Static Analysis Results Interchange Format (SARIF) Version 1.0

Working Draft 01

15 September 2017

Technical Committee:

[OASIS Static Analysis Results Interchange Format (SARIF) TC](https://www.oasis-open.org/committees/sarif/)

Chairs:

David Keaton ([dmk@dmk.com](mailto:dmk@dmk.com)), Individual Member

Luke Cartey ([luke@semmle.com](mailto:luke@semmle.com)), [Semmle](http://www.semmle.com/)

Editor:

Michael Fanning ([mikefan@microsoft.com](mailto:mikefan@microsoft.com)), [Microsoft](http://www.microsoft.com/)

Laurence J. Golding ([larrygolding@comcast.net](mailto:larrygolding@comcast.net)), Individual Member

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

* JSON schemas: (list file names or directory name)
* Other parts (list titles and/or file names)
* (**Note:** Any normative computer language definitions that are part of the Work Product, such as XML instances, schemas and Java(TM) code, including fragments of such, must be (a) well formed and valid, (b) provided in separate plain text files, (c) referenced from the Work Product; and (d) where any definition in these separate files disagrees with the definition found in the specification, the definition in the separate file prevails. Remove this note before submitting for publication.)

Related work:

This specification replaces or supersedes:

* None

This specification is related to:

* None

Declared XML namespaces:

* None

Abstract:

This document defines a standard format for the output of static analysis tools. The format is referred to as the “Static Analysis Results Interchange Format”, and is abbreviated as SARIF.

Status:

This [Working Draft](https://www.oasis-open.org/policies-guidelines/oasis-defined-terms-2017-05-26#dWorkingDraft) (WD) has been produced by one or more TC Members; it has not yet been voted on by the TC or [approved](https://www.oasis-open.org/policies-guidelines/tc-process#committeeDraft) as a Committee Draft (Committee Specification Draft or a Committee Note Draft). The OASIS document [Approval Process](https://www.oasis-open.org/policies-guidelines/tc-process#standApprovProcess) begins officially with a TC vote to approve a WD as a Committee Draft. A TC may approve a Working Draft, revise it, and re-approve it any number of times as a Committee Draft.

This Working Draft is being developed under the [RF on RAND Terms](https://www.oasis-open.org/policies-guidelines/ipr#RF-on-RAND-Mode) Mode of the [OASIS IPR Policy](https://www.oasis-open.org/policies-guidelines/ipr), the mode chosen when the Technical Committee was established. All members of the TC should be familiar with this document, which may create obligations regarding the disclosure and availability of a member's patent, copyright, trademark and license rights that read on an approved OASIS specification. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC’s web page (<https://www.oasis-open.org/committees/sarif/ipr.php>).

Any machine-readable content ([Computer Language Definitions](https://www.oasis-open.org/policies-guidelines/tc-process#wpComponentsCompLang)) declared Normative for this Work Product must also be provided in separate plain text files. In the event of a discrepancy between such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

URI patterns:

Initial publication URI:  
http://docs.oasis-open.org/sarif/sarif/v1.0/csd01/sarif-v1.0-csd01.docx

Permanent “Latest version” URI:  
http://docs.oasis-open.org/sarif/sarif/v1.0/sarif-v1.0.docx

Copyright © OASIS Open 2017. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](https://www.oasis-open.org/policies-guidelines/ipr) may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1 Introduction 9

1.1 IPR Policy 9

1.2 Terminology 9

1.3 Normative References 13

1.4 Non-Normative References 13

2 Conventions 14

2.1 General 14

2.2 Format examples 14

2.3 Property notation 14

3 File format 15

3.1 General 15

3.2 URI-valued properties 15

3.3 URI base id properties 15

3.4 String properties 16

3.5 Object properties 17

3.6 Array properties 17

3.7 Property bags 17

3.7.1 General 17

3.7.2 Tags 17

3.8 Date/time properties 17

3.9 Array properties with unique values 17

3.10 Message properties 18

3.11 sarifLog object 18

3.11.1 General 18

3.11.2 version property 18

3.11.3 $schema property 18

3.11.4 runs property 19

3.12 run object 19

3.12.1 General 19

3.12.2 id property 19

3.12.3 stableId property 19

3.12.4 baselineId property 19

3.12.5 automationId property 20

3.12.6 architecture property 20

3.12.7 tool property 20

3.12.8 invocation property 20

3.12.9 files property 20

3.12.10 logicalLocations property 22

3.12.11 results property 23

3.12.12 toolNotifications property 23

3.12.13 configurationNotifications property 24

3.12.14 rules property 25

3.12.15 properties property 26

3.13 tool object 26

3.13.1 General 26

3.13.2 name property 26

3.13.3 fullName property 26

3.13.4 semanticVersion property 26

3.13.5 version property 27

3.13.6 fileVersion property 27

3.13.7 language property 27

3.13.8 sarifLoggerVersion property 27

3.13.9 properties property 28

3.14 invocation object 28

3.14.1 General 28

3.14.2 commandLine property 28

3.14.3 responseFiles property 28

3.14.4 startTime property 29

3.14.5 endTime property 29

3.14.6 machine property 29

3.14.7 account property 29

3.14.8 processId property 29

3.14.9 fileName property 29

3.14.10 workingDirectory property 30

3.14.11 environmentVariables property 30

3.14.12 properties property 30

3.15 file object 30

3.15.1 General 30

3.15.2 uri property 30

3.15.3 uriBaseId property 31

3.15.4 parentKey property 32

3.15.5 offset property 32

3.15.6 length property 32

3.15.7 mimeType property 32

3.15.8 hashes property 32

3.15.9 contents property 33

3.15.10 properties property 33

3.16 hash object 33

3.16.1 General 33

3.16.2 value property 33

3.16.3 algorithm property 34

3.17 result object 34

3.17.1 General 34

3.17.2 ruleId property 34

3.17.3 ruleKey property 35

3.17.4 level property 35

3.17.5 message property 37

3.17.6 formattedRuleMessage property 37

3.17.7 locations property 38

3.17.8 snippet property 38

3.17.9 toolFingerprintContribution property 38

3.17.10 codeFlows property 38

3.17.11 stacks property 39

3.17.12 relatedLocations property 39

3.17.13 suppressionStates property 40

3.17.13.1 General 40

3.17.13.2 suppressedInSource value 40

3.17.13.3 suppressedExternally value 40

3.17.14 baselineState property 40

3.17.15 fixes property 41

3.17.16 properties property 41

3.18 location object 41

3.18.1 General 41

3.18.2 Constraints 41

3.18.3 analysisTarget property 42

3.18.4 resultFile property 42

3.18.5 fullyQualifiedLogicalName property 43

3.18.6 logicalLocationKey property 43

3.18.7 decoratedName property 44

3.18.8 properties property 44

3.19 physicalLocation object 45

3.19.1 General 45

3.19.2 uri property 45

3.19.3 uriBaseId property 46

3.19.4 region property 46

3.20 region object 46

3.20.1 General 46

3.20.2 Text regions 46

3.20.3 Binary regions 48

3.20.4 startLine property 48

3.20.5 startColumn property 48

3.20.6 endLine property 48

3.20.7 endColumn property 48

3.20.8 offset property 48

3.20.9 length property 48

3.21 logicalLocation object 49

3.21.1 General 49

3.21.2 name property 49

3.21.3 kind property 50

3.21.4 parentKey property 50

3.22 codeFlow object 50

3.22.1 General 50

3.22.2 message property 50

3.22.3 locations property 50

3.22.4 properties property 51

3.23 stack object 51

3.23.1 General 51

3.23.2 message property 51

3.23.3 frames property 51

3.23.4 properties property 51

3.24 stackFrame object 51

3.24.1 General 51

3.24.2 message property 51

3.24.3 uri property 51

3.24.4 uriBaseId property 52

3.24.5 line property 52

3.24.6 column property 52

3.24.7 module property 52

3.24.8 threadId property 52

3.24.9 fullyQualifiedLogicalName property 52

3.24.10 logicalLocationKey property 52

3.24.11 address property 53

3.24.12 offset property 53

3.24.13 parameters property 53

3.24.14 properties property 53

3.25 annotatedCodeLocation object 53

3.25.1 General 53

3.25.2 step property 53

3.25.3 physicalLocation property 53

3.25.4 fullyQualifiedLogicalName property 54

3.25.5 logicalLocationKey property 54

3.25.6 module property 54

3.25.7 threadId property 54

3.25.8 message property 54

3.25.9 kind property 54

3.25.10 kind-dependent properties: target, targetLocation, values and state 55

3.25.11 targetKey property 61

3.25.12 importance property 61

3.25.13 taintKind property 62

3.25.14 snippet property 62

3.25.15 annotations property 62

3.25.16 properties property 63

3.26 annotation object 63

3.26.1 General 63

3.26.2 message property 63

3.26.3 locations property 63

3.27 rule object 63

3.27.1 General 63

3.27.2 Constraints 63

3.27.3 id property 64

3.27.4 name property 64

3.27.5 shortDescription property 64

3.27.6 fullDescription property 64

3.27.7 defaultLevel property 65

3.27.8 messageFormats property 65

3.27.9 helpUri property 66

3.27.10 properties property 66

3.28 formattedMessage object 66

3.28.1 General 66

3.28.2 formatId property 66

3.28.3 arguments property 66

3.29 fix object 66

3.29.1 General 66

3.29.2 description property 67

3.29.3 fileChanges property 67

3.30 fileChange object 67

3.30.1 General 67

3.30.2 uri property 68

3.30.3 uriBaseId property 68

3.30.4 replacements property 68

3.31 replacement object 68

3.31.1 General 68

3.31.2 Constraints 69

3.31.3 offset property 69

3.31.4 deletedLength property 69

3.31.5 insertedBytes property 69

3.32 notification object 69

3.32.1 General 69

3.32.2 id property 70

3.32.3 ruleId property 70

3.32.4 ruleKey property 70

3.32.5 physicalLocation property 70

3.32.6 message property 71

3.32.7 level property 71

3.32.8 threadId property 71

3.32.9 time property 71

3.32.10 exception property 71

3.32.11 properties property 71

3.33 exception object 71

3.33.1 General 71

3.33.2 kind property 71

3.33.3 message property 72

3.33.4 stack property 72

3.33.5 innerExceptions property 72

4 Conformance 73

Appendix A. Acknowledgments 74

Appendix B. Use of fingerprints by result management systems 75

Appendix C. Use of SARIF by log file viewers 76

Appendix D. Production of SARIF by converters 77

Appendix E. Locating rule metadata 78

Appendix F. Producing deterministic SARIF log files 79

F.1 General 79

F.2 Non-deterministic file format elements 79

F.3 Array and dictionary element ordering 80

F.4 Absolute paths 80

F.5 Compensating for non-deterministic output 80

F.6 Interaction between determinism and baselining 81

Appendix G. Guidance on fixes 82

Appendix H. Examples 83

H.1 Minimal valid SARIF file resulting from a scan 83

H.2 Minimal recommended SARIF file with source information 83

H.3 Minimal recommended SARIF file without source information 84

H.4 SARIF file for exporting rule metadata 85

H.5 Comprehensive SARIF file 86

Appendix I. Revision History 92

# Introduction

Software developers use a variety of analysis tools to assess the quality of their programs. These tools report results which can indicate problems related to program qualities such as correctness, security, performance, conformance to contractual or legal requirements, conformance to stylistic standards, understandability, and maintainability. To form an overall picture of program quality, developers must often aggregate the results produced by all of these tools. This aggregation is more difficult if each tool produces output in a different format.

This document defines a standard format for the output of static analysis tools. The goals of the format are:

* Comprehensively capture the range of data produced by commonly used static analysis tools.
* Be a useful format for analysis tools to emit directly, and also an effective interchange format into which the output of any analysis tool can be converted.
* Be suitable for use in a variety of scenarios related to analysis result management, and be extensible for use in new scenarios.
* Reduce the cost and complexity of aggregating the results of various analysis tools into common workflows.
* Capture information that is useful for assessing a project’s compliance with corporate policy or conformance to certification standards.
* Adopt a widely used serialization format that can be parsed by readily available tools.
* Represent analysis results for all kinds of programming artifacts, including source code and object code.
* Represent the logical construct against which a result is produced, such as a function, class, or namespace.
* Represent the physical location at which a result is produced, including problems that are detected in nested files (such as a source file within a compressed container).

## IPR Policy

This Working Draft is being developed under the [RF on RAND Terms](https://www.oasis-open.org/policies-guidelines/ipr#RF-on-RAND-Mode) Mode of the [OASIS IPR Policy](https://www.oasis-open.org/policies-guidelines/ipr), the mode chosen when the Technical Committee was established.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC’s web page (<https://www.oasis-open.org/committees/sarif/ipr.php>).

## Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [[RFC2119](#RFC2119)].

For purposes of this document, the following terms and definitions apply:

file

sequence of bytes accessible *via* a URI

Example: A physical file in a file system, a specific version of a file in a version control system.

top-level file

file which is not contained within any other file

nested file

file which is contained within another file

parent (file)

file which contains one or more nested files

(programming) artifact

file, produced manually by a person or automatically by a program, which results from the activity of programming

Example: Source code, object code, program configuration data, documentation.

result

condition present in a programming artifact

problem

result which indicates a condition that has the potential to detract from the quality of the program

Example: A security vulnerability, a deviation from conformance to contractual or legal requirements, a deviation from conformance to stylistic standards.

(static analysis) tool

program that examines programming artifacts in order to detect problems, without executing the program

Example: Lint

conversion tool, converter

program that converts the output of another program into a different format

analysis target

programming artifact which a static analysis tool is instructed to analyze

result file

file in which a static analysis tool detects a result

rule

specific criterion for correctness verified by a static analysis tool

NOTE 1: Many static analysis tools associate a “rule id” with each result they report, but some do not.

NOTE 2: Some rules verify generally accepted criteria for correctness; others verify conventions in use in a particular team or organization.

Example: “Variables must be initialized before use”, “Class names must begin with an uppercase letter”.

stable value

value which, once established, never changes over time

rule id

stable value which a static analysis tool associates with a rule

NOTE: A rule id is more likely to remain stable if it is a symbolic or numeric value, as opposed to a descriptive string.

