsigned variable?
  Scalar scal;
  Var var;
  if (MATCH(var) && MATCH(scal) && scal.get_type()->is_unsigned()) {
```

```

int v;
if (GET_STATE(CURRENT_TREE, v) && v == AV_POSITIVE) {
    // 'var' is already known to be positive, so leave it alone
}
else {
    // set it to >= 0
    CLEAR_STATE(CURRENT_TREE); // avoid lots of 'unsigned' events
    SET_STATE(CURRENT_TREE, AV_POS_ZERO);
    ADD_EVENT(CURRENT_TREE, "unsigned", "Variable is unsigned");
}
}

// arithmetic?
Scalar a,b;
if (MATCH(a+b)) {
    // any prior info we might have had regarding "a+b" is irrelevant
    CLEAR_STATE(CURRENT_TREE);

    int va, vb;
    if (GET_STATE(a, va) && GET_STATE(b, vb)) {
        AbsValue v = abstractAdd(toAbsValue(va), toAbsValue(vb));
        if (v != AV_UNKNOWN) {
            // at this time, there is no way to copy the events from two
            // different sources, so just get what I can ... bug 3439
            COPY_STATE(CURRENT_TREE, a);

            SET_STATE(CURRENT_TREE, v);
            ADD_EVENT(CURRENT_TREE, "addition",
                "Addition: " << a << " (" << toAbsValue(va) <<
                ") plus " << b << " (" << toAbsValue(vb) <<
                ") yields " << v);
        }
    }
    return;
}

if (MATCH(a-b)) {
    CLEAR_STATE(CURRENT_TREE);

    int va, vb;
    if (GET_STATE(a, va) && GET_STATE(b, vb)) {
        AbsValue v = abstractSub(toAbsValue(va), toAbsValue(vb));
        if (v != AV_UNKNOWN) {
            COPY_STATE(CURRENT_TREE, a);
            SET_STATE(CURRENT_TREE, v);
            ADD_EVENT(CURRENT_TREE, "subtraction",
                "Subtraction: " << a << " (" << toAbsValue(va) <<
                ") minus " << b << " (" << toAbsValue(vb) <<
                ") yields " << v);
        }
    }
    return;
}
}

```

```

// assignment?
if (MATCH(a = b)) {
    COPY_STATE(a, b);
    ADD_EVENT(a, "var_assign",
        "Assigning " << a << " to value of " << b);
    return;
}

// possible conversion error?
IntegralType destType;
Cast cast(a, destType); // cast from expression 'a' to type 'destType'
if (MATCH(cast) && !a.get_type()->is_unsigned() && destType.is_unsigned()) {
    int v;
    if (GET_STATE(a, v)) {
        if (v == AV_NEGATIVE) {
            COMMIT_ERROR(a, "conversion_error",
                a << " is converted to 'unsigned' but is known to be negative");
        }
        else if (v == AV_NEG_ZERO) {
            COMMIT_ERROR(a, "conversion_error",
                a << " is converted to 'unsigned' but may be negative");
        }
        else {
            // we know it is *not* negative, so the cast is safe
        }
    }
    else {
        OUTPUT_ERROR(a << " is converted to 'unsigned' but may be negative");
    }
}

// query for abstract value?
if (MATCH(CallSite("whatis")(a))) {
    int val;
    if (GET_STATE(a, val)) {
        COMMIT_ERROR(a, "whatis", a << " has value " << toAbsValue(val));
    }
    else {
        // here, COMMIT_ERROR would do nothing
        OUTPUT_ERROR("whatis: " << a << " has unknown value");
    }
    return;
}

// print entire store?
if (MATCH(CallSite("print_store"))) {
    std::ostream os;
    os << "print_store: ";

    int mappings = 0;
    const ASTNode* t;
    int v;

```

```

FOREACH_IN_STORE(t, v) {
  if (mappings > 0) {
    os << ", ";
  }
  os << t << " has value " << toAbsValue(v);
  mappings++;
}

os << "; " << mappings << " mappings";

OUTPUT_ERROR(os.str());
return;
}
}

ANALYZE_CONDITION()
{
/*
  const cond_cfg_edge_t *cond_edge = dynamic_cast<const cond_cfg_edge_t *>(edge);
  if(!cond_edge) return;
  bool cov_polarity = cond_edge->polarity;
  const Expression *cond = cond_edge->cond;
  const ASTNode* astnode = (const ASTNode*)cond;
*/

  // comparison?
  Expr a, b;
  for (int i=0; i < NUM_RELATIONAL_OPERATORS; i++) {
    RelationalOperator *relop = relationalOperators[i];
    if (MATCH_COND(Binop(relop->binaryOp, a, b))) {
      int va, vb;
      if (!GET_STATE(a, va)) { va = AV_UNKNOWN; }
      if (!GET_STATE(b, vb)) { vb = AV_UNKNOWN; }