Example: CA2001

rule metadata

information that describes a rule

Example: Category (for example, “Style” or “Security”), documentation URI.

log file

output file produced by a static analysis tool, which enumerates the results produced by the tool

run

1. invocation of a specified static analysis tool on a specified version of a specified set of analysis targets, with a specified set of runtime parameters

2. set of results produced by such an invocation

triage

process of deciding whether a result reported by a static analysis tool indicates a problem that should be corrected

(end) user

person who uses the information in a log file to investigate, triage, or resolve results detected by a static analysis tool

false positive

result which an end user decides does not actually represent a problem

(log file) viewer

program that reads a log file, displays a list of the results it contains, and allows an end user to view each result in the context of the programming artifact in which it occurs

result management system

software system that consumes the log files produced by static analysis tools, produces reports that enable software development teams to assess the quality of their software artifacts at a point in time and to observe trends in the quality over time, and performs functions such as filing bugs and displaying information about individual results

NOTE: A result management system can interact with a log file viewer to display information about individual defects.

fingerprint

stable value that can be used by a result management system to uniquely identify a result over time, even if the programming artifact in which it occurs is modified

baseline

set of results produced by a single run of a set of static analysis tools on a set of programming artifacts

NOTE: A result management system can compare the results of a subsequent run to a baseline to determine whether new results have been introduced.

code flow

sequence of program locations that specify a possible execution path through the code

call stack

sequence of nested function calls

camelCase name

name that begins with a lowercase letter, in which each subsequent word begins with an uppercase letter

Example: camelCase, version, fullName.

property bag

JSON object consisting of a set of name/value pairs with arbitrary camelCase names

newline sequence

sequence of one or more characters representing the end of a line of text

NOTE: Some systems represent a newline sequence with a single newline character; others represent it as a carriage return character followed by a newline character.

text file

file considered as a sequence of characters organized into lines and columns

line

contiguous sequence of characters, starting either at the beginning of a file or immediately after a newline sequence, and ending at and including the nearest subsequent newline sequence, if one is present, or else extending to the end of the file

column

1-based index of a character within a line

binary file

file considered as a sequence of bytes

region

contiguous portion of a file

text region

region representing a contiguous range of zero or more character in a text file

binary region

region representing a contiguous range of zero or more bytes in a binary file

physical location

location specified by reference to a programming artifact together with a region within that artifact

logical location

location specified by reference to a programmatic construct, without specifying the programming artifact within which that construct occurs

Example: A class name, a method name, a namespace.

top-level logical location

logical location that is not nested within another logical location

Example: A global function in C++

nested logical location

logical location that is nested within another logical location

Example: A method within a class in C++

empty array

array that contains no elements, and so has a length of 0

empty object

object that contains no properties

empty string

string that contains no characters, and so has a length of 0

response file

file containing arguments for a tool, which are interpreted as if they had appeared directly on the command line

tainted data

data that enters a program from an untrusted source, such as user input

taint analysis

the process of tracing the path of tainted data through a program

## Normative References

[ECMA404] “The JSON Data Interchange Format”, 1st Edition, ECMA-404, October 2013, <http://ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.

[FIPSPUB180-4] “Secure Hash Standard (SHS)”, FIPS PUB 180-4, August 2015, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[ISO8601:2004] “Data elements and interchange formats -- Information interchange -- Representation of dates and times”, ISO 8601:2004, December 2004, <https://www.iso.org/standard/40874.html>.

[JSCHEMA01]Wright, A., “JSON Schema: A Media Type for Describing JSON Documents”, April 2017 (expires October 2017), <http://json-schema.org/latest/json-schema-core.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.ietf.org/rfc/rfc2119.txt>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <http://www.rfc-editor.org/info/rfc2045>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://www.rfc-editor.org/info/rfc3629>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

[RFC5646] Phillips, A., Ed., and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <http://www.rfc-editor.org/info/rfc5646>.

[SEMVER] “Semantic Versioning 2.0.0”, <http://semver.org/>.

[UNICODE10] Unicode 10.0, June 2017, <http://www.unicode.org/versions/Unicode10.0.0/>

## Non-Normative References

[ISO9899:2011] “Information technology – Programming languages – C”, ISO/IEC 9899, December 2011, <https://www.iso.org/standard/57853.html>

[ISO14882:2014] “Information technology – Programming languages – C++”, ISO/IEC 14882, December 2014, <https://www.iso.org/standard/64029.html>

[ISO23270:2006] “Information technology – Programming languages – C#”, ISO/IEC 23270, September 2006, <https://www.iso.org/standard/42926.html>

# Conventions

## General

The following conventions are used within this document.

## Format examples

This document contains several partial examples of the SARIF format. The examples are formatted for clarity, as permitted by [[ECMA404](#ECMA404)], which allows “insignificant whitespace” before or after any token; implementations do not need to follow the whitespace convention used in these examples. In these examples, an ellipsis (…) is used to indicate that portions of the log file text required by this specification have been omitted for brevity. A ‘#’ character introduces a comment that extends to the end of the line. These comments are present for explanatory purposes and are not part of the SARIF file format. When a JSON string is too long to fit on a line, it is broken into multiple lines. This is not part of the SARIF format, since JSON strings cannot contain control characters such as newlines.

## Property notation

A JSON object consists of a set of name/value pairs. The value may itself be an object, allowing arbitrary nesting. When necessary for clarity or to avoid ambiguity, we use the “dot” notation to refer to nested values. For example, the physicalLocation object defines a property region whose value is a region object, which in turn contains a length property. For clarity, we can refer to the length property as physicalLocation.region.length.

# File format

## General

A SARIF log file **SHALL** contain the results of a one or more analysis runs. The runs do not need to be produced by the same analysis tool.

A SARIF log file **SHALL** conform to the requirements of the JSON format. The top-level value in the log file **SHALL** conform to the JSON object grammar; that is, it **SHALL** consist of a comma-separated sequence of name/value pairs, enclosed in curly brackets, as described in [[ECMA404](#ECMA404)]. We refer to the object represented by this top-level value as the sarifLog object (§3.11).

## URI-valued properties

Certain properties in this specification specify the URI of a file. The value of every such property, if present, **SHALL** be a valid URI as described in [[RFC3986](#RFC3986)].

If a URI refers to a file stored in a version control system (VCS), the value **SHALL** preserve relevant details that permit the target file to be retrieved from the VCS. If the URI refers to a file stored on a physical file system, it **MAY** be specified as a relative URI that omits root information details (such as hard drive letter and an arbitrarily named root directory associated with a source code enlistment).

NOTE 1: An absolute URI might contain information that represents unwanted information disclosure, particularly in cases where a tool is analyzing files stored on a physical file system. For example, a file path might contain the account name of a developer.

Two URIs **SHALL** be considered equivalent if their normalized forms are the same, as described in [[RFC3986](#RFC3986)].

NOTE 2: For example, in the normalized form specified in RFC 3986:

* Percent-encoded characters use upper-case hexadecimal digits.
* Characters in the ALPHA and DIGIT ranges are not be percent-encoded, nor are hyphen, underscore, or tilde.
* The “:” delimiter is omitted if the port component of the authority is empty.
* In the host component, registered names and hexadecimal addresses use lower-case.

Aside from normalization, tools that produce SARIF files **SHALL NOT** make any other changes to the text of the URI; for example, they **SHALL NOT** convert the URI path to upper case or to lower case.

NOTE 3: This is especially important when the same SARIF file might be consumed on multiple platforms, for example, a platform such as Windows, whose NTFS file system is case-insensitive but case-preserving, and a platform such as Linux, whose file system is case-sensitive. Consider a scenario where a tool runs on a Windows system using NTFS, and the tool decides to lower-case the file names in the log. If the source files and the SARIF log were transferred to a Linux system, the URIs in the log file would not match the path names on the destination system.

## URI base id properties

Certain objects in this specification which have a URI-valued property (§3.2) also have a property that is described as being a “URI base id”. The value of such a property, if present, **SHALL** be a string which indirectly specifies the base URI for the file whose location is specified in the corresponding URI-valued property by a relative URI. If the URI-valued property contains an absolute URI, the URI base id property **SHALL** be absent. If the URI-valued property is absent, the URI base id property **SHALL** be absent.

If the consumer of the log file requires an absolute URI (for example, to display the specified file to a user), then the consumer must have the necessary information to resolve the value of the URI base id property to an absolute URI, which can then be combined with the relative URI stored in the URI-valued property.

The value of a URI base id property **MAY** be any string; it does not need to have any particular syntax or follow any particular naming convention. In particular, it does not need to designate a machine environment variable or similar value, although it may. The tool that produces the log file and any systems that consume the log file must agree on the meanings of any values for the URI base id property that appear in the log file.

EXAMPLE 1: In this example, the analysis tool has set the URI-valued property resultFile.uri (§3.19.2) to the relative URI of the file in which a result was detected. The tool has also set the value of the URI base id property resultFile.uriBaseId (§3.19.3) to "%srcroot%". The analysis tool and the log file consumers have agreed upon a convention whereby this indicates that the relative URI is expressed relative to the root of the source tree in which the file appears.

"resultFile": {

"uri": "drivers/video/hidef/driver.c",

"uriBaseId": "%srcroot%"

}

EXAMPLE 2: In this example, the analysis tool has set the URI-valued property analysisTarget.uri (§3.19.2) to the relative URI of the file which the tool was instructed to scan. The tool has also set the value of the URI base id property analysisTarget.uriBaseId (§3.19.3) to "$bindrop". The analysis tool and the log file consumers have agreed upon a convention whereby this indicates that the relative URI is expressed relative to the directory containing the binary files produced by a build.

"analysisTarget": {

"uri": "hidef.dll",

"uriBaseId": "$bindrop"

}

NOTE: There are various reasons for providing URI base id properties:

* Portability: A log file that contains relative URIs together with URI base id properties can be interpreted on a machine where the files are located at a different absolute location.
* Determinism: A log file that uses URI base id properties has a better chance of being “deterministic”; that is, of being identical from run to run if none of its inputs have changed, even if those runs occur on machines where the files are located at different absolute locations.
* Security: The use of URI base id properties avoids the persistence of absolute path names in the log file. Absolute path names can reveal information that might be sensitive.
* Semantics: Assuming the reader of the log file (an end user or another tool) has the necessary context, they can understand the meaning of the location specified by the "uri" property, for example, “this is a source file”.
* Brevity: The URI base id property might be shorter than the absolute path it represents.

## String properties

Unless otherwise specified in the description of a specific property, all properties whose values are of type "string" **MUST** have a non-empty value.

## Object properties

Certain properties in this specification are defined to be JSON objects whose property names satisfy certain conditions. Examples are the run.files property (§3.12.9) and the rule.messageFormats property (§3.27.8). Unless otherwise specified in the description of a specific property, if any such object is empty, then either the property **SHALL** be represented as an empty object {}, or it **SHALL** be absent.

## Array properties

Certain properties in this specification are defined to be JSON arrays. Examples are the run.toolNotifications property (§3.12.12) and the file.hashes property (§3.15.8). Unless otherwise specified in the description of a specific property, if any such array is empty, then either the property **SHALL** be represented as an empty array [], or it **SHALL** be absent.

## Property bags

### General

Certain properties in this specification are defined to be “property bags”. A property bag is a JSON object containing an arbitrary set of properties. The names of the properties **SHOULD** be camelCase strings, but see [Appendix D](#AppendixConverters) for exceptions. The values of the properties **MAY** be of any JSON type, including strings, numbers, arrays, objects, Booleans, and null. If the value of a property is a string, it **MAY** be an empty string.

### Tags

If a property bag contains a property with the name tags, then the value of that property **SHALL** be an array containing zero or more arbitrary strings, no two of which **SHALL** be the same. Two strings **SHALL** be considered the same if they consist of the same sequence of Unicode [[UNICODE10](#UNICODE10)] code points.

## Date/time properties

Certain properties in this specification specify a date and time. The value of every such property, if present, **SHALL** be a string in the following format, which is compatible with [[ISO86012004](#ISO86012004)]:

<dateTime>: <date>T<time>Z

<date>: YYYY-MM-DD

<time>: hh:mm:ss[.sss]

Here YYYY is a 4-digit year, MM is a 2-digit month from 01 to 12, DD is a 2-digit day from 01 to 31, the literal character “T” separates the date from the time, hh is a 2-digit hours from 00 to 23, mm is a 2-digit minutes from 00 to 59, ss is a 2-digit seconds from 00 to 59, [.sss] is an optional 3-digit number of milliseconds from 000 to 999, and the literal character “Z” specifies UTC time.

EXAMPLES:  
2016-02-08T16:08:25Z  
2016-02-08T16:08:25.943Z

## Array properties with unique values

Certain properties in this specification whose values are JSON arrays are described as having “unique” elements. When a property is so described, it shall mean that no two elements of the array **SHALL** have equal values. For purposes of this specification, two array elements **SHALL** be considered equal when they satisfy the condition for equality described in [[JSCHEMA01](#JSCHEMA01)], §4.3, “Instance equality”.

## Message properties

Certain properties in this specification are string values containing messages intended to be viewed by a user. No such property **SHALL** have a value that is the empty string.

In addition, such properties **SHOULD** conform to the following guidelines:

The message **SHOULD** be expressed as a single paragraph of plain text, consisting of one or more complete sentences, each ending with a period (or appropriate punctuation for the language in which the message is written). The message **SHOULD NOT** contain formatting information such as HTML tags. The message **SHOULD NOT** contain JSON escaped line breaks (\r or \n).

If the message consists of more than one sentence, the first sentence of the message **SHOULD** provide a useful summary of the message, suitable for display in cases where UI space is limited.

NOTE 1: If a tool does not construct the message in this way, the initial portion of the message that a viewer displays where UI space is limited might not be understandable.

NOTE 2: The rationale for these guidelines is that the SARIF format is intended to make it feasible to merge the outputs of multiple tools into a single user experience. A uniform approach to message authoring enhances the quality of that experience.

## sarifLog object

### General

A sarifLog object specifies the version of the file format and contains the output from one or more runs.

EXAMPLE:

{

"version": "1.0.0", # see §3.11.2

"runs":[ # see §3.11.4

{

... # a run object (see §3.12)

},

...

{

... # another run object

}

]

}

### version property

A sarifLog object **SHALL** contain a property named version whose value is a string designating the version of the SARIF format to which this log file conforms. This string shall have the value "1.0.0".

Although the order in which the name/value pairs appear in a JSON object value is not semantically significant, the version property **SHOULD** appear first.

NOTE: This will make it easier for parsers to handle multiple versions of the SARIF format, if new versions are defined in the future.

### $schema property

A sarifLog object **MAY** contain a property named $schema whose value is a string containing a URI from which a JSON schema document describing the version of the SARIF format to which this log file conforms can be obtained.

If the $schema property is present, the JSON schema obtained from the specified URI **MUST** describe the version of the SARIF format specified by the version property (§3.11.2).

NOTE: The purpose of the $schema property is to allow JSON schema validation tools to locate an appropriate schema against which to validate the log file. This is useful, for example, for tool authors who wish to ensure that logs produced by their tools conform to the SARIF format.

### runs property

A sarifLog object **SHALL** contain a property named runs whose value is an array of one or more run objects (§3.12).

## run object

### General

A run object describes a single run of an analysis tool, and contains the output of that run.

EXAMPLE:

{

"tool": # see §3.12.7

{

... # a tool object (see §3.13)

},

"results": # see §3.12.11

[

{

... # a result object (see §3.17)

},

...

{

... # another result object

}

]

}

### id property

A run object **MAY** contain a property named id whose value is a string which uniquely identifies the run.

NOTE: A result management system can use run.id to associate the information in the log with additional information not provided by the analysis tool that produced it.

### stableId property

A run object **MAY** contain a property named stableId whose value is a string containing a stable identifier for the run. Multiple runs of the same type may have the same stableId.

EXAMPLE:

{

"stableId": "Nightly security scanner run"

}

### baselineId property

A run object **MAY** contain a property named baselineId whose value is a string which **SHALL** match the id property (§3.12.2) of some previous run.

If the baselineId property is present, the result.baselineState property (§3.17.14) of every result object (§3.17) in the current run shall be computed with respect to the run specified by baselineId.

If the baselineId property is absent, and any result object has a value for its baselineState property, there must be out of band information available to determine the run with respect to which result.baselineState was computed.

### automationId property

A run object **MAY** contain a property named automationId whose value is a string containing an identifier that allows the run to be correlated with other artifacts produced by a larger automation process.

EXAMPLE: In an environment where an analysis tool is executed as part of an automated build process, the “build id” assigned by the build system might serve as the automationId, allowing the tool run to be associated with other artifacts produced by the build.

{

...

"runs": [

{

"automationId": "Build-14.0.1.2-20160518-15:48:02",

...

}

]

}

### architecture property

A run object **MAY** contain a property named architecture whose value is a string that specifies the hardware architecture at which the analysis targets are targeted. This does not need to be the same as the architecture on which the analysis tool is executed.

This specification does not specify a set of valid values for the architecture property.

EXAMPLE: An analysis tool running on a x86 architecture might be run once for a set of binaries that target x86, and then again for another set of binaries that target AMD64. The tool might set the architecture property for the first run to "x86", and for the second run to "AMD64".

### tool property

A run object **SHALL** contain a property named tool whose value is a tool object (§3.13) that describes the analysis tool that was run.

### invocation property

A run object **MAY** contain a property named invocation whose value is an invocation object (§3.14) that describes the invocation of the analysis tool that was run.

### files property

A run object **SHOULD** contain a property named files whose value is a JSON object, each of whose properties represents a file that was scanned in the course of the run.

The object specified by the files property **SHOULD** contain properties representing at least those files in which results were detected, but it **MAY** contain properties representing all files examined by the tool (whether or not results were detected in those files), or any subset of those files.

NOTE 1: file objects contain information that is useful for viewers. Viewers will be able to provide the most information to users if the files property is present and contains information for every file in which results were detected.

EXAMPLE 1:

"files": {

"file:///C:/Code/main.c": {

"mimeType": "text/x-c",

"hashes": [

{

"value": "b13ce2678a8807ba0765ab94a0ecd394f869bc81",

"algorithm": "sha256"

}

]

}

}

Each property name in the files object **SHALL** be the URI of a file examined by the tool. No two of these property names **SHALL** be equivalent as defined in §3.2. If the absolute location of the file is available, the URI **SHOULD** be an absolute URI; otherwise, the URI **SHALL** be a relative URI.

Each property value in the files object **SHALL** be a file object (§3.15) which contains information about the file identified by the URI in the property name.

In some cases, a file might be nested within another file (for example, a compressed container), referred to as its “parent.” A file that is not nested within another file is referred to as a “top-level file”. A file that is nested within another file is referred to as a “nested file”.

If the file is a nested file, then the property name **SHALL** be the URI of the outermost parent, together with a fragment that describes the nesting of the file within its parent or parents. The fragment shall be expressed as an absolute path; that is, it shall begin with a forward slash character (“/”).

EXAMPLE 2: Valid: The fragment is expressed as an absolute path:

"files": {

"file:///C:/bin/archive.zip#/images/grape.jpg": {

...

}

}

EXAMPLE 3: Invalid: The fragment is not expressed as an absolute path:

"files": {

"file:///C:/bin/archive.zip#images/grape.jpg": {

...

}

}

If the file is nested more than one level deep in the outermost parent, the fragments representing each level of nesting may be combined in any way desired, as long as no two of the resulting property names are equivalent as defined in §3.2.

NOTE 2: It does not need to be possible to use this URI to navigate directly to the nested file. The information necessary to do that is specified in the uri property (§3.15.2), or in the offset (§3.15.5) and length (§3.15.6) properties, of each file object.

EXAMPLE 4: Suppose a result is detected within a Flash object contained in a word processing document which is in turn contained in a compressed archive. Suppose the path to the word processing document within the compressed archive is /docs/intro.docx. Then one possible value for the property name within the files object would be:

file:///C:/Code/presentation.zip#/docs/intro.docx/Flash1

If the fragment contains any characters which cannot occur in a fragment as specified in [[RFC3986](#RFC3986)], those character shall be percent-encoded as specified in [[RFC3986](#RFC3986)].

EXAMPLE 5: Suppose a compressed container contains a file named /docs/chapter#1.doc. Then one possible value for the property name within the files property would be:

file:///C:/Code/presentation.zip#/docs/chapter%231.doc

The “#” character has been percent-encoded as %23.

EXAMPLE 6: This example shows a files property that represents a file nested two levels deep in its outermost container. The first level of nesting is specified by a path within a compressed container. The second level of nesting is specified by a byte offset from the start of the container, together with a length. See §3.15.

"files": {

"file:///C:/Code/app.zip": {

"mimeType": "application/zip",

},

"file:///C:/Code/app.zip#/docs/intro.docx": {

"uri": "/docs/intro.docx",

"mimeType": "application/vnd.openxmlformats-officedocument.wordprocessingml.document",

"parentKey": "file:///C:/Code/app.zip" # See §3.15.4

},

"file:///C:/Code/app.zip#/docs/intro.docx/Flash1": {

"offset": 17522,

"length": 4050,

"mimeType": "application/x-shockwave-flash",

"parentKey": "file:///C:/Code/app.zip#/docs/intro.docx"

}

}

### logicalLocations property

Depending on the circumstances, a run object either **MAY** or **SHOULD** contain a property named logicalLocations whose value is an object, each of whose properties represents the logical location of one or more results detected in the course of the run.

If the tool has source location information available, and therefore can produce result objects with physical location information (such as the source file name, line, and column), the logicalLocations property **MAY** be present.

If the tool does not have source location information available, and therefore can only produce result objects with logical location information (such as a namespace, type, and method name), the logicalLocations propertys **SHOULD** be present.

With one exception described in §3.18.6, each property name in the logicalLocations object **SHALL** be a string representing the logical location where the result was detected, in a format consistent with the programming language in which the programmatic construct specified by that logical location was expressed. We refer to this string as a “fully qualified logical name”. See §3.18.5 for examples.

Each value in the object specified by the logicalLocations property shall be a logicalLocation object (§3.21).

In some cases, a logical location might be nested within another logical location (for example, a class nested within a namespace), referred to as its “parent.” A logical location that is not nested within another logical location is referred to as a “top-level logical location”. A logical location that is nested within another logical location is referred to as a “nested logical location”.

If a result is detected in a nested logical location, then the logicalLocations object **SHALL** contain properties describing not only that logical location, but also properties describing each of its parents, up to and including the top-level logical location.

EXAMPLE: In this example, a result was detected in the C++ class namespaceA::namespaceB::classC. The logicalLocations object contains not only a property describing the class, but also properties describing its parents.

"logicalLocations": {

"namespaceA::namespaceB::classC": {

"name": "classC",

"kind": "type",

"parentKey": "namespaceA::namespaceB"

},

"namespaceA::namespaceB": {

"name": "namespaceB",

"kind": "namespace"

"parentKey": "namespaceA"

},

"namespaceA": {

"name": "namespaceA",

"kind": "namespace"

}

}

NOTE: The detailed information in logicalLocations is useful, even though much of it is captured in the location.fullyQualifiedLogicalName property (§3.18.5), because it allows results management systems and other programs to organize analysis results, for example, by asking questions such as “How many results were found in the namespace namespaceA::namespaceB?”. Programs can ask these questions without having to know how to parse the fullyQualifiedLogicalName string.

### results property

If the analysis tool was run with the intent of scanning files and producing results, then the run object **SHALL** contain a property named results whose value is an array containing zero or more unique (§3.9) result objects (§3.17), each of which represents a single result detected in the course of the run.

The results array **SHALL** be empty if the tool invocation that produced the run object did not detect any results.

If the tool was run solely for the purpose of exporting rule metadata (see §3.12.14), the results property **SHALL** be absent.

### toolNotifications property

A run object **MAY** contain a property named toolNotifications whose value is an array of zero or more notification objects (§3.32). Each element of the array represents a runtime condition detected by the tool. The presence within this array of any notification object whose level property (§3.32.7) is "error" **SHALL** mean that the run failed.

NOTE 1: The information in toolNotifications is primarily intended for the developers of the analysis tool, to aid them in diagnosing bugs in the tool. This is in contrast to the information in results, which is intended for the developers of the code being analyzed. However, viewers may still present tool notifications to users, so users are aware of any tool problems. At a minimum, viewers should make users aware of tool notifications whose level property is "error".

NOTE 2: Depending on the nature of the error, a tool that encounters a runtime error might or might not be able to continue running.  
  
If the error occurs in the course of evaluating a rule, the tool might report the error in toolNotifications, disable the rule, and continue to execute the remaining rules.  
  
If the error occurs outside of the evaluation of a rule, the tool might report the error in toolNotifications and then halt. If the tool exits abnormally, it might not have the opportunity to report the error.

### configurationNotifications property

A run object **MAY** contain a property named configurationNotifications whose value is an array of zero or more notification objects (§3.32). Each element of the array represents a condition relevant to the tool's configuration. The presence within this array of any notification object whose level property (§3.32.7) is "error" **SHALL** mean that the run failed.

NOTE 1: The information in configurationNotifications is primarily intended for the engineers who configure the analysis tool, to aid them in diagnosing errors in the configuration. This is in contrast to the information in results, which is intended for the developers of the code being analyzed. However, viewers may still present configuration notifications to users, so users are aware of any configuration problems. At a minimum, viewers should make users aware of configuration notifications whose level property is "error".

NOTE 2: Many tools can be parameterized with information about which rules to run, and how those rules should be configured. In some cases, if the configuration information is invalid, the tool can ignore the invalid information and continue to run.

EXAMPLE 1: A tool is invoked with a configuration file which specifies that the tool should disable rule ABC0001, but there is no rule whose id is ABC0001. The tool should report the problem in configurationNotifications. The tool might continue to run, reporting results for the rules that are correctly configured.

"configurationNotifications": [

{

"id": "UnknownRule",

"ruleId": "ABC0001",

"level": "warning",

"message": "Could not disable rule \"ABC0001\"

because there is no rule with that id."

}

]

EXAMPLE 2: A tool is invoked with an unknown command-line argument. The tool should report the problem in configurationNotifications. The tool might report the problem as a warning and continue to run, or it might report the problem as an error and terminate.

"configurationNotifications": [

{

"id": "UnknownCommandLineArgument",

"level": "error",

"message": "Command line argument \"/X\" is unknown."

}

]

EXAMPLE 3: A tool is invoked with a command-line argument that specifies the name of a directory containing files to analyze, but the user who invoked the tool does not have read access to that directory. The tool should report the problem as an error in configurationNotifications and then terminate.

"configurationNotifications": [

{

"id": "CannotLocateRulePlugin",

"level": "error",

"message": "Cannot locate rule plugin

\"C:\\AnalysisTool\\CustomChecks.dll."

}

]

### rules property

Depending on the circumstances, a run object either **SHALL** or **MAY** contain a property named rules whose value is a JSON object, each of whose properties represents an analysis rule. If the tool was run solely for the purpose of exporting rule metadata, the rules property **SHALL** be present. Otherwise, the rules property **MAY** be present.

Each property value within the rules property **SHALL** be a rule object (§3.27).

If there is only one rule object with a particular id (§3.27.3), then the property name for that rule object **SHALL** be the rule id.

EXAMPLE 1: In this example, two rules have different ids. The property names match the rule ids.

"rules": {

"CA1001": {

"id": "CA1001",

"shortDescription": "Types that own disposable fields should be

disposable."

},

"CA1002": {

"id": "CA1002",

"shortDescription": "Do not expose generic lists."

}

}

Some tools use the same rule id to refer to multiple distinct (although logically related) rules. In that case, the property names for those rule objects **SHALL** be distinct, even though the rule ids are the same. The property names **SHOULD** be clearly related to the rule id.

EXAMPLE 2: In this example, two distinct but related rules have the same rule id. The property names are distinct, and are clearly related to the rule id.

"rules": {

"CA1711-1": {

"id": "CA1711",

"messageFormats": {

"default": "Rename type name {0} so that it does not end in '{1}'"

}

},

"CA1711-2": {

"id": "CA1711",

"messageFormats": {

"default": "Either replace the suffix '{0}' in member name '{1}' with

the suggested numeric alternate or provide

a more meaningful suffix"

}

}

}

NOTE: This property is a dictionary, rather than simply an array of rule objects, to facilitate looking up the rule associated with each result object (§3.17) by means of the result's ruleId property (§3.17.2) or ruleKey property (§3.17.3).

### properties property

A run object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the run that is not explicitly specified in the SARIF format.

## tool object

### General

A tool object contains information describing the analysis tool that was run.

NOTE: If another tool post-processes the log file (for example, by removing certain results, or by adding information that was not known to the analysis tool), the post-processing tool should not alter any part of the tool object.

EXAMPLE:

{

"name": "CodeScanner", # see §3.13.2

"fullName": "CodeScanner 1.1, Developer Preview (en-US)", # see §3.13.3

"semanticVersion": "1.1.2-beta.12", # see §3.13.4

"version": "1.1.2b12, # see §3.13.5

"fileVersion": "1.1.1502.2" # see §3.13.6

}

### name property

A tool object **SHALL** contain a property named name whose value is a string containing the name of the tool that produced the log file.

EXAMPLE: "CodeScanner"

### fullName property

A tool object **MAY** contain a property named fullName whose value is a string containing the name of the tool along with its version and any other useful identifying information, such as its locale.

EXAMPLE: "CodeScanner 1.1, Developer Preview (en-US)"

### semanticVersion property

In a log file produced by an analysis tool, a tool object **SHALL** contain a property named semanticVersion whose value is a string containing the tool version in the format specified by SemVer ([[SEMVER](#SEMVER)]).

EXAMPLE 1:

"tool": {

"semanticVersion": "1.1.2-beta.12"

}

NOTE 1: Semantic versions have the property of being sortable in chronological order of release. The presence of the semanticVersion property allows results management systems to (for example) restrict the results they display to versions newer than a specified version, or to restrict the results to a particular major version.

If the tool does not natively present its version string in SemVer format, it **SHALL** synthesize a SemVer string to populate the semanticVersion property.

EXAMPLE 2: Suppose an analysis tool natively presents its version string as "2.0" (no “patch level” is available). The tool might synthesize a SemVer string "2.0.0".

EXAMPLE 3: Suppose an analysis tool natively presents its version string as "1.1.2b12" (the “pre-release” information is not in SemVer format). The tool might synthesize a SemVer string "1.1.2-beta.12".

In a log file produced by a conversion tool, the semanticVersion property **SHALL** be absent.

NOTE: The rationale is that an analysis tool knows whether its version string is intended to be interpreted according to SemVer. A converter will in general not know this, even if the tool's version string conforms to the pattern specified by SemVer.

### version property

In a log file produced by an analysis tool, a tool object **MAY** contain a property named version whose value is a string containing the tool version in whatever format the tool natively provides.

In a log file produced by a converter, the version property **SHALL** be present.

### fileVersion property

If the operating system on which the tool runs provides a value for the file version of the tool's primary executable file, then the tool object **MAY** contain a property named fileVersion whose value is a string representation of that file version. If the operating system does not provide such a value, the fileVersion property **SHALL** be absent.

EXAMPLE: On the Windows platform, this information is available in the FILEVERSION member of the VERSIONINFO structure.

### language property

A tool object **SHOULD** contain a property named language whose value is a string specifying the language of the messages produced by the tool, in the format specified by [[RFC5646](#RFC5646)].

EXAMPLE 1: The tool language is English:

"tool": {

"language": "en"

...

}

EXAMPLE 2: The tool language is French as spoken in France:

"tool": {

"language": "fr-FR"

}

### sarifLoggerVersion property

If the tool that produced the log relied on another software component to generate the log, then the tool object **SHOULD** contain a property named sarifLoggerVersion whose value is a string specifying the version of the logging component.