      DIAGNOSTIC("matched conditional " <<
        (cov_polarity? " " : "!") << CURRENT_TREE <<
        "; " << a << " = " << toAbsValue(va) <<
        ", " << b << " = " << toAbsValue(vb));

      // do the abstract comparison
      AbstractComparisonResult &res = relop->map[va][vb];
      if (!res.consistent) {
        DIAGNOSTIC(" backtracking due to inconsistency");
        force_backtrack();
        return;
      }
    }

    // update store?
    if (res.newAValue != va) {
      SET_STATE(a, res.newAValue);
      ADD_EVENT(a, "conditional", "Refined via conditional " << CURRENT_TREE);
      DIAGNOSTIC(" refined " << a << " to " << res.newAValue);
    }
  }
}

```

```

    }
    if (res.newBValue != vb) {
        SET_STATE(b, res.newBValue);
        ADD_EVENT(b, "conditional", "Refined via conditional " << CURRENT_TREE);
        DIAGNOSTIC(" refined " << b << " to " << res.newBValue);
    }
}
}
}

END_EXTEND_CHECKER();

MAKE_MAIN( sign3 )

```

### 3.1.4. print\_types.cpp

```

// print type information for every local variable

#include "extend-lang.hpp" // Extend API
#include <string>          // std::string
#include <set>             // std::set

using namespace types;
using namespace std;

// set of classes whose info has been printed
set<string> printedClasses;

// type_recurisve_visitor_t is defined in extend-types.hpp
class ClassTypePrinter : public type_recursive_visitor_t {
public:
    virtual void on_class(const class_type_t &ct);
};

void ClassTypePrinter::on_class(const class_type_t &c)
{
    defined_class_type_t ct = c.load_definition();
    if (!ct) {
        return;
    }

    // obtain qualified name as a string, e.g., "A::B::C"
    ostringstream os;
    os << ct;
    string name = os.str();

    // check to see if we've already printed it
    if (printedClasses.find(name) != printedClasses.end()) {
        return;
    }
    printedClasses.insert(name);

    // print class/struct name

```

```

cout << name << endl;

// print what this class inherits from
foreach(p, ct->get_parents()) {
    cout << " parent: " << p->get_class() << endl;
}

// print fields
foreach(f, ct->get_fields()) {
    cout << " field: " << (*f)->get_pretty_name()
        << ", type: " << (*f)->get_type() << endl;
}

// re-examine field types, looking for classes to print; do this
// after the above loop so we don't get fields from different
// classes mixed together
foreach(f, ct->get_fields()) {
    (*f)->get_type()->visit(*this);
}

// similarly for parent classes
foreach(p, ct->get_parents()) {
    p->get_class()->visit(*this);
}
}

void printVarInfo(const Expression* varTree)
{
    if (!varTree) {
        return;
    }

    type_t const *t = get_type_of_tree(varTree);
    if (!t) {
        return;
    }

    cout << "local variable:\n"
        << " file: " << current_file_get_name() << "\n"
        << " line: " << current_file_lineno() << "\n"
        << " function: " << current_function_get_name() << "\n"
        << " var: " << varTree << "\n"
        << " type: " << *t << endl;

    // visit all the types in 't', looking for classes to print
    ClassTypePrinter ctp;
    ctp(t);
}

START_EXTEND_CHECKER( print_types, simple );

```

```

ANALYZE_TREE()
{
  Decl decl;
  if (MATCH(decl)) {
    printVarInfo(decl.var());
  }
}

END_EXTEND_CHECKER();

MAKE_MAIN( print_types )

```

### 3.1.5. switch\_default.cpp

```

// find switch/case statements with no "default"

#include "extend-lang.hpp"      // Extend API

START_EXTEND_CHECKER( switch_default, simple );

ANALYZE_TREE()
{
  SwitchPat sw;

  if (MATCH(sw)) {
    const S_switch *s = sw.last_stmt();
    vector<const Statement *> cases;

    // getTargets gets all the "S_case" and the "S_default", if
    // any, for the switch.
    // See cc.ast
    s->getTargets(cases);
    bool hasDefault = false;
    foreach(i, cases) {
      const Statement *stmt = *i;
      // as<class>() -> downcast with assert, similar to
      // dynamic_cast<class &>
      // as<class>C() -> same, pointer to const
      // if<class>() -> downcast, NULL on failure, similar do
      // dynamic_cast<class *>
      const S_default *def = stmt->ifS_defaultC();
      if(def) {
        hasDefault = true;
        break;
      } else {
        // Only possibilities = S_default and S_case, so this
        // must be an S_case.
        const S_case *case_stmt =
          stmt->asS_caseC();
        // You can obtain the value of the "case"
        const E_intLit *case_expr = case_stmt->expr;

```

```

        long long case_value = case_expr->i;