NOTE: This information is useful, for example, when a tool produces invalid output, and the author of the tool wishes to file a bug report with the author of the logging component. In this case, it is helpful to the author of the logging component to know the precise version number of the logging component that produced the invalid output.

### properties property

A tool object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the themselves that is not explicitly specified in the SARIF format.

## invocation object

### General

An invocation object contains information describing the invocation of the analysis tool that was run.

### commandLine property

An invocation object **MAY** contain a property named commandLine whose value is a string containing the completely specified command line used to invoke the tool, starting with the name of the tool's executable or script file, optionally qualified by the relative or absolute path to the file.

NOTE 1: The information in the commandLine property makes it possible to precisely repeat a run of an analysis tool, and to verify that the results reported in the log file were generated by an appropriate invocation of the tool.

If the information in commandLine contains information which should not be disclosed, such as passwords, tokens, database connection strings, or in some circumstances even the fully qualified path to the tool's executable or script file, that information **SHOULD** be redacted or omitted. Redacted information **SHOULD** be replaced with the token [REMOVED].

NOTE 2: Redacting sensitive information from commandLine makes it more difficult to precisely reproduce an analysis run. The value of commandLine would have to be combined with information from another source to allow the run to be repeated.

EXAMPLE 1: Suppose a tool is invoked with the command line  
  
C:\Users\mary\Tools\DbScanner.exe /ConnectionString  
 "Server=Corp;Db=Accounting;User=Admin;Password=S3cr#t"  
 /input \*.sql  
  
Then the value of the commandLine property might contain the redacted command line  
  
[REMOVED]\DbScanner.exe /connectionString=[REMOVED] /input=\*.sql

The commandLine property might describe a command that would be harmful if it were executed. For this reason, the recipient of a SARIF log file from an untrusted source should not execute the command line without first examining it carefully. In particular, an automated system should not execute a command line in a SARIF log file from an untrusted source.

EXAMPLE 2: An example of a harmful command line:

"invocation": {

"commandLine": "rm -rf /"

}

### responseFiles property

An invocation object **MAY** contain a property named responseFiles whose value is an object, each of whose properties represents the contents of a response file specified on the tool's command line.

Each property name in the object shall be the URI of a response file specified on the tool's command line. If the absolute location of the file is available, the URI **SHOULD** be an absolute URI; otherwise, the URI **SHALL** be a relative URI.

Each property value in the object **SHALL** be a string containing the textual contents of the file specified by the property name. If the file has zero length, the value **SHALL** be an empty string. Characters that cannot appear directly in a JSON string **SHALL** be escaped as specified in the JSON specification.

EXAMPLE:

"invocation": {

"commandLine": "/quiet @analyzer.rsp @strict.rsp" @options.rsp,

"responseFiles": {

"analyzer.rsp": "/rules:basic\n/out:analyzer.sarif",

"strict.rsp": "/rules:security /rules:reliability",

"options.rsp": ""

}

...

}

### startTime property

An invocation object **MAY** contain a property named startTime whose value is a string specifying the date and time at which the run started. The string **SHALL** be in the format specified by (§3.8).

### endTime property

An invocation object **MAY** contain a property named endTime whose value is a string specifying the date and time at which the run ended. The string **SHALL** be in the format specified by (§3.8).

### machine property

An invocation object **MAY** contain a property named machine whose value is a string containing the name of the machine on which the tool was run.

### account property

An invocation object **MAY** contain a property named account whose value is a string containing the name of the account under which the tool was run.

### processId property

An invocation object **MAY** contain a property named processId whose value is an integer containing the id of the process in which the tool was run.

### fileName property

An invocation object may contain a property named fileName whose value is a string containing the fully qualified path name of the tool's executable file.

NOTE 1: This property is defined in the invocation object rather than in the tool object (§3.13) because the identical tool might be invoked from different paths on different machines.

NOTE 2: This property might duplicate information in the commandLine property (§3.14.2). It is necessary because the command line might not explicitly specify the path to the tool (for example, if the tool directory is on the execution path), and this information is important for troubleshooting.

NOTE 3: Absolute path names can reveal information that might be sensitive.

### workingDirectory property

An invocation object **MAY** contain a property named workingDirectory whose value is a string containing the fully qualified path name of the directory in which the analysis tool was invoked.

NOTE: Absolute path names can reveal information that might be sensitive.

### environmentVariables property

An invocation object **MAY** contain a property named environmentVariables whose value is an object. The property names in this object **SHALL** contain the names of all the environment variables in the tool's execution environment. The value of each property **SHALL** be a string containing the value of the specified environment variable. If the value of the environment variable is an empty string, the value of the corresponding property **SHALL** be an empty string.

NOTE 1: Environment variable names and values are likely to reveal highly sensitive information. For example, on a Windows machine, environment variables reveal the directories on the execution path, user account name, machine name, logon domain controller, *etc.*

NOTE 2: The result of setting an environment variable to an empty string is operating system-dependent. On Windows, it removes the variable from the environment. In Unix, an environment variable can have an empty value.

### properties property

An invocation object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the tool invocation that is not explicitly specified in the SARIF format.

## file object

### General

A file object represents a single file.

### uri property

Depending on the circumstances, a file object either **SHALL**, **MAY**, or **SHALL NOT** contain a property named uri whose value is a string containing a valid URI (§3.2).

If the file object represents a top-level file, then the uri property **MAY** be present. If present, it **SHALL** be equal to the name of the property within run.files (§3.12.9) whose value is this file object. If absent, it **SHALL** be interpreted as having that same value.

If the file object represents a nested file whose location relative to its parent can be expressed only by means of a path, then the uri property **SHALL** be present, and its value **SHALL** be a valid relative URI expressing that path.

If the file object represents a nested file whose location within its parent can be expressed only by a byte offset from the start of the parent, and not by means of a path, then the uri property **SHALL** be absent.

If the file object represents a nested file whose location within its parent can be expressed either by means of a path or by means of a byte offset from the start of the parent, then either the uri property or the offset property (§3.15.5) or both **SHALL** be present; they **SHALL NOT** both be absent. If the uri property is present, its value **SHALL** be a valid relative URI expressing the path of the nested file within the parent.

EXAMPLE 1: The uri property of the top-level file repeats the property name. The uri property of the nested file specifies the relative URI of the nested file with respect to its parent.

"files": {

"http://www.example.com/a.zip": {

"uri": "http://www.example.com/a.zip",

"mimeType": "application/zip"

},

"http://www.example.com/a.zip#/src/file.c": {

"uri": "/src/file.c",

"mimeType": "x-c",

"parentKey": "http://www.example.com/a.zip" # See §3.15.4

}

}

EXAMPLE 2: The uri property of the top-level file is omitted. It is interpreted as "http://www.example.com/a.zip".

"files": {

"http://www.example.com/a.zip": {

"mimeType": "application/zip"

},

"http://www.example.com/a.zip#/src/file.c": {

"uri": "/src/file.c",

"mimeType": "x-c",

"parentKey": "http://www.example.com/a.zip"

}

}

The value of the uri property for a nested file does not need to match the value of the fragment portion of the URI specified in the property name. This allows multiple levels of nesting to be represented.

EXAMPLE 3: There are two levels of nesting. The uri property of the most deeply nested file does not match the fragment portion of the URI specified in the property name.

"files": {

"http://www.example.com/a.zip": {

"mimeType": "application/zip"

},

"http://www.example.com/a.zip#/media/b.zip": {

"uri": "/media/b.zip",

"mimeType": "application/zip",

"parentKey": "http://www.example.com/a.zip"

},

"http://www.example.com/a.zip#/media/b.zip/images/c.png": {

"uri": "/images/c.png",

"mimeType": "image/png",

"parentKey": "http://www.example.com/a.zip#/media/b.zip"

}

}

### uriBaseId property

If the uri property (§3.15.2) is present and contains a relative URI, then the file object **MAY** contain a property named uriBaseId whose value is a string containing a URI base id (§3.3) which indirectly specifies the absolute URI with respect to which uri **SHALL** be interpreted.

If the uri property is absent or contains an absolute URI, then the uriBaseId property **SHALL** be absent.

### parentKey property

If the file represented by the file object is a nested file, then the file object **SHALL** contain a property named parentKey whose value is a string containing a URI that matches the property name of the parent file's file object within run.files (§3.12.9).

If the file represented by the file object is a top-level file, then the parentKey property **SHALL** be absent.

NOTE: The presence of the parentKey property makes it possible to navigate from the file object representing a nested file to the file objects representing each of its parent files in turn, up to the top-level file. It is necessary because the URI specified by a file object's property name within run.files does not necessarily contain enough information to do so.

### offset property

Depending on the circumstances, a file object either **SHALL**, **MAY**, or **SHALL NOT** contain a property named offset whose value is a non-negative integer.

If the file object represents a top-level file, then the offset property **SHALL** be absent.

If the file object represents a nested file whose location relative to its parent can be expressed only by means of a byte offset from the start of its parent file, then the offset property **SHALL** be present, and its value **SHALL** be that byte offset.

If the file object represents a nested file whose location within its parent can only be expressed by means of a path, and not by means of a byte offset from the start of the parent, then the offset property **SHALL** be absent.

If the file object represents a nested file whose location within its parent can be expressed either by means of a path or by means of a byte offset from the start of the parent, then either the uri property (§3.15.2) or the offset property or both **SHALL** be present; they **SHALL NOT** both be absent. If the offset property is present, its value shall be that byte offset.

### length property

A file object **MAY** contain a property named length whose value is a non-negative integer specifying the length of the file in bytes.

### mimeType property

A file object **SHOULD** contain a property named mimeType whose value is a string that specifies the MIME type [[RFC2045](#RFC2045)] of the file.

### hashes property

A file object **MAY** contain a property named hashes whose value is an array of unique (§3.9) hash objects (§3.16), each of which specifies a hashed value for the file specified by the file object, along with the name of the algorithm used to compute the hash.

If present, the array specified by hashes **SHALL NOT** be empty.

NOTE: A hash value for an analysis target can be useful when a log file is processed by a result management system. The value can be used as a key when persisting results in a database. This allows a build system to use cached results, rather than repeating the analysis, when a target has not changed. A file hash can also be useful for validating results in a policy compliance system, allowing an auditor to validate that rerunning analysis against a target that hashes to a specific value reproduces the provided results.  
  
The file object defines an array of hash values, rather than a single hash value, to allow a log file to be consumed by multiple tool chains that might expect hash values produced by differing algorithms. Compliance systems, for example, will favor the use of secure hash algorithms (such as SHA-256) that minimize the possibility that two different targets will produce the same hash (at the expense of speed to produce the hash). In situations where compliance and security are not a concern, a system might prefer to use a fast hash algorithm (such as MD5 or SHA-1) that occasionally produces hash collisions.  
  
To populate the hashes property, an analysis tool must support the ability to produce hashes for its analysis targets. Alternatively, the hashes could be added to the log file as a post-processing step.  
  
To make the best use of such an analysis tool, a user (such as a build engineer) would determine what systems in their build environment will consume the log file. The user would then configure the tool to produce hashes using the algorithms required by those systems. Analysis tools that are configurable to produce hashes with a variety of commonly used algorithms will interoperate most easily with such systems.

### contents property

A file object **MAY** contain a property named contents whose value **SHALL** be a string representation of the contents of the file.

If the file object represents a binary file, the value of the contents string **SHALL** be the MIME Base64 encoding of the bytes contained in the file.

If the file object represents a text file, the value of the contents string shall be computed by first encoding the characters in the file to UTF-8, and then encoding the resulting byte sequence with MIME Base64.

### properties property

A file object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the file that is not explicitly specified in the SARIF format.

## hash object

### General

A hash object represents a hash value of some file or collection of files, together with the algorithm used to compute the hash.

EXAMPLE:

{

"value":"b13ce2678a8807ba0765ab94a0ecd394f869bc81", # see §3.16.2

"algorithm":"sha256" # see §3.16.3

}

### value property

A hash object **SHALL** contain a property named value whose value is a string representation of the hash value of some file or collection of files, computed by the algorithm named in the algorithm property (§5.16.3).

NOTE: The value is represented as a string because hash values are typically represented in hexadecimal notation, and JSON integer values must be decimal.

### algorithm property

A hash object **SHALL** contain a property named algorithm whose value is a string specifying the name of the algorithm used to compute the hash value specified in the value property (§3.16.2). This string shall be one of the following:

* "authentihash"
* "blake256"
* "blake512"
* "ecoh"
* "fsb"
* "gost"
* "groestl"
* "has160"
* "haval"
* "jh"
* "md2"
* "md4"
* "md5"
* "md6"
* "radioGatun"
* "ripeMD"
* "ripeMD128"
* "ripeMD160"
* "ripeMD320"
* "sdhash"
* "sha1"
* "sha224"
* "sha256"
* "sha384"
* "sha512"
* "sha3"
* "skein"
* "snefru"
* "spectralHash"
* "ssdeep"
* "swifft"
* "tiger"
* "tlsh"
* "whirlpool"

## result object

### General

A result object describes a single result detected by an analysis tool.

### ruleId property

Depending on the circumstances, a rule object either **SHALL** or **SHALL NOT** contain a property named ruleId whose value is a string containing the stable, opaque identifier for the rule that was evaluated to produce the result.

EXAMPLE:

"results": [

{

"ruleId": "CA2101"

...

}

]

If the log was created by an analysis tool (as opposed to a conversion tool), then ruleId **SHALL** be present.

Not all existing analysis tools emit the equivalent of a ruleId in their output. A conversion tool which converts the output of such an analysis tool to the SARIF format **SHALL NOT** set the ruleId property, and in particular, it **SHALL NOT** attempt to synthesize it from other information available in the original analysis tool's output.

### ruleKey property

If there is more than one rule with the id specified by the ruleId property (§3.17.2), and if the run object (§3.12) in which this result occurs contains a rules property (§3.12.14), then the result object **SHALL** contain a property named ruleKey whose value is a string that matches one of the property names in the run.rules object.

The value of the ruleId property on this result object must match the id property (§3.27.3) of the rule object (§3.27) identified by ruleKey.

EXAMPLE: In this example, there is more than one rule with id CA1711. When the log includes a result with that rule id, it provides a value for ruleKey to specify which of the rules with that id is meant.

"runs": [

{

"results": [

{

"ruleId": "CA1711", # Matches the "id" value of the specified

# property value within "rules"

"ruleKey": "CA711-1" # Specifies a property name within "rules".

}

],

"rules": {

"CA1711-1": {

"id": "CA1711"

},

"CA1711-2": {

"id": "CA1711"

}

}

}

]

### level property

A result object **MAY** contain a property named level whose value is one of a fixed set of strings that specify the severity level of the result.

If present, the level property **SHALL** have one of the following values, with the specified meanings:

* "pass": The rule specified by the ruleId property (§3.17.2) was evaluated, and no problem was found.
* "warning": The rule specified by the ruleId property was evaluated, and a problem was found.
* "error": The rule specified by the ruleId property was evaluated, and a serious problem was found.
* "notApplicable": The rule specified by the ruleId property was not evaluated, because it does not apply to the file specified by analysisTarget (§3.18.3).