        // This is not relevant for this checker, but is only
        // included as an example
    }
}
if(!hasDefault) {
    OUTPUT_ERROR("switch statement doesn't have a \"default\"");
}
}
}
}

END_EXTEND_CHECKER();

MAKE_MAIN( switch_default )

```

### 3.1.6. javascript\_match\_local.cpp

```

#include "extend-lang.hpp" // Extend API

START_EXTEND_CHECKER( javascript_match_local, simple );

PREFER_TO_ANALYZE_JAVASCRIPT();

ANALYZE_TREE()
{
    LocalVar local1;
    LocalVar local2;

    if (MATCH(local1 = local2))
    {
        OUTPUT_ERROR("Found JavaScript local from " << local2 << " to " << local1);
    }
}

END_EXTEND_CHECKER();

MAKE_MAIN( javascript_match_local )

```

---

# Part 4. Coverity Runtime Library Development Guide

## Table of Contents

4.1. Overview .....	112
4.2. Directory Structure .....	113
4.3. Building the Runtime Library for Daemon and Linux .....	114
4.4. Testing the Runtime Library for Linux .....	115
4.5. Deploying a Runtime Library and Daemon for Linux .....	116
4.5.1. Dynamic Library Deployment .....	116
4.5.2. Static Library Deployment .....	116
4.5.3. Daemon Deployment .....	116
4.6. Building the Runtime Library and Daemon for Windows .....	118
4.7. Testing the Runtime Library for Windows .....	119
4.8. Deploying a Runtime Library and Daemon for Windows .....	120
4.9. Configuring the Runtime Library Build for Linux and Windows .....	121
4.9.1. Common Environment Variables .....	121
4.9.2. Linux-specific Environment Variables .....	122
4.9.3. Windows-specific Environment Variables .....	122
4.10. Instrumentation Predicate Language .....	124
4.10.1. Non-CIT compiler support .....	124
4.10.2. CIT predicate language extensions .....	130

---

## Chapter 4.1. Overview

When `cov-build` is used to build an instrumented binary for function coverage analysis, code for gathering coverage data is injected into the program. The runtime library is required to gather the coverage data and write it to an output channel to make it available for analysis.

As provided, the runtime library supports file and network based output. See *Test Advisor 2020.12 User and Administrator Guide* [↗](#) for implementation information.

---

## Chapter 4.2. Directory Structure

The Runtime Library source and build scripts reside in the directory `$PREVENT_ROOT/sdk/runtime/ta-runtime`.

The `windows-scripts` subdirectory contains the Windows command line scripts needed to compile and test the Coverity Runtime Library.

The `tests` subdirectory contains the unit and integration tests. When the library is built and tests executed, the following subdirectories will be created:

- `lib`: Contains the static and shared versions of the Runtime Library.
- `objs`: Contains the object files produced when the Runtime Library source files are compiled.
- `test-results`: Contains one subdirectory for each test that ran, along with the captured output for that test. Each test subdirectory will contain the shared and static versions of the test's executable and the output files produced by the executables.

---

## Chapter 4.3. Building the Runtime Library for Daemon and Linux

Bash shell scripts are provided with the Runtime Library to build the library, build and run a suite of unit and integration tests, and to clean up all build artifacts.

It is advisable to make your own copy of the source code tree and place it under version control, then verify that the code can be built and pass all tests before making modifications.

To perform a build, run the command:

```
bash ./unix-build.sh all
```

Build artifacts will be placed in the following subdirectories:

- `objs`: Will contain the `.o` files compiled from source.
- `lib`: Will contain the new versions of the daemon executable and the shared and static runtime libraries.

In this example, a simple wrapper script changes the compiler name from the default "gcc" to "cc", builds the libraries and runs the tests, with all other variables left at their default values:

```
#!/usr/bin/env bash
export CC=cc
bash ./unix-build.sh all
bash ./unix-build.sh testsuite
```

---

## Chapter 4.4. Testing the Runtime Library for Linux

To run the unit and integration tests on Linux, use the following command within the source directory:

```
bash ./unix-build.sh testsuite
```

Upon completion of the tests, the `unix-build.sh` script will write out a summary of the number of tests run, and the number that failed.

Individual tests can be run by going to the directory for that test and executing:

```
bash ./unix-build.sh
```

Providing `BUILD_TYPE=DEBUG` to the `unix-build.sh` script will enable `errexit` and `trace` modes for the bash scripts. The libraries and tests will be compiled with debugging enabled. Example:

```
BUILD_TYPE=DEBUG bash ./unix-build.sh
```

The unit tests are written to be standalone tests so as not to introduce a dependency on any given unit test framework.

Each integration test exercises specific layers and components of the Runtime Library, with an emphasis on the correct handling of error conditions.

Tests are built and run against the shared and static versions of the Runtime Library.

Many test directories contain copies of expected output data. This can range from binary data (`test-flush`) to expected error messages (`test-logging`).

Cleanup of test artifacts can be done by running the following command in the parent source directory:

```
bash ./unix-build.sh clean
```

---

## Chapter 4.5. Deploying a Runtime Library and Daemon for Linux

### Table of Contents

4.5.1. Dynamic Library Deployment .....	116
4.5.2. Static Library Deployment .....	116
4.5.3. Daemon Deployment .....	116

There are several options for deploying a Runtime Library. The choice of deployment will depend on whether the shared or static version or both will be used, and in the case of a shared library, whether tests will be run on the same machine as where it was built.

### 4.5.1. Dynamic Library Deployment

To deploy a dynamic library:

1. Copy the shared library to a location where it can be read in at build time. Set the access rights so that the library is readable by any user accounts building tests.
2. Modify the XML Configuration files for the gnu compilers and linker to reference this location. See Instrumentation Predicate Language for instructions on how to modify the configuration files.

If the test machine is different from the build machine, or if for some other reason the copy of the shared library used at build time is not accessible at runtime, the developer can make a local copy of the Runtime Library on the test machine and provide access to the copy using one of the following approaches:

1. Add the full path to the local copy, or the environment variable `LD_LIBRARY_PATH`.
2. Modify the XML Configuration files to change the `-rpath=/dir` link directive to point at the correct location. For this option, the Runtime Library must be in the same location on all test machines.
3. Install the Runtime Library in one of the standard library directories, such as `/lib`, `/usr/lib`, `/lib64` or `/usr/lib64`.

### 4.5.2. Static Library Deployment

To deploy a static library:

- Copy the static library to a location where it can be read in at build time. Set the permissions so that the library is readable by any accounts used to build tests.

### 4.5.3. Daemon Deployment

An executable "`ci-daemon`" is built as part of the runtime used for collecting function coverage when instrumented code is executed. Depending on the use case, you may be required to install this executable on the test machine.

If you are using `cov-build` to collect test coverage, no special installation of `ci-daemon` is required. A copy of `ci-daemon` already installed as part of the Coverity Analysis package will be used in this case.

If you are not using `cov-build` to collect test coverage, you will need to copy this executable to the test machine. This executable should be placed in a directory which is included in the `PATH` environment variable when the tests are run. Either a directory already existing in the `PATH` can be used, or this can be placed in a new directory with `PATH` updated accordingly.

---

## Chapter 4.6. Building the Runtime Library and Daemon for Windows

Windows batch scripts are provided with the Runtime Library to build the library as a shared library or DLL, build and run a suite of unit and integration tests, and to clean up all build artifacts.

 **Note**

A static Runtime Library is not provided with Windows.

It is recommended that you make your own copy of the source code tree and place it under version control, then verify that the code can be built and pass all tests before making modifications.

Prior to running the build scripts, the command line environment needs to be configured to target the desired Visual Studio compiler. This can be accomplished in one of the following ways:

Option 1:

Open the appropriate Visual Studio Tools Command Prompt in the Visual Studio Tools folder.

Option 2:

From a command line prompt, run the appropriate `VCVARSALL.BAT` file. For example, in a standard Visual Studio 2013 installation, the command would be:

```
> "c:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\vcvarsall.bat"
```

Option 3:

1. Set the environment variable `VCVARSALL` to point to the location of the `VCVARSALL.BAT` file you want to run. The build script will use this variable to automatically run the file.
2. Set the `CPU_ARCH` variable to `x64` to build a 64-bit library (the default), or to `x86` to build a 32-bit library. This value is passed to the `VCVARSALL.BAT` file to set the target processor architecture for the Visual Studio compiler.

See the documentation on the MSDN website for more information about `VCVARSALL.BAT` and the desired version of Visual Studio and processor architecture. Once the command line environment is configured for Visual Studio, perform a build by running the command:

```
CMD /E:ON /C windows-build.bat all
```

Build artifacts will be placed in the following directories:

- `objs`: Will contain the `.obj` files compiled from source.
- `lib`: Will contain the new version of the Runtime Daemon `EXE`, Runtime Library `DLL` and `LIB` files. When compiled with debug enabled, will also contain the `.PDB` file.

---

## Chapter 4.7. Testing the Runtime Library for Windows

To run the unit and integration tests on Windows, use the following command within the source directory:

```
CMD /E:ON /C windows-build.bat testsuite
```

Upon completion of the tests, the script will write out a summary of the number of tests run, and the number of tests that failed.

Cleanup of test artifacts can be done by running the following command in the parent source directory:

```
CMD /E:ON /C windows-build.bat clean
```

---

## Chapter 4.8. Deploying a Runtime Library and Daemon for Windows

When running an executable instrumented with Coverity Function Coverage, the Runtime Library must be loaded by the program for successful execution.

If the program is run under `cov-build`, the `<cov_lib_deployment_path>` and the `<cov_lib_name>` specified in the compiler configuration will be used to resolve the location of the instrumentation runtime library and daemon.



### Note

If there are multiple compiler configurations with different values for the `<cov_lib_deployment_path>` but the same value for the `<cov_lib_name>`, the results are undefined. Alternatively, different versions of the runtime library can be identified with unique names and referenced in separate compiler configurations. For more information, see "Configuring compilers for Coverity Analysis" in the *Coverity Analysis 2020.12 User and Administrator Guide*.

If instrumented programs are run standalone, then Windows will resolve the location of the Runtime Library and Daemon using the following order:

1. The directory where the executable module for the current process is located.
2. The current directory.
3. The Windows system directory. The `GetSystemDirectory` function retrieves the path of this directory.
4. The Windows directory. The `GetWindowsDirectory` function retrieves the path of this directory.
5. The directories listed in the `PATH` environment variable.

If all of the programs will use the same Runtime Library and Daemon, then the library and daemon can be installed in the locations described in steps 3, 4, or 5. If different versions of the library and daemon are required, then a runtime library and daemon should not be installed in these directories, and the locations described in steps 1 and 2 are preferable.

---

## Chapter 4.9. Configuring the Runtime Library Build for Linux and Windows

### Table of Contents

4.9.1. Common Environment Variables .....	121
4.9.2. Linux-specific Environment Variables .....	122
4.9.3. Windows-specific Environment Variables .....	122

Configuration for building the ci-runtime libraries is controlled via a set of environment variables defined in the file `unix-build-common.sh` on Linux, and `windows-scripts\windows-build-common.bat` on Windows.

Overriding these variables will allow the developer to control compiler and linker options, such as the location of input and output files, and flags passed to the compiler and linker.

### 4.9.1. Common Environment Variables

The following environment variables are common to both the Windows and Linux build scripts:

#### PREVENT\_ROOT

This environment variable must be set if the run-time source distribution is not located inside a Coverity Analysis installation.

#### RUNTIME\_DEST\_DIR

Default destination for build artifacts in the source directory.

#### TESTSUITE\_DIR

Destination directory for test executables and output. Each test appears in its own subdirectory.

#### RT\_OBJ\_DIR

Destination for the compiled object files.

#### LIB\_DIR

Destination for the library files.

#### SHARED\_LIB

Path name for the shared library.

#### CC

The C compiler command.

#### CC\_LD

Linker command used to build the shared library.

#### AR

Archive command used to build the static library.

**CFLAGS**

Compiler options common to both the static and shared versions.

**SHARED\_CFLAGS**

Flags for compiling object files for the shared library.

**SHARED\_LIB\_FLAGS**

The flags needed to link the shared library to the integration tests.

**BUILD\_TYPE**

Normally this variable is not set. Setting it to `DEBUG` builds the library with debug information on the target platform.

**VERBOSE**

Generate detailed output of the build script's execution.

### **4.9.2. Linux-specific Environment Variables**

The following environment variables are only used by the Linux build scripts:

**SHARED\_OBJ\_DIR**

Destination for the shared version of the compiled object files.

**STATIC\_OBJ\_DIR**

Destination for the static version of the compiled object files.

**STATIC\_LIB**

Path name for the static library.

**SHARED\_CC\_LD\_FLAGS**

Flags for linking the shared library.

**LIB\_FLAGS**

Common library flags used by integration tests.

**STATIC\_LIB\_FLAGS**

The flags needed to link the static library to the integration tests.

### **4.9.3. Windows-specific Environment Variables**

The following environment variables are only used by the Windows build scripts:

**IMPORT\_LIB**

The name of the import library associated with the DLL.

**LIB\_OUTPUT\_FLAG**

Flag to indicate the name of the Runtime Library DLL.

**IMPORT\_LIB\_FLAG**

Flag to indicate the name of the associated import library.

**INCLUDE\_FLAG**

Flag to indicate a path for include files.

**SOURCE\_INCLUDE\_DIR**

Directory containing the Runtime Library header files.

**TESTS\_INCLUDE\_DIR**

Directory containing header files shared by the tests.

**C\_OUTPUT\_FILE\_FLAG**

Flag to indicate the name of an output file (usually an executable) for the C compiler.

**VCVARSALL**

Optional variable containing the full path name of the VCVARSALL.BAT file that will be used to set up compiler dependencies for the build. Only used when cl.exe is not among the PATH directories.

**CPU\_ARCH**

Indicates the processor architecture to target when the VCVARSALL.BAT file is used. Legal values are "x86" for 32-bit applications and "x64" for 64-bit applications. The default value is "x86".

---

## Chapter 4.10. Instrumentation Predicate Language

### Table of Contents

4.10.1. Non-CIT compiler support .....	124
4.10.2. CIT predicate language extensions .....	130

The Instrumentation Predicate Language supports Instrumentation Predicate Language and the runtime library by allowing the specification of rules and conditional logic to modify native compiler command lines. The Instrumentation Predicate Language is specified in the form of nested xml tags, within the standard compiler configuration files.

### 4.10.1. Non-CIT compiler support

Default Instrumentation Predicate Language rules are generated for the gcc family of compilers and linkers (gcc, g++, ld, collect2). Compiler configuration files specify whether the target binary is a compiler and/or a linker with the following xml fields:

- `<is_compiler>[true|false]</is_compiler>`
- `<is_linker>[true|false]</is_linker>`

Instrumentation is only carried out when `is_linker` is set to true, otherwise any Instrumentation Predicate Language rules will be ignored. These xml fields must appear after all other fields in the `compiler` node for each relevant compiler variant. Example:

```
...
<config>
  <build>
    <compiler>
      <template_compiler>true</template_compiler>
      <comp_name>ld</comp_name>
      <comp_translator>ld</comp_translator>
      <comp_lang>C/C++</comp_lang>
      <fake_compiler>true</fake_compiler>
      <is_compiler>false</is_compiler>
      <is_linker>true</is_linker>
    </compiler>
  ...
...

```

Additionally, all Instrumentation Predicate Language xml fields must be contained within opening and closing `<instrument_predicate>` tags, and appear in the compiler-specific options node above the `<begin_command_line_config>` field. Example:

```
...
<config>
  <build>
    <compiler>

```

```

...
<is_compiler>>false</is_compiler>
<is_linker>>true</is_linker>
</compiler>
<options>
  <id>ld-ld-.*</id>
  <opt_preinclude_file>${CONFIGDIR}/../user_nodefs.h</opt_preinclude_file>
  <instrument_predicates>
    <remove_regex>-Werror</remove_regex>
  </instrument_predicates>
  ...
  <begin_command_line_config></begin_command_line_config>
  <pre_prepend_arg>--no_error_recovery</pre_prepend_arg>
</options>
...
...

```

If the `<instrument_predicate>` tags are placed under the `<begin_command_line_config>` field, the compilation will result in a fatal error.

#### Note

In template configuration files, the `is_compiler` and `is_linker` fields are immutable and are displayed in the file only for informational purposes. Other Instrumentation Predicate Language elements (`instrument_predicates`, `instrument_variables`, `instrument_unsupported`) are present and modifiable in template files.

#### 4.10.1.1. Removing arguments from the native command line

Instrumentation Predicate Language provides the `remove_regex` rule for removing arguments from the native command line. This xml field specifies a regex to search for, and remove from, the command line. For example, consider the following scenario.

Original command line:

```
> thing1 foo1 thing2 foo2
```

Removal rule:

```
<remove_regex>foo</remove_regex>
```

Resulting command line:

```
> thing1 thing2
```

The regex "foo" matched `foo1` and `foo2`, so both were removed from the command line.

#### 4.10.1.2. Inserting arguments onto the native command line

Instrumentation Predicate Language also provides rules for inserting strings onto the command line. This process is slightly more complex than removing arguments, as the relative position of the inserted string must also be specified.

Each insertion rule must be surrounded by opening and closing `<insert>` tags. Specify the string you want to insert onto the command line with the `insertion_string` field. By default, the inserted string will be placed at the very beginning of the command line. For example:

Original command line:

```
> thing1 fool
```

Insertion rule:

```
<insert>
  <insertion_string>arg1</insertion_string>
</insert>
```

Resulting command line:

```
> arg1 thing1 fool
```

Insertion rules can also specify conditions on the placement of the new string within the native command line. These conditions use regex matching to locate an existing argument, and specify whether to place the new string before or after the matched regex. Condition tags must be placed in the same set of `<insert>` tags as the string to be inserted.

#### 4.10.1.2.1. Insert condition: `precedes_regex`

The `precedes_regex` condition specifies that the insertion string should be placed on the command line, just before the left-most regex match. Multiple `precedes_regex` conditions may be specified, and the inserted string will be placed immediately before the first argument that satisfies any condition. For example:

Original command line:

```
> thing1 fool
```

Insertion rule:

```
<insert>
  <insertion_string>arg1</insertion_string>
  <precedes_regex>foo</precedes_regex>
</insert>
```

Resulting command line:

```
> thing1 arg1 fool
```

If there is no match on the command line for the regex, the insertion string will be placed at the default location (the beginning of the command line).

You can also specify a special value of "nothing" to `precedes_regex`. This specifies that the inserted string should precede no other argument, and will instead be placed at the very end of the command line. Alternatively, using the special value, "anything", will cause the string to be placed at the start of the command line.

#### 4.10.1.2.2. Insert condition: `follows_regex`

The `follows_regex` condition specifies that the insertion string should be placed on the command line, after the right-most regex match. Multiple `follows_regex` conditions may be specified, and the inserted string will be placed directly after the first argument that satisfies any condition. For example:

Original command line:

```
> thing1 fool
```

Insertion rule:

```
<insert>
  <insertion_string>arg1</insertion_string>
  <follows_regex>foo</follows_regex>
</insert>
```

Resulting command line:

```
> thing1 fool arg1
```

If there is no match on the command line for the regex, the insertion string will be placed at the default location (the beginning of the command line).

You can also specify a special value of "anything" to `follows_regex`. This specifies that the inserted string should follow all other arguments, and will be placed at the very end of the command line. Alternatively, using the special value, "nothing", will cause the string to be placed at the start of the command line.

#### 4.10.1.2.3. Combined conditions

It is possible to specify both `precedes_regex` and `follows_regex` conditions in the same insertion rule. They will be combined to place the insertion string between two existing arguments. For example:

Original command line:

```
> -one -two -three -four
```

Insertion rule:

```
<insert>
  <insertion_string>arg1</insertion_string>
  <precedes_regex>three</precedes_regex>
  <follows_regex>two</follows_regex>
</insert>
```

Resulting command line:

```
> -one -two arg1 -three -four
```

In this case, `arg1` *follows* the matched argument, `-two`, and *precedes* the matched argument, `-three`. If the `precedes_regex` and `follows_regex` conditions contradict each other (so that satisfying one condition makes it impossible to satisfy the other), the result is a fatal error.

If there is no match on the command line for either regex, the insertion string will be placed at the default location (the beginning of the command line).

#### 4.10.1.2.4. Insertion groups

Instrumentation Predicate Language allows the grouping of insertion rules into if/else style blocks. This places individual insertion rules in order, so that if the first insertion rule is unsatisfiable (because it has contradictory `precedes_regex` and `follows_regex` conditions), the next insertion rule is executed instead. Insertion groups are specified by opening and closing `<insert_group>` tags. Within these tags will be one `<insert>` rule to attempt first, followed by one or more `<or_insert>` rules, to be attempted in order until one is executed. For example:

Original command line:

```
> -one -two -three -four
```

Insertion rule:

```
<insert_group>
  <insert>
    <insertion_string>something</insertion_string>
    <precedes_regex>two</precedes_regex>
    <follows_regex>three</follows_regex>
  </insert>
  <or_insert>
    <insertion_string>something_else</insertion_string>
    <precedes_regex>three</precedes_regex>
    <follows_regex>two</follows_regex>
  </or_insert>
</insert_group>
```

Resulting command line:

```
> -one -two arg1 -three -four
```

The first insert rule in the group is unsatisfiable against the given command line. This means that the `or_insert` rule is attempted, and because it is satisfiable, results in the final command line.

If none of the insert rules inside an insertion group are satisfiable, the result is a fatal error.

#### 4.10.1.2.5. Instrumentation variables

Instrumentation variables are placeholders in an insertion string, the value of which can be specified in the configuration file by using the following syntax:

```
<instrument_variable>
  <var_name>variable name</var_name>
  <var_value>value of the variable</var_value>
</instrument_variable>
```

These `instrument_variable` specifications should appear outside of the `<instrument_predicates>` tags. The supported instrumentation variables are named:

- cov\_lib\_name
- cov\_lib\_path
- cov\_lib\_deployment\_path

The correct syntax for specifying a instrumentation variable within an insertion string is `[[:variable_name:]]`. For example:

```
<insertion_string>-r[[:cov_lib_path:]] -l[[:cov_lib_name:]]</insertion_string>
```

### 4.10.1.3. Conditional branching

Instrumentation Predicate Language supports conditional logic, using the following if/elif conditions:

- `<contains_regex>regex</contains_regex>`
- `<not_contains_regex>regex</not_contains_regex>`
- `<is_compiler>true|false</is_compiler>`
- `<is_linker>true|false</is_linker>`

The syntax for conditional statements in Instrumentation Predicate Language is as follows:

```
<if>
  <conditions>
    <cond>
      <is_compiler>true</is_compiler>
    </cond>
  </conditions>
  <then>
    ...           // insertion and/or removal rules
  </then>
</if>
<elif>
  <conditions>
    <cond>
      <is_linker>true</is_linker>
    </cond>
  </conditions>
  <then>
    ...           // different insertion and/or removal rules
  </then>
</elif>
...
```

#### Conjunctions: and\_cond/or\_cond

You can use and/or conjunctions in if/elif statements, using the `<and_cond>` and `<or_cond>` xml fields. See below:

```
<and_cond>
  <cond>
```

```
<contains_regex>some regex</contains_regex>
</cond>
<cond>
  <is_compiler>>false</is_compiler>
</cond>
<cond>
  <is_linker>>true</is_linker>
</cond>
</and_cond>
```

In the example above, the `<and_cond>` field returns true if each of the nested conditions evaluate to true, otherwise the entire block is false. When using `<or_cond>`, the field is true if at least one of the nested conditions is true, and will only be false if all of the nested conditions are false.

Conjunctions can be nested within each other. For example:

```
<and_cond>
  <cond>A</cond>
  <or_cond>
    <cond>B</cond>
    <cond>C</cond>
  </or_cond>
</and_cond>
```

The example above will only return true if:

- A is true
- AND
- B OR C is true

#### 4.10.1.4. Unsupported switches

Coverity instrumentation does not support certain switches for certain compilers. For example, `-spec file` is unsupported for `gcc`. If this appears on a native command line, it will result in a fatal error for `cov-translate`. You can specify which switches are unsupported by using `<instrument_unsupported>` tags. For example:

```
<instrument_unsupported>switch1</instrument_unsupported>
<instrument_unsupported>switch2</instrument_unsupported>
```

These tags must be placed outside of the `instrument_predicates` tags.

#### 4.10.2. CIT predicate language extensions

As with non-CIT compilers, CIT compilers can also make use of regex matching to achieve removal and insertion of command-line arguments. However, because CIT compilers support switch table rules, more complex extensions to the Instrumentation Predicate Language can be used.

Supported CIT predicate language extensions include non-regex versions of the standard, non-CIT predicates:

- `<contains>..</contains>`
- `<not_contains>..</not_contains>`
- `<remove>..</remove>`
- `<precedes>..</precedes>`
- `<following>..</following>`



### Note

Each of the above is evaluated using the switch-parsing rules in the relevant `<compiler>_switches.dat` switch table. Note that use of CIT predicate language extensions for non-CIT compilers will result in a fatal error in `cov-translate`.

Syntax of the above CIT extensions is common to each of the predicates. For example:

```
<contains>
  <switch>foo</switch>
  <arg>arg1</arg>
  <arg>arg2</arg>
  <arg>...</arg>
</contains>
```

The "switch" xml tag should not contain any switch prefixes. For example, if the command line contains any of the following switches, then the `switch` tag should only contain "foo":

- `-foo`
- `/foo`
- `--foo`

Arguments to the `switch` tag are considered regex matches. Specifically, consider the following command line:

```
> -foo=two /foo=three --foo=four
```

The condition below will match on `-foo=two /foo=three`.

```
<contain>
  <switch>foo</switch>
  <arg>^t</arg>
</contains>
```

Note in particular that the regex matches on `switch` arguments do not cross argument boundaries.

#### 4.10.2.1. `oa_split`

Switches such as `-Ipath1,path2,path3` that are really equivalent to `-Ipath1 -Ipath2 -Ipath3` (where the switch arguments are separated by one or more delimiters) are specified in the switch table

as `oa_split` followed immediately by the delimiter(s). Specifying CIT instrumentation predicates for switches like this behave as follows:

```
<contains>
  <switch>I</switch>
  <arg>path</arg>
</contains>
```

The above rule evaluates to true if there is a switch `-I` on the command line that has any argument component matching the regex "path".

In the following condition:

```
<contains>
  <switch>I</switch>
  <arg>path1</arg>
  <arg>path2</arg>
</contains>
```

If there exists a switch `-I` on the command line where either `path1` or `path2` is one of its arguments, there is a match.

#### 4.10.2.2. `oa_alternate_table`

When a switch is marked `oa_alternate_table` in the switch table, this designates that the switch is for specifying switches to another program, and that you need to use an alternate switch table to interpret it. For example, the following signifies that the value to `Xpreprocessor` should be interpreted by `<compiler>_preprocessor_switches.dat` and the results should be appended to the command line:

```
{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor", oa_append}
```

Suppose a command line contains:

```
-Wp,-MD file.c -Wp,foo,-MM,-MP -Wp,-MD,bar -Wp,-MD,foo2
```

where the preprocessor switch table contains entries

```
{"MD", oa_dash|oa_unattached},
{"MM", oa_dash},
{"MP", oa_dash}
```

Example 1:

```
<contains>
  <switch>Wp,</switch>
  <arg>MD</arg>
</contains>
```

This will result in a match against any switch grouping with `-Wp,MD` in it. In the above case, a match will be made on all of `-Wp,-MD -Wp,foo,-MM,-MP -Wp,-MD,bar -Wp,-MD,foo2`.

Example 2:

```
<contains>
```

```
<switch>Wp,</switch>  
<arg>MD</arg>  
<arg>foo</arg>  
</contains>
```

This will match on `-Wp, -MD -Wp,foo, -MM, -MP -Wp, -MD,foo2`.

### Example 3:

```
<contains>  
<switch>Wp,</switch>  
<arg>MD</arg>  
<arg>bar</arg>  
</contains>
```

This will match on `-Wp, -MD,bar`