EXAMPLE 1: In this example, a binary checker has a rule that applies to 32-bit binaries only. It produces a "notApplicable" result if it is run on a 64-bit binary:

"results": [

{

"ruleId": "ABC0001",

"level": "notApplicable",

"message": "\"MyTool64.exe\" was not evaluated for rule ABC0001

because it is not a 32-bit binary."

"locations": [

{

"analysisTarget": {

"uri": "file://C:/bin/MyTool64.exe"

}

}

]

}

]

* "note": A purely informational log entry.  
    
  The ruleId property for a result object whose level property is "note" **MAY** be present, if the note relates to a particular rule; otherwise ruleId **MAY** be absent.

EXAMPLE 2: In this example, the tool reports an observation about the code that does not represent a problem.

"results": [

{

"ruleId": "ABC0002",

"level": "note",

"message": "Consider using 'nameof(start)' instead of hard-coding

the parameter name 'start'."

"locations": [

{

"analysisTarget": {

"uri": "file:///C:/code/a.cs",

"region": {

"startLine": 6

}

}

}

]

}

]

EXAMPLE 3: In this example, the tool reports information that is relevant to a particular rule, but does not represent an observation about the code.

"results": [

{

"ruleId": "ABC0003",

"level": "note",

"message": "A new version of rule ABC0001 is available."

}

]

EXAMPLE 4: In this example, the tool reports information that is not related to any particular rule, and is not an observation about the code.

"results": [

{

"level": "note",

"message": "Version 11.0 of SuperLint is now available."

}

]

If the level property is absent, its value **SHALL** be considered to be the value of the defaultLevel property (§3.27.7) of the rule object (§3.27) specified by this result object's ruleId property (§3.17.2) or ruleKey property (§3.17.3).

In that case, if the run object (§3.12) containing this result does not include a rules property (§3.12.14), or if the run.rules property does not specify information for the rule associated with this result, or if the rule object associated with this result does not specify a defaultLevel property, then the value of the level property **SHALL** be considered to be "warning".

### message property

A result object **SHALL** contain a property named message whose value is a string that describes the result.

The message property **SHOULD** conform to the guidelines for message properties (§3.10).

The message property **SHOULD** provide sufficient details to allow an end user to resolve any problem that the result might indicate. In particular, message **SHALL** include all of the following information that is available and relevant to the result:

* Information sufficient to identify the analysis target, and the location within the target where the problem occurred.
* The condition within the analysis target that led to the problem being reported.
* The risks potentially associated with not fixing the problem.
* The full range of responses to the problem that the end user could take (including the definition of conditions where it might be appropriate not to fix the problem, or to conclude that the result is a false positive).

EXAMPLE: This is an example of a message:

"results": [

{

"message": "Deleting member 'x' of variable 'y' may compromise

performance on subsequent accesses of 'y'. Consider

setting object member 'x' to null instead, unless this

object is a dictionary or if runtime semantics otherwise

dictate that the existence of a null member is distinct

from one that is not present at all. This violation can

also be ignored for infrequently called code paths."

}

]

### formattedRuleMessage property

A result object **MAY** contain a property named formattedRuleMessage whose value is a formattedMessage object (§3.28) that can be used to construct a formatted message that describes the result.

If the formattedRuleMessage property is present on a result, the message property (§3.17.5) **SHALL** be absent. If the message property is present on a result, the formattedRuleMessage property **SHALL** be absent.

### locations property

A result object **SHOULD** contain a property named locations whose value is an array of one or more unique (§3.9) location objects (§3.18), each of which specifies a location where the result occurred.

NOTE: In rare circumstances, it might not be possible to specify a location for a result. However, locations is very valuable information for anyone who needs to diagnose and correct the condition described by the result, so the authors of analysis tools should make every effort to provide it.

EXAMPLE 1: If a C++ analyzer detects that no file defines a global function main, then that result cannot be associated with a file.

The locations array **SHALL NOT** contain more than one element unless the condition indicated by the result, if any, can only be corrected by making a change at every location specified in the array.

EXAMPLE 2: In C#, which support “partial” classes, portions of the declaration of a single class can occur at multiple locations in the source code. If an analysis tool reports that the name of such a class does not conform to a specified convention, then the resulting log file should contain a single result object, which should contain a locations array each of whose elements specifies a location in the source code where the class name occurs.

The locations array **SHALL NOT** be used to specify distinct occurrences of the same result, which can be corrected independently.

EXAMPLE 3: Consider an analysis tool which locates misspelled words in documentation, and suppose this tool scans a document in which the same word is misspelled in two distinct locations. Then the resulting log file must contain two distinct result objects, each of which contains a locations array containing a single location object specifying the location of one instance of the misspelled word.  
  
In contrast, consider a tool which locates misspelled words in variable names. If the tool detects a misspelled variable name, it must produce a single result object whose locations array contains the location of every reference to the variable, since fixing some but not all of the references would cause a compilation error.

### snippet property

A result object **MAY** contain a property named snippet whose value is a string containing a source code or other file fragment that illustrates the result, for example, the text of the source code line on which the result was detected, or a small range of lines surrounding the result location.

### toolFingerprintContribution property

A result object **MAY** contain a property named toolFingerprintContribution whose value is a string that contributes to the unique identity of the result. [Appendix B](#AppendixFingerprints) explains how a result management system can use this value.

### codeFlows property

A result object **MAY** contain a property named codeFlows whose value is an array of one or more unique (§3.9) codeFlow objects (§3.22). The codeFlows property is intended for use by analysis tools that provide execution path details that illustrate a possible problem in the code. We refer to this execution path as a code flow. Each codeFlow object in the codeFlows array **SHALL** describe a single code flow.

NOTE: The SARIF file format allows multiple codeFlow objectss within a single result object to allow for the possibility that more than one path through the program might be relevant to a single result.

### stacks property

A result object **MAY** contain a property named stacks whose value is an array of one or more unique (§3.9) stack objects (§3.23). The stacks property is intended for use by analysis tools that collects call stack information in the process of producing results.

NOTE: The SARIF file format allows multiple stack objects within a single result object to allow for the possibility that more than one call stack might be relevant to a single result.

### relatedLocations property

A result object **MAY** contain a property named relatedLocations whose value is an array of one or more unique (§3.9) annotatedCodeLocation objects (§3.25), each of which represents a location relevant to understanding the result.

EXAMPLE: Suppose that a tool for analyzing JavaScript has a rule that reports a problem when a variable declared in an inner scope hides a variable with the same name in an enclosing scope. The tool would report the problem on the line where the inner variable is declared. The tool could choose to add an element to the relatedLocations array, specifying the location where the outer variable was declared.  
  
The result might appear in the log file like this:

results: [

{

"ruleId": "JS3056",

"level": "error",

"message": "Name 'index' cannot be used in this scope because

it would give a different meaning to 'index'.",

"locations": [

{

"analysisTarget": {

"uri": "file:///C:/Code/a.js",

"region": {

"startLine": "6",

"startColumn": "10"

}

}

}

],

"relatedLocations": [ # An array of annotatedCodeLocation objects

# (§3.25)

{

"message": "The previous declaration of 'index' was here.",

"physicalLocation": {

"uri": "file:///C:/Code/a.js",

"region": {

"startLine": "2",

"startColumn": "6"

}

}

}

]

},

...

]

The tool might write messages to the console like this:

C:\Code\a.js(6,10-10) : error : JS3056: Name 'index' cannot be used in this scope because it would give a different meaning to 'index'.

C:\Code\a.js(2,6-6) : info : JS3056: The previous declaration of 'index' was here.

### suppressionStates property

#### General

A result object **MAY** contain a property named suppressionStates whose value is an array of unique (§3.9) strings. This property **SHALL** be present if and only if the analysis tool that produced the log file wishes to convey the information that the condition described by the result object should be “suppressed”.

NOTE: The treatment of “suppressed” results depends on the development environment within which the log file is used, for example, a build system, an integrated development environment (IDE), or a result management system. Typically, development environments do not expose suppressed results to the user. For example, they do not include them in build log files, display them in error lists, or include them in bug counts.

If present, this property conveys the reason or reasons that the result has been suppressed. The supported reasons for suppressing a result are:

* The developer has suppressed the result in the source code (see §3.17.13.2).
* The result is marked as suppressed in an external store such as a database (see §3.17.13.3).

#### suppressedInSource value

Some programming languages offer a syntactic construct for suppressing compiler warnings.

EXAMPLE: In C#, #pragma warning is such a construct.

For tools that examine source code written in such a language, the suppressionStates array **SHALL** include the value "suppressedInSource" if the tool determines that the result occurred at a location within the scope of an instance of such a construct which is intended to suppress that particular class of result. If the tool determines that the result did not occur at such a location, or if the tool cannot or chooses not to determine whether the result occurred at such a location, or if the tool examines source code written in a language that lacks such a construct, the suppressionStates array **SHALL NOT** include the value "suppressedInSource".

#### suppressedExternally value

Some development environments provide a persistent store, for example a database, containing historical information about the results from static analysis tools. Such a store might offer the ability to mark a result as “suppressed,” meaning that if the result is encountered again, it should be ignored.

When a tool with access to such a database detects such a result, it **MAY** choose not to add the result to the log. If the tool does include such a result in the log, the suppressionStates array **SHALL** include the value "suppressedExternally".

If the tool does not have access to a database of suppression information, or if the tool does have access to such a database and determines that the result is not marked for suppression in that database, then the suppressionStates array **SHALL NOT** include the value "suppressedExternally".

### baselineState property

A result object **MAY** contain a property named baselineState whose value is a string that specifies the state of this result with respect to some previous run. We refer to this previous run as the “baseline”.

If the run.baselineId property (§3.12.4) of the current run is present, the baseline **SHALL BE** the run specified by run.baselineId.

If the run.baselineId property of the current run is absent, then there **MUST** be out of band information available to determine the baseline.

This property **SHALL** have one of the following values, with the specified meanings:

* "new": This result was detected in the current run but was not detected in the baseline.
* "existing": This result was detected both in the current run and in the baseline.
* "absent": This result was detected in the baseline but was not detected in the current run.

If the run.baselineId property is present but the baselineState property is absent, the baselineState property **SHALL** be considered to have the value "new".

NOTE: The purpose of the baselineState property is to allow (for example) a measurement of how many new results were introduced in the run, and how many previously existing results no longer appear.  
  
To assign a value to baselineState, a tool must have a way to determine whether a result is “the same”, in some sense, as a result that appeared in the baseline. [Appendix B](#AppendixFingerprints) discusses how a result management system can assign a “fingerprint” to each result. An analysis tool that works together with such a result management system can use the fingerprint to determine whether two results are the same; two results with the same fingerprint are considered the same.

### fixes property

A result object **MAY** contain a property names fixes whose value is an array of one or more unique (§3.9) fix objects (§3.29).

### properties property

A result object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the result that is not explicitly specified in the SARIF format.

## location object

### General

A location object specifies the location where an analysis tool detected a result. Depending on the circumstances, a location object specifies the physical location (§3.19) of the result, the logical location (§3.18.5) of the result, or both.

A logical location specifies a programmatic construct, for example, a class name or a function name, without specifying the programming artifact within which that construct occurs.

NOTE: There are two reasons to include logical locations in the SARIF format in addition to physical locations:

1. In the absence of symbol information, binary analysis tools might not have source code locations available, so information about line and column numbers might not be present in the log file. In this case, code editors, other programs, or end users can use logical location to navigate from a result to the correct source code location.  
  
2. Logical location information is an important contributor to fingerprinting scenarios, because it is typically more resilient to changes in source code than are line locations. See [Appendix B](#AppendixFingerprints) for more information about fingerprinting. The fullyQualifiedLogicalName property (§3.18.5) is particularly convenient for fingerprinting.

### Constraints

Depending on the information available to the tool that produces the SARIF log file, either or both of the analysisTarget property (§3.18.3) and the resultFile property (§3.18.4) **SHALL** be present.

If the tool that produces the log file knows the analysis target, then the analysisTarget property **SHALL** be present. If the tool knows that the result file is different from the analysis target, then the resultFile property **SHALL** be present; otherwise the resultFile property **SHALL** be absent.

NOTE: Generally, an analysis tool will know both the file it was instructed to scan (the analysis target) and the file in which it detects a problem (the result file).

EXAMPLE 1: Suppose an analysis tool for C++ source code is instructed to scan the source file *a.cpp*, and suppose the tool detects a problem in *a.cpp*. In this case, the tool should set the analysisTarget property to *a.cpp*, and it should not set the resultFile property.

EXAMPLE 2: Suppose an analysis tool for C++ source code is instructed to scan the source file *a.cpp*, which includes the header file *b.h*, and suppose the tool detects a problem in *b.h*. In this case, the tool should set the analysisTarget property to *a.cpp*, and it should set the resultFile property to *b.h*.

EXAMPLE 3: Suppose an analysis tool for object code detects a problem in the binary file *c.dll*, and suppose the tool has available symbol information which maps that location within the binary to a specific line in a source file *d.cpp*. In this case, the tool should set the analysisTarget property to *c.dll*, and it should set the *resultFile* property to *d.cpp*.

If the tool that produces the log file does not know the analysis target, then the resultFile property **SHALL** be present and the analysisTarget property **SHALL** be absent.

NOTE: Some analysis tools produce output in a format that does not include both the analysis target and the result file. In such cases, a conversion tool which translates the output into the SARIF format might only have the result file available.

EXAMPLE 4: Suppose an analysis tool for C++ source code is instructed to scan the source file *a.cpp*, which includes the header file *b.h*, and suppose the tool detects a problem in *b.h*. Suppose further that the tool produces output in a format other than SARIF, for example:  
  
{ "file": "b.h", "line": 6, "col": 1, "msg": "Uninitialized" }  
  
Suppose a conversion tool attempts to translate this output into SARIF format. Suppose that the conversion tool does not know whether the analysis tool was instructed to scan a source file that included *b.h*, or whether it was instructed to scan *b.h* directly. In this case, the conversion tool only knows that the problem occurred in *b.h*. The conversion tool should set the resultFile property to *b.h*, and it should not set the analysisTarget property.

### analysisTarget property

Depending on the information available to the tool that produces the log file (see §3.18.2), a location object either **SHALL** or **SHALL NOT** contain a property named analysisTarget whose value is a physicalLocation object (§3.19) which identifies the file that the analysis tool was instructed to scan. This does not need to be the same as the file where the result actually occurred.

### resultFile property

Depending on the information available to the tool that produces the log file (see §3.18.2), a location object either **SHALL** or **SHALL NOT** contain a property named resultFile whose value is a physicalLocation object (§3.19) that identifies the file where the analysis tool detected the result.

### fullyQualifiedLogicalName property

Depending on the circumstances, a location object either **SHOULD** or **MAY** contain a property named fullyQualifiedLogicalName whose value is a string which specifies the fully qualified name of the logical location where the analysis tool detected the result. If physical location information is not available, fullyQualifiedLogicalName **SHOULD** be present. Otherwise, fullyQualifiedLogicalName **MAY** be present.

The format of the fullyQualifiedLogicalName string **SHALL** be consistent with the programming language in which the programmatic construct specified by that logical location was expressed.

EXAMPLE 1: C: create\_process

EXAMPLE 2: C++: Namespace1::Class::Method(int, double) const &&

EXAMPLE 3: C#: Namespace1.Class.Method(string, int[])

If the run.logicalLocations property (§3.12.10) is present, the value of the fullyQualifiedLogicalName property **SHALL** be equal to the name of one of the properties on the run.logicalLocations object, with one exception, described in §3.18.6.

NOTE: There are a few reasons the fullyQualifiedLogicalName property exists, even though the information it contains is presented in more detail in the run.logicalLocations property:

* It allows a log file viewer to display the logical location in a way that is easily understood by users.
* As mentioned in §3.18.1, fullyQualifiedLogicalName is also particularly convenient for fingerprinting, although the more detailed information in run.logicalLocations could be used instead.
* It relieves viewers from having to format the logical location from the more detailed information in run.logicalLocations.
* It is useful for producing readable in-source suppressions (for example, “suppress all instance of rule CA2101 in the class NamespaceA.NamespaceB.ClassC”).

### logicalLocationKey property

A location object **MAY** contain a property named logicalLocationKey whose value is a string. If present, this string **SHALL** be equal to the name of one of the properties on the run.logicalLocations object (§3.12.10), which provides additional information about the logical location specified by fullyQualifiedLogicalName (§3.18.5).

logicalLocationKey is only necessary if, in the course of a run, the tool produces results in two or more distinct logical locations with the same fullyQualifiedLogicalName. In that case, the tool **SHALL** synthesize a unique name by appending a suffix to fullyQualifiedLogicalName, assign the resulting string to logicalLocationKey, and use that string as the key into the run.logicalLocations dictionary.

EXAMPLE: Suppose a tool analyzes two C++ source files:

// file1.cpp

namespace A {

class B {

}

}

// file2.cpp

namespace A {

namespace B {

class C {

}

}

}

(These could not coexist in the same compilation, but there is no reason two such source files could not exist.)

If the tool detected one result in class B in *file1.cpp*, and another result in namespace B in *file2.cpp*, the fullyQualifiedLogicalName for both would be A::B. In that case, the tool might set the logicalLocationKey property in either one of the results to A::B-1, and it might populate the logicalLocations property as follows:

"logicalLocations": {

"A::B": [

{

"name": "A",

"kind": "namespace"

},

{

"name": "B",

"kind": "namespace"

}

],

"A::B-1": [

{

"name": "A",

"kind": "namespace"

},

{

"name": "B",

"kind": "type"

}

]

}

### decoratedName property

A location object **MAY** contain a property named decoratedName whose value is a string containing the compiler's internal representation of the logical location associated with this location object.

Even though decoratedName describes a logical location, the presence of decoratedName does not require that fullyQualifiedLogicalName (§3.18.5) also be present.

EXAMPLE In this example, the decoratedName property contains a “mangled” name emitted by a C++ compiler:

{ # A "location" object

"fullyQualifiedLogicalName": "b::c(float)",

"decoratedName": "?c@b@@AAGXM@Z"

}

### properties property

A location object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the location that is not explicitly specified in the SARIF format.

## physicalLocation object

### General

A physicalLocation object represents the physical location where a result was detected. A physical location specifies a reference to a programming artifact together with a region within that artifact.

### uri property

With certain exceptions, a physicalLocation object **SHALL** contain a property named uri whose value is a string that represents the location of the file as a valid URI (§3.2).

The exceptions are as follows:

* Under certain circumstances, if the physicalLocation object appears as the value of an annotatedCodeLocation.targetLocation property (§3.25.10), the uri property **MAY** be absent, as described in §3.25.10.
* Under certain circumstances, if the physicalLocation object appears as a member of an annotation.locations array (§3.26.3) which in turn appears as the value of an annotatedCodeLocation.annotations property (§3.25.15), the uri property **MAY** be absent, as described in §3.25.15.

If the run.files property (§3.12.9) is present, the value of the uri property **SHOULD** be equal to the name of one of the properties on the run.files object, which provides additional information about the file specified by uri.

EXAMPLE:

{

"version": "1.0",

"runs": [

{

"files": {

"file:///C:/Code/main.c": [

{

"mimeType": "text/x-c",

}

]

},

"results": [

{

"ruleId": "CA2101",

"level": "error",

"locations": [

{

"analysisTarget": {

"uri": "file:///C:/Code/main.c",

"region: {

"startLine": 24,

"startColumn": 9

}

}

}

]

}

]

}

]

}

### uriBaseId property

If the uri property (§3.19.2) is present and contains a relative URI, then the physicalLocation object **MAY** contain a property named uriBaseId whose value is a string containing a URI base id (§3.3) which indirectly specifies the absolute URI with respect to which uri **SHALL** be interpreted.

If the uri property is absent or contains an absolute URI, then the uriBaseId property **SHALL** be absent.

### region property

A physicalLocation object **MAY** contain a property named region whose value is a region object (§3.20) that represents the region within a file where the result was detected.

If the result occurs in a nested file, then the region property **SHALL** specify the location of the result with respect to the innermost nested file.

EXAMPLE: If a result occurs in a C++ file contained in a compressed archive, then the region would represent the line and column number of the result with the C++ file. It would not represent (for example) the offset of the C++ file from the start of the archive.

## region object

### General

A region object represents a region, that is, a contiguous portion of a file. Every property in a region object shall be represented by a non-negative integer, that is, by a JSON number value with no sign, no fractional part, and no exponent part.

SARIF defines two types of regions: text regions and binary regions.

* A text region represents a contiguous range of of zero or more characters.
* A binary region represents a contiguous range of zero or more bytes.

SARIF defines different properties to represent text regions and binary regions.

In a text region, the startLine property (§3.20.4) **SHALL** be present and **SHALL** have a value greater than 0. In a binary region, the startLine property **SHALL** be absent.

NOTE 1: Consumers of SARIF files can use the presence or absence of the startLine property to determine whether to treat a region as a text region or as a binary region.

NOTE 2: It is up to each analysis tool whether to treat a given file as a text file (in which case it should emit text regions for results detected in the file) or as a binary file (in which case it should emit binary regions).

### Text regions

The line number of the first line in a text file **SHALL** have the value 1. The column number of the first character in each line shall **HAVE** the value 1.

NOTE 1: SARIF defines column number as a count of characters. If a line in a text file contains tab characters, viewers may choose to present column numbers that match the visual offset of each character from the beginning of the line. These “visual” column numbers will not match the column numbers contained in the SARIF file.

Depending on the file's character encoding, each character might be represented by one byte or by multiple bytes. In source files encoded in UTF-16, characters outside the Basic Multilingual Plane (BMP) are represented as a sequence of two 16-bit code points; this sequence is called a “surrogate pair” [[UNICODE10](#UNICODE10)]. Tools that report results in UTF-16-encoded files **SHALL** consider characters outside the BMP as occupying two columns.

NOTE 2: The reason for this requirement is that is common for existing tools to ignore surrogate pairs when calculating column numbers.

Programs such as viewers that process SARIF log files together with the analysis target files to which those log files refer **SHOULD** attempt to determine the character encoding of the target files. In the absence of internal information such as a Byte Order Mark, viewers **MAY** use external information (for example, command line arguments, project settings, or other configuration information) to determine the character encoding. If external information is also lacking, viewers **SHOULD** assume that each character occupies one byte.

The start of a text region **SHALL** be represented by a combination of the startLine (§3.20.4) and startColumn (§3.20.5) properties. startLine **SHALL** be present. If startColumn is absent, the region **SHALL** be considered to start at column 1. For the remainder of this section, whenever startColumn is mentioned, it includes the case where startColumn is absent and so is considered to be 1.

The end of a text region **SHALL** be represented either by a combination of the endLine (§3.20.6) and endColumn (§3.20.7) properties, or by the length property (§3.20.9).

If endLine is absent and endColumn is present, endLine **SHALL** be considered to be the same as startLine.

If endLine is present and endColumn is absent, then:

* If endLine is the same as startLine, then endColumn **SHALL** be considered to be the same as startColumn.
* If endLine is different from startLine, then endColumn **SHALL** be considered to be 1.

For the remainder of this section, whenever endLine is mentioned, it includes the case where endLine is absent and so is considered to be the same as startLine.

For the remainder of this section, whenever endColumn is mentioned, it includes the case where endColumn is absent and so has its default value, which depends on the value of endLine as described above.

If endLine is the same as startLine and endColumn is the same as startColumn, the length of the region **SHALL** be considered to be 0.

If length is present, it **SHALL** be non-negative and **SHALL** represent a count of characters.

If none of endLine, endColumn, or length is present, the length of the region **SHALL** be considered to be 0.

endLine **SHALL** be greater than or equal to startLine.

If endLine is equal to startLine, then endColumn **SHALL** be greater than or equal to startColumn.

To represent a region that includes the last character in a line, excluding any trailing newline sequence, endColumn **SHALL** be set to a value 1 greater than the number of characters in the line, excluding the newline sequence if present. This is the case even for the last line of the file, which might not end with a newline sequence.

EXAMPLE 1: Suppose a text file contains the following line, on line 5:

abcde

Then the region with startLine = 5, startColumn = 3, endLine = 5, and endColumn = 6 represent the three characters cde. This is the case whether or not the line ends with a newline sequence.

To include a newline sequence in a region, endLine **SHALL** be greater than startLine.

EXAMPLE 2: Suppose a text file contains the following lines, starting on line 5:

abcde

fg

Then the region with startLine = 5, startColumn = 3, endLine = 6, and endColumn = 1 represent the three characters cde plus a newline sequence.

### Binary regions

The start of a binary region **SHALL** be represented by the offset property (§3.20.8), which denotes the offset in bytes from the start of the file. The offset of the first byte in a file **SHALL** have the value 0.

The end of a binary region **SHALL** be represented by the length property (§3.20.9), which denotes a count of bytes. If length is absent, the length of the region **SHALL** be considered to be 0.

In a binary region, the startLine (§3.20.4), startColumn (§3.20.5), endLine (§3.20.6), and endColumn (§3.20.7) properties **SHALL** be absent.

### startLine property

When a region object represents a text region, it **SHALL** contain a property named startLine, which **SHALL** have an integer value equal to the line number of the line containing the first character in the region.

The line number of the first line in the file **SHALL** be 1.

### startColumn property

When a region object represents a text region, it **MAY** contain a property named startColumn, which **SHALL** have an integer value equal to the column number of the first character in the region.

The column number of the first column on each line **SHALL** be 1.

If startColumn is absent, it **SHALL** be inferred as specified in §3.20.2.

### endLine property

When a region object represents a text region, it **MAY** contain a property named endLine, which **SHALL** have an integer value equal to the line number of the line containing the last character in the region.

If endLine is absent, it **SHALL** be inferred as specified in §3.20.2.

### endColumn property

When a region object represents a text region, it **MAY** contain a property named endColumn, which **SHALL** have an integer value equal to the column number of the last character in the region.

If endColumn is absent, it **SHALL** be inferred as specified in §3.20.2.

### offset property

When a region object represents a binary region, it **SHALL** contain a property named offset, which **SHALL** have a non-negative integer value equal to the byte offset from the beginning of the file of the first byte in the region.

When a region object represents a text region, the offset property **MAY** be present. In this case, it **SHALL** represent the character offset from the beginning of the file of the first character in the region.

### length property

A region object **MAY** contain a property named length whose value is a non-negative integer.

When the region object represents a text region, the value of length **SHALL** be the number of characters in the region. If the region consists of 0 characters, then either length **SHALL** either be absent or if **SHALL** have the value 0.

When a region object represents a binary region, the value of length **SHALL** be the number of bytes in the region. If the region consists of 0 bytes, then either length **SHALL** either be absent or if **SHALL** have the value 0.

The sum of the offset (§3.20.8) and length properties **SHALL** be greater than or equal to 0, and less than or equal to the length the file, which is measured in characters for a text region and in bytes for a binary region.

A region whose offset is equal to the length of the file and whose length is 0 permitted, and **SHALL** represent an insertion point at the end of the file.

## logicalLocation object

### General

A logicalLocation object describes a logical location.

logicalLocation objects occur as property values within the run.logicalLocations object (§3.12.10).

### name property

A logicalLocation object **SHALL** contain a property named name whose value is a string that identifies the construct in which the result occurred. For example, this property might contain the name of a class or a method.

The name property does need to be suitable for display.

EXAMPLE 1: A C++ analysis tool might emit the name property of a function as the “decorated” function name, which encodes the function signature in a manner that is compiler-dependent and not easily readable.

If the logicalLocation object describes a top-level logical location, and if the name property would be equal to the name of the property for which this object provides the value, then the name property **MAY** be absent.

EXAMPLE 2: In this example, the logical location is a top-level C++ function named functionF, and name is omitted.

"logicalLocations": {

"functionF": {

"kind": "function"

}

}

EXAMPLE 3: In this example, the logical location is a top-level C++ function, and name is equal to the property name.

"logicalLocations": {

"functionF": {

"name": "functionF",

"kind": "function"

}

}

EXAMPLE 4: In this example, the logical location is a top-level C++ function, but name is not equal to the property name, so it cannot be omitted.

"logicalLocations": {

"functionF-0": {

"name": "functionF",

"kind": "function"

}

}

### kind property

A logicalLocation object **SHOULD** contain a property named kind whose value is one of the following strings, if any of those strings accurately describes the construct identified by this object:

* "function"
* "member"
* "module"
* "namespace"
* "package"
* "resource"
* "type"

If none of those strings accurately describes the construct, kind may contain any value specified by the analysis tool.

### parentKey property

If the logical location represented by the logicalLocation object is a nested logical location, then the logicalLocation object **SHALL** contain a property named parentKey whose value is a string that matches the property name of the parent logicalLocation object within run.logicalLocations (§3.12.10).

If the logical location represented by the logicalLocation object is a top-level logical location, then the parentKey property **SHALL** be absent.

## codeFlow object

### General

A code flow is a sequence of locations that specify a possible execution path through the code.

### message property

A codeFlow object **MAY** contain a property named message whose value is a string containing a message relevant to the code flow.

### locations property

A codeFlow object **SHALL** contain a property named locations whose value is an array of one or more annotatedCodeLocation objects (§3.25). Each element of the array **SHALL** represent a single location visited by the tool in the course of producing the result. This array does not need to include every location visited by the tool, but the elements that are present **SHALL** occur in the order that the tool visited them. The elements do need to be unique within the array.

NOTE: The locations array might include multiple identical elements if, for example, the analysis tool simulated the execution of a loop in the course of producing the result.

### properties property

A codeFlow object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the code flow that is not explicitly specified in the SARIF format.

## stack object

### General

A stack object describes a single call stack. A call stack is a sequence of nested function calls, each of which is referred to as a stack frame.

### message property

A stack object **MAY** contain a property named message whose value is a string containing a message relevant to this call stack.

### frames property

A stack object **SHALL** contain a property named frames whose value is an array of one or more stackFrame objects (§3.24). This array **SHALL** include every function call in the stack for which the tool has information, and the entries that are present **SHALL** occur in chronological order with the most recent (innermost) call first and the least recent (outermost) call last. The entries in this array do not need to be unique within the array.

NOTE 1: It is possible for the same frame to occur multiple times if the call stack includes a recursion.

NOTE 2: It is possible that the analysis tool will not have location information for every frame in the call stack. This might happen if, for example, application code for which location information is available calls into operating system code for which location information is not available, which in turn calls back into application code.

### properties property

A stack object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the stack that is not explicitly specified in the SARIF format.

## stackFrame object

### General

A stackFrame object describes a single stack frame within a call stack (§3.23).

### message property

A stackFrame object **MAY** contain a property named message whose value is a string containing a message relevant to this stack frame.

### uri property

A stackFrame object **MAY** contain a property named uri whose value is a string containing a valid URI (§3.2) of the source code file to which this stack frame refers.

### uriBaseId property

If the uri property (§3.24.3) is present and contains a relative URI, then the stackFrame object **MAY** contain a property named uriBaseId whose value is a string containing a URI base id (§5.3) which indirectly specifies the absolute URI with respect to which uri **SHALL** be interpreted.

If the uri property is absent or contains an absolute URI, then the uriBaseId property **SHALL** be absent.

### line property

A stackFrame object **MAY** contain a property named line whose value is an integer containing the 1-based line number within the file specified by uri (§3.24.3) to which this stack frame refers.

If the uri property is absent, the line property **SHALL** be absent.

### column property

A stackFrame object **MAY** contain a property named column whose value is an integer representing the 1-based column number within the line specified by line (§3.24.5) to which this stack frame refers.

If the line property is absent, the column property **SHALL** be absent.

### module property

A stackFrame object **MAY** contain a property named module whose value is a string containing the name of the module that contains the location to which this stack frame refers.

### threadId property

A stackFrame object **MAY** contain a property named threadId whose value is an integer which identifies the thread on which the code at the location specified by this object was executed.

### fullyQualifiedLogicalName property

A stackFrame object **SHALL** contain a property named fullyQualifiedLogicalName whose value is a string containing the fully qualified name of the method to which this stack frame refers. See §3.18.5 for examples.

If the run.logicalLocations property (§3.12.10) is present, the value of the fullyQualifiedLogicalName property **SHOULD** be equal to the name of one of the properties on the run.logicalLocations object, with one exception, described in §3.24.10.

### logicalLocationKey property

A stackFrame object **MAY** contain a property named logicalLocationKey whose value is a string. If present, this string **SHALL** be equal to the name of one of the properties on the run.logicalLocations object (§3.12.10), which provides additional information about the logical location specified by fullyQualifiedLogicalName (§3.24.9).

logicalLocationKey is only necessary if, in the course of a run, the tool produces results in two or more distinct logical locations with the same fullyQualifiedLogicalName. In that case, the tool **SHALL** synthesize a unique name by appending a suffix to fullyQualifiedLogicalName, assign the resulting string to logicalLocationKey, and use that string as the key into the run.logicalLocations dictionary.

### address property

A stackFrame object **MAY** contain a property named address whose value is a non-negative integer containing the address in memory of the location represented by this stack frame.

### offset property

A stackFrame object **MAY** contain a property named offset whose value is a non-negative integer containing the byte offset of the location represented by this stack frame from the start of the method represented by this stack frame.

### parameters property

A stackFrame object **MAY** contain a property named parameters whose value is an array of strings representing the parameters of the function call represented by this stack frame.

### properties property

A stackFrame object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the stack frame that is not explicitly specified in the SARIF format.

## annotatedCodeLocation object

### General

An annotatedCodeLocation object represents a physical location together with additional information relevant to the use of the location in a particular context.

### step property

If an annotatedCodeLocation object occurs within a codeFlow, it **MAY** contain a property named step. If the annotatedCodeLocation does not occur within a codeFlow, the step property **shall** be absent.

The value of the step property **SHALL** be an integer whose value is the 1-based sequence number of the location within the code flow, that is, it shall be 1 for the first location, 2 for the second, and so on.

NOTE This property has two primary purposes:

* A viewer can display the identifier next to each location when it displays a code flow.
* A user reading the log file can easily refer to the location in conversation, for example, “I think the problem occurs at step 6.”

### physicalLocation property

An annotatedCodeLocation object **SHOULD** contain a property named physicalLocation whose value is a physicalLocation object (§3.19) that specifies the file location to which the annotatedCodeLocation object refers.

This property **SHOULD NOT** be absent unless the tool does not have physical location information for this annotatedCodeLocation.

NOTE: This could happen if, for example:

* This annotatedCodeLocation refers to a location within a binary for which the tool does not have associated symbol information.
* This annotatedCodeLocation occurs within a codeFlow (§3.22), the value of the kind property (§3.25.9) is "functionExit", and the tool has chosen not to associate the function exit with a source code location.
* This annotatedCodeLocation occurs within a codeFlow, the value of the kind property is "continuation", and the continuation is used purely to record a change to global state (which might happen asynchronously with respect to the code flow).

### fullyQualifiedLogicalName property

Depending on the circumstance, an annotatedCodeLocation object either **SHOULD** or **MAY** contain a property named fullyQualifiedLogicalName whose value is a string containing the fully qualified name of the method to which this annotatedCodeLocation refers. If the physicalLocation property (§3.25.3) is absent, fullyQualifiedLogicalName **SHOULD** be present. Otherwise, fullyQualifiedLogicalName **MAY** be present. See §3.18.5 for examples.

If the run.logicalLocations property (§3.12.10) is present, the value of the fullyQualifiedLogicalName property **SHOULD** be equal to the name of one of the properties on the run.logicalLocations object, with one exception, described in §3.25.5.

### logicalLocationKey property

An annotatedCodeLocation object **MAY** contain a property named logicalLocationKey whose value is a string. If present, this string **SHALL** be equal to the name of one of the properties on the run.logicalLocations object (§3.12.10), which provides additional information about the logical location specified by fullyQualifiedLogicalName (§3.25.4).

logicalLocationKey is only necessary if, in the course of a run, the tool produces results in two or more distinct logical locations with the same fullyQualifiedLogicalName. In that case, the tool **SHALL** synthesize a unique name by appending a suffix to fullyQualifiedLogicalName, assign the resulting string to logicalLocationKey, and use that string as the key into the run.logicalLocations dictionary.

### module property

An annotatedCodeLocation object **MAY** contain a property named module whose value is a string containing the name of the module that contains the code location specified by this object.

### threadId property

An annotatedCodeLocation object **MAY** contain a property named threadId whose value is an integer which identifies the thread that was executing when the execution of a code flow reached the location specified by this object. If this annotatedCodeLocation does not occur within a codeFlow, the threadId property **SHALL** be absent.

### message property

An annotatedCodeLocation object **MAY** contain a property named message whose value is a string that describes the significance of this location within a particular context.

### kind property

An annotatedCodeLocation object **MAY** contain a property named kind whose value is a string that categorizes the location.

If present, the kind property **SHALL** have one of the following values, with the specified meanings:

* "alias": This location defines an additional name for a variable defined in a declaration.
* "assignment": At this location, an assignment to a variable occurred.
* "branch": At this location, a branch in the execution path occurred.
* "call": This location is the site of a function or method call. Every annotatedCodeLocation whose kind property is "call" **SHALL** be paired with a subsequent annotatedCodeLocation whose kind property is "callReturn" and which refers to the same function, unless the codeFlow in which the call occurs terminates before the function returns.
* "callReturn": This location is the target of a return from a function or method.

NOTE 1: Viewers can use the "call" and "callReturn" values to clarify the presentation of a code flow that crosses function boundaries. For example, when displaying the list of locations in a code flow, a viewer could indent the locations between a "call" and a "callReturn".

* "continuation": Execution continued at this location.

NOTE 2: This can be used, for example, to designate the target of a jump instruction, or the statement after the end of a loop.

* "declaration": The location introduces into the program a name which denotes an entity such as a variable, function, template, *etc.*
* "functionEnter": This location is an entry point to a function or method. Every annotatedCodeLocation whose kind property is "functionEnter" **SHALL** be paired with a subsequent annotatedCodeLocation whose kind property is "functionExit" and which refers to the same function, unless the codeFlow in which the call occurs terminates before a function exit point is reached.
* "functionExit": This location represents the conceptual exit from the function, used by some analysis tools to represent the final node in the directed acyclic graph that represents the control flow through a function. A "functionExit" **MAY** be preceded in the code flow by a "functionReturn".

NOTE 3 A tool might choose (for example) to associate a "functionExit" with the closing brace of a function, or to associate it with the final statement in the function, or not to associate it with a source code location at all.

* "functionReturn": This is the location of a statement that returns control from a function or method (for example, a return statement).
* "usage": At this location, data is used.

NOTE 4: Some analysis tools track the usage of *untrusted* data.

EXAMPLE: Suppose an analysis tool produces a result which states that a piece of data from an insecure source has been used at a particular location. The tool might provide a “related location” (§3.17.12) whose value is an annotatedCodeLocation object with the message “Insecure data entered the system here”.

### kind-dependent properties: target, targetLocation, values and state

Depending on the value of its kind property (§3.25.9), an annotatedCodeLocation object either **MAY**, **SHOULD**, or **SHALL NOT** contain:

* a property named target whose value is a string.
* a property named targetLocation whose value is a physicalLocation object (§3.19).
* a property named values whose value is an array of strings.
* a property named state whose value is an object.

These properties **SHALL** appear only in annotatedCodeLocation objects that are part of a codeFlow (§3.22).

The precise interpretation of these properties, and whether they **MAY**, **SHOULD**, or **SHALL NOT** be present, depends on the value of the kind property.

NOTE 1: In imprecise terms, the meanings of these properties are as follows:

* target represents the thing being operated on at the specified location.
* targetLocation represents the physical location of that thing.
* values represents a set of values that are input to the operation or produced by the operation.
* state is a set of key/value pairs, each of which represents a variable or expression which participates in the operation.

If both the targetLocation property and the physicalLocation property (§3.25.3) of this annotatedCodeLocation object are present, then targetLocation.uri (§3.19.2) **MAY** be absent, in which case it is considered to have the same value as physicalLocation.uri.

The format of the string value of the target property, the elements of the values array, the property names in the state object, and the property values in the state object, **SHALL** be consistent with the syntax of the programming language in which the code being analyzed was written.

In this section, a “variable name” may be any of the following, unless otherwise specified:

* A simple variable name.
* An object property reference.
* An array element reference.
* A reference to the current object.
* Any combination of these.
* Any valid expression that produces a value.

EXAMPLE 1: Examples of valid “variable names” in C++:

* count
* str->length
* values[0]
* this
* this->size
* this->car->wheels[0]
* func() (assuming that func returns a value)

In this section, whenever a “value” is mentioned, it means a string representation of the value.

EXAMPLE 2: Examples of valid “values”:

* A integer value of 2 would be represented as "2".
* A string value of "2" would be represented as "\"2\"".
* A Boolean value of true would be represented as "true".

NOTE 2: In languages where all objects have a built-in string representation (for example, by means of a method such as ToString()), the analysis tool might choose to obtain the string representation by calling that method. For example, in C#, given an object uri of type System.Uri, the tool might choose to obtain the string value by calling uri.ToString(), perhaps resulting in "http://www.example.com".

The requirements and interpretation of the target, targetLocation, values, and state properties are as follows:

* When kind is "alias":
  + target **SHOULD** be present. If present, its value **SHALL** be the name of the alias being created. If multiple aliases are created in the same source language statement, the analysis tool **SHALL** create a separate annotatedCodeLocation object for each alias that the tool wishes to represent in the log.
  + targetLocation **SHALL** be absent.
  + values **SHOULD** be present. If present, its value **SHALL** be an array with one element, whose value is the name of the variable being aliased.
  + state **MAY** be present. If present, it **SHALL** contain a single property whose name is the name of the variable being aliased, and whose value is the value of that variable.
* When kind is "assignment":
  + target **SHOULD** be present. If present, its value **SHALL** be the name of the variable being assigned to. If multiple variables are assigned to in the same source statement, the analysis tool **SHALL** create a separate annotatedCodeLocation object for each assignment that the tool wishes to represent in the log.
  + targetLocation **SHALL** be absent.
  + values **SHOULD** be present. If present, its value shall be an array with one element, whose value is the value assigned to the target variable.
  + state **MAY** be present. If present, it **SHALL** contain properties which specify the names and values of selected variables or subexpressions which participate in the expression on the right-hand side of the assignment.
* When kind is "branch":
  + target **SHOULD** be present if the target of the branch is a named label, in which case its value **SHALL** be the name of the label; otherwise, it **SHALL** be absent.
  + targetLocation **MAY** be present. If present, its value **SHALL** specify the location of the target of the branch.
  + values **MAY** be present if the branch is the result of a test, in which case its value **SHALL** be an array with one element, whose value is the Boolean value of the test condition; otherwise, it **SHALL** be absent.
  + state **MAY** be present if the branch is the result of a test, in which case it shall contain properties which specify the names and values of selected variables or subexpressions which participate in the expression being tested; otherwise, it **SHALL** be absent.
* When kind is "call":
  + target **SHOULD** be present. If present, its value **SHALL** be the fully qualified name of the function being called.
  + targetLocation **MAY** be present. If present, its value **SHALL** specify the physical location of the function being called.
  + values **MAY** be present. If present, its value **SHALL** be an array containing the values of the arguments level
  + object reference (for example, this) passed to object method calls.
  + state **MAY** be present. If present, it **SHALL** contain properties which specify the names and values of selected variables or subexpressions participating in the expressions passed as arguments to the function. For object method calls, this **MAY** include the name and value of the object on which the method was invoked, or any variables or subexpressions which participate in an expression which resolves to that object.
* When kind is "callReturn":
  + target **SHOULD** be present. If present, its value **SHALL** be the fully qualified name of the function being returned from.
  + targetLocation **SHALL** be absent.
  + values **MAY** be present, in which case its value **SHALL** be an array containing the value or values returned from the function; otherwise, it **SHALL** be absent.
  + state **MAY** be present. If present, it **SHALL** contain the names and values of any parameters that were passed by reference to the called function and whose value was reassigned by the called function.
* When kind is "continuation":
  + target **SHALL** be absent.
  + targetLocation **SHALL** be absent.
  + values **SHALL** be absent.
  + state **MAY** be present. If present, it **SHALL** contain the names and values of selected variables or expressions at the specified location. Any variable that is in scope at the specified location **MAY** be mentioned or used in an expression.
* When kind is "declaration":
  + target **SHOULD** be present. If present, its value **SHALL** be the name of the variable being declared. If multiple variables are declared in the same source statement, the analysis tool **SHALL** create a separate annotatedCodeLocation object for each declaration that the tool wishes to represent in the log.
  + targetLocation **SHALL** be absent.
  + values **MAY** be present if the declaration has an initializer, in which case its value **SHALL** be an array containing one element, whose value **SHALL** be the value of the initializer expression, or if the variable is automatically initialized to a default value, in which case its value **SHALL** be an array containing one element, whose value **SHALL** be that default value; otherwise, it **SHALL** be absent.
  + state **MAY** be present if the declaration has an initializer, in which case it **SHALL** contain the names and values of selected variables or subexpressions participating in the initializer expression; otherwise, it **SHALL** be absent.
* When kind is "functionEnter":
  + target **SHOULD** be present. If present, its value **SHALL** be the fully qualified name of the function being entered. If there is a matching "functionExit", then either both of them or neither of them **SHALL** specify target, and if they do, their values **SHALL** be the same.
  + targetLocation **SHALL** be absent.
  + values **MAY** be present. If present, its value **SHALL** be an array containing the values of the arguments passed to the function. This array **SHALL NOT** include the implicit object reference (for example, this) passed to object method calls.
  + state **MAY** be present. If present, it **SHALL** contain the names and values of selected variables or expressions at the specified location. Any variable whose value is available at the specified location **MAY** be mentioned or used in an expression.
* When kind is "functionExit":
  + target **SHOULD** be present. If present, its value **SHALL** be the fully qualified name of the function being returned from. If there is a matching "functionEnter", then either both of them or neither of them **SHALL** specify target, and if they do, their values **SHALL** be the same.
  + targetLocation **SHALL** be absent.
  + values **MAY** be present if the function returns a value or values, in which case its value **SHALL** be an array containing the value or values returned from the function; otherwise, it **SHALL** be absent.
  + state **SHALL** be absent.
* When kind is "functionReturn":
  + target **SHOULD** be present. If present, its value **SHALL** be the fully qualified name of the function being returned from. If there is a matching "functionEnter", then either both of them or neither of them **SHALL** specify target, and if they do, their values **SHALL** be the same.
  + targetLocation **SHALL** be absent.
  + values **MAY** be present if the function returns a value or values, in which case its value **SHALL** be an array containing the value or values returned from the function; otherwise, it **SHALL** be absent.
  + state **MAY** be present if the function returns a value or values, in which case it **SHALL** contain properties which specify the names and values of selected variables or subexpressions which participate in the expressions which produce the returned value or values; otherwise, it **SHALL** be absent.
* When kind is "usage":
  + target **SHOULD** be present. If present, its value **SHALL** be the name of the variable being used. If multiple variables are used in the same source statement, the analysis tool **SHALL** create a separate annotatedCodeLocation object for each usage that the tool wishes to represent in the log.
  + targetLocation **SHALL** be absent.
  + values **MAY** be present. If present, its value **SHALL** be an array with one element, whose value is the value of the used variable at the specified location.
  + state **SHALL** be absent.

EXAMPLE 3: In C++, if the source code contains the declaration

std::string &str = name;

then the value of kind should be "alias", the value of target should be "str", the value of values should be

[ "name" ]

and the value of state might be

{ "name": "\"John\"" }

EXAMPLE 4: In C++, if the source code contains the declaration

std::string &str = name, &str2 = address;

and if the tool creating the log wished to represent both aliases in the log file, then the tool should create two annotatedCodeLocation objects, each with kind set to "alias", and referring to the same source line.

EXAMPLE 5: In C++ or C#, if the source code contains the assignment

m = n + p;

then the value of kind should be "assignment", the value of target should be "m", the value of values might be

[ "5" ]

and the value of state might be

{ "n": "2", "p": "3" }

Or, since state can include expressions, the value of state might be

{ "n + p": "5" }

or even

{ "n": "2", "p": "3", "n + p": "5" }

EXAMPLE 6: In C#, if the source code contains the test

if (s.Length > 0 && y > 2 && valid())

then the value of kind should be "branch", target should be absent, the value of values might be

[ "true" ]

and the value of state might be

{ "s": "\"A string\"", "y": "3" }

or perhaps

{ "s": "\"A string\"", "s.Length": "8", "y": "3", "valid()": "true" }

EXAMPLE 7: In C++ or C#, if the source code contains the function call

func(7, m + n, "s", this, g(2));

then the value of kind should be "call", the value of target might be "func" (or, for example, "N.C.func" if the function func occurred in class C in namespace N), the value of values should be

[ "7", "m + n", "\"s\"", "this", "g(2)" ]

and the value of state might be

{ "m": "2", "n": "3" }

If present, the value of targetLocation would be the physical location where func is defined.

EXAMPLE 8: In C#, if the source code contains the method invocation

example.Func(n);

where example is an object of type SomeClass, then the value of kind should be "call", the value of target should be "SomeClass.Func", the value of values might be

[ "5" ]

and the value of state might be

{ "example": "null", "n": "5" }

(assuming that the method was mistakenly invoked on a null reference).

EXAMPLE 9: In C++ or C#, if the source code contains the function call:

int n = func();

then the value of kind should be "callReturn", the value of target might be "func" (or, for example, "N.C.func" if the function func occurred in class C in namespace N), the value of values might be

[ "5" ]

(assuming that the function returned the value 5), and state should be absent.

EXAMPLE 10: In C++ or C#, if the source code contains the declaration

int m = n + p;

then the value of kind should be "declaration", the value of target should be "m", the value of values might be

[ "5" ]

and the value of state might be

{ "n": "2", "p": "3" }

EXAMPLE 11: In C++ or C#, if the source code contains the declaration

int m = n + p, q = k + r;

and if the tool creating the log wished to represent the declarations of both variables in the log file, then the tool should create two annotatedCodeLocation objects, each with kind set to "declaration", and referring to the same source line.

EXAMPLE 12: In C++ or C#, if the source code contains the return statement

int func()

{

...

return m + n;

}

then the value of kind should be "functionExit", the value of target might be "func" (or, for example, "N.C.func" if the function func occurred in class C in namespace N), the value of values might be

[ "5" ]

and the value of state might be

{ "m": "2", "n": "3" }

If the run.logicalLocations property (§3.12.10) is present, and the value of kind is "call", then the value of the target property **SHOULD** be equal to the name of one of the properties on the run.logicalLocations object, with one exception, described in §3.25.11.

### targetKey property

The annotatedCodeLocation object **MAY** contain a property named targetKey whose value is a string. If present, this string **SHALL** be equal to the name of one of the properties on the run.logicalLocations object (§3.12.10), which provides additional information about the function specified by target (§3.25.10).

targetKey is only necessary if, in the course of a run, the tool encounters two or more distinct functions with the same fully qualified logical name. In that case, the tool **SHALL** synthesize a unique name by appending a suffix to target, assign the resulting string to targetKey, and use that string as the key into the run.logicalLocations dictionary.

### importance property

An annotatedCodeLocation object **MAY** contain a property named importance whose value is a string that specifies the importance of this annotatedCodeLocation in understanding the codeFlow object (§3.22) in which it occurs. If this annotatedCodeLocation does not occur within a codeFlow, the importance property **SHALL** be absent.

If present, the importance property **SHALL** have one of the following values, with the specified meanings:

* "important": this location is important for understanding the code flow.
* "essential": this location is essential for understanding the code flow.
* "unimportant": this location contributes to a more detailed understanding of the code flow, but is not normally needed.

If this property is absent, it **SHALL** be considered to have the value "important".

NOTE: A viewer might use this property to offer the user three options for viewing a lengthy code flow:

* A “normal view,” which omits locations whose importance property is "unimportant".
* An “abbreviated view,” which displays only those locations whose importance property is "essential".
* A “verbose view,” which displays all the locations in the code flow.

### taintKind property

An annotatedCodeLocation object **MAY** contain a property named taintKind whose value is a string which classifies state transitions in code locations relevant to a taint analysis.

If present, the taintKind property **SHALL** have one of the following values, with the specified meanings:

* "source": At this location, untrusted data enters the system (for example, by being provided by a user or read from a file on disk).
* "sanitizer": This is the location of a statement (for example, a function call), after the execution of which data that entered the system from outside (for example, from user input) is presumed to be safe.
* "sink": At this location, untrusted data enters some security-sensitive code (for example, an eval statement that converts untrusted text to executable code).

### snippet property

An annotatedCodeLocation object **MAY** contain a property named snippet whose value is a string containing the text of the source code lines specified by annotatedCodeLocation.physicalLocation.region (§3.19.4).

### annotations property

An annotatedCodeLocation object **MAY** contain a property named annotations whose value is an array containing one or more unique (§3.9) annotation objects (§3.26), each of which describes one or more additional physical locations which are relevant to this annotatedCodeLocation object.

EXAMPLE Consider an annotatedCodeLocation object which describes the declaration statement

int x = (y + z) \* q;

The kind property should be "declaration", the target property should be "x", the values property might be "42", and the state property might be

{ "y": "2", "z": "4", "y + z": "6", "q": "7" }

Now, if the analysis tool wanted to emphasize the value of the expression (y + z), for example, to allow a viewer to highlight the expression, or to display a message when the mouse hovered over the expression, it might set the annotations property to

[ # an array of annotation objects

{ # an annotation object

"message": "(y + z) = 42",

"locations": [ # an array of physicalLocation objects

{ # a physicalLocation object

# The uri property can be omitted if it is

# the same as

# annotatedCodeLocation.physicalLocation.uri

"region": {

"startLine": 12,

"startColumn": 13,

"endColumn": 19

}

}

]

}

]

For any integer array indices i and j, if value the of the property annotatedCodeLocation.annotations[i].locations[j].uri is the same as the value of the property annotatedCodeLocation.physicalLocation.uri, then the uri property **MAY** be omitted from the physicalLocation object annotatedCodeLocation.annotations[i].locations[j], as in the example above. In that case, annotatedCodeLocation.annotations[i].locations[j].uri is considered to have the same value as annotatedCodeLocation.physicalLocation.uri.

### properties property

An annotatedCodeLocation object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include additional information about the use of the location in this context that is not explicitly specified in the SARIF format.

## annotation object

### General

An annotation object associates a message with one or more physical locations.

### message property

An annotation object **SHALL** contain a property named message whose value is a string that describes the physical location or locations specified by the locations property (§3.26.3).

### locations property

An annotation object **SHALL** contain a property named locations whose value is an array containing one or more unique (§3.9) physicalLocation objects (§3.19) to which the message (§3.26.2) is relevant.

## rule object

### General

A rule object contains information that describes a rule.

### Constraints

Either the shortDescription property (§3.27.5) or the fullDescription property (§3.27.6) or both **SHALL** be present.

### id property

A rule object **SHALL** contain a property named id whose value is a string containing a stable, opaque identifier for the rule.

EXAMPLE: "CA2101"

NOTE: Rule identifiers must be stable for two reasons:

* So build automation scripts can refer to specific checks, for example, to disable them, without the risk of a script breaking if a rule id changes.
* So result management systems can compare results from one run to the next, without erroneously designating results as “new” because a rule id has changed.

Rule identifiers should be opaque – that is, they should not convey information to a user – because a rule's implementation might change over time. Suppose a rule id is "DoNotDoXOrY", suppose circumstances change so that “Y” is now acceptable, and suppose the implementation of the rule changes accordingly. Because the rule id must not change, the string "DoNotDoXOrY" will continue to be persisted to logs, where it will convey outdated guidance to users in a way that an opaque identifier such as "CA2101" would not.

### name property

A rule object **MAY** contain a property named name whose value is a string containing a rule identifier that is understandable to an end user. If name contains implementation details that change over time, a tool author might alter a rule's name (while leaving the stable id property unchanged).

NOTE: A rule name is suitable in contexts where a readable identifier is preferable and where the lack of stability is not a concern.

EXAMPLE: "SpecifyMarshalingForPInvokeStringArguments"

### shortDescription property

A rule object **MAY** contain a property named shortDescription whose value is a string containing a concise description of the rule. The shortDescription property **SHOULD** be a single sentence that is understandable when visible space is limited to a single line of text.

EXAMPLE: "Specify marshaling for P/Invoke string arguments"

### fullDescription property

A rule object **SHOULD** contain a property named fullDescription whose value is a string that describes the rule.

The fullDescription property **SHOULD**, as far as possible, provide details sufficient to enable resolution of any problem indicated by the result.

The fullDescription property **SHOULD** conform to the guidelines for message properties (§3.10); in particular, the first sentence of the fullDescription property **SHOULD** provide a concise description of the rule, suitable for display in cases where available space is limited. Tools that construct fullDescription in this way do not need to provide a value for the shortDescription property. Tools that do not construct fullDescription in this way **SHOULD** provide a value for the shortDescription property, because otherwise, the initial portion of fullDescription that a viewer displays where available space is limited might not be understandable.

### defaultLevel property

A rule object **MAY** contain a property named defaultLevel whose value is one of the strings "warning", "error", or "note", with the same meanings as when those strings appear as the value of the result.level property (§3.17.4).

If this property is absent, it **SHALL** be considered to have the value "warning".

The value of this property **SHALL** provide value for the level property for any result object which refers to this rule through its ruleId property (§3.17.2) or its ruleKey property (§3.17.3), and which does not itself specify a level property.

### messageFormats property

A rule object **MAY** contain a property named messageFormats whose value is a JSON object consisting of a set of name/value pairs with arbitrary names.

The value within each name/value pair **SHALL** be a string, which we refer to as a “message format,” that can be used to construct a formatted message in combination with an arbitrary number of additional strings, which we refer to as “arguments” (see §3.28.3).

A message format **SHALL** consist of plain text interspersed with zero or more placeholders. Each placeholder **SHALL** be of the form “{n}”, where n is a non-negative integer which represents a 0-based index into the list of arguments. When a viewer or other program displays a message whose format is specified by a message format, it **SHALL** replace every occurrence of the placeholder {n} with the string value at index n in the list of arguments. Within a message format, the characters “{” and “}” **SHALL** be represented by the character sequences “{{” and “}}” respectively.

Aside from the presence of the placeholders, a message format **SHOULD** conform to the guidelines for message properties (§3.10).

EXAMPLE 1: Given a message format:

"The variable \"{0}\" defined on line {1} is never used. Consider removing \"{0}\"."

together with the arguments "x" and "12", a viewer should display the formatted string

The variable "x" defined on line 12 is never used. Consider removing "x".

The set of names appearing in the messageFormats property **SHALL** contain at least the set of strings which occur as values of the result.formattedMessage.formatId property (§3.28.2) in the log file. The messageFormats property **MAY** contain additional name/value pairs whose names do not appear as the value of the result.formattedMessage.formatId property for any result in the log file.

NOTE: Additional name/value pairs are permitted in the messageFormats property for the convenience of tool vendors, who might find it easier to emit the entire set of messages supported by a rule, rather than restricting it to those messages that happen to appear in the log file.

EXAMPLE 2:

{

"objectCreation": "{0} creates a new instance of {1} which is never used.

Pass the instance as an argument to another method,

assign the instance to a variable,

or remove the object creation if it is unnecessary.",

"stringReturnValue": "{0} calls {1} but does not use the new string

instance that the method returns.

Pass the instance as an argument to another method,

assign the instance to a variable,

or remove the call if it is unnecessary."

}

### helpUri property

A rule object **MAY** contain a property named helpUri whose value is a string containing the URI where the primary documentation for the rule can be found.

NOTE: The documentation might include examples, contact information for the rule authors, and links to additional information about the rule.

### help property

A rule object **MAY** contain a property named help whose value is a string containing a message which provides the documentation for the rule.

This property is useful when help information is not available at a URI, for example, when the rule is a custom rule written by a developer, as opposed to one supplied by the tool vendor.

### properties property

A rule object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the rule that is not explicitly specified in the SARIF format.

## formattedMessage object

### General

A formattedMessage object contains information that can be used to construct a formatted message that describes a result.

### formatId property

A formattedMessage object **SHALL** contain a property named formatId whose value is a string that identifies the message format used to format the message that describes this result. The value of formatId **SHALL** correspond to one of the names in the set of name/value pairs contained in the messageFormats property (§3.27.8) of the rule object (§3.27) whose id property (§3.27.3) matches the ruleId property (§3.17.2) of this result object.

### arguments property

If the message format string specified by formatId contains any placeholders, the formattedMessage object **SHALL** contain a property named arguments, whose value is an array of string values that **SHALL** be used, in combination with a message format, to construct a result message. The array **SHALL** have exactly as many elements are there are distinct placeholders in the message format. The array element at index n **SHALL** correspond to the placeholder {n} in the message format.

If the message format string specified by formatId does not contain any placeholders, the arguments property **SHALL** be absent.

EXAMPLE: Suppose formatId refers to the following message format:

"The variable "{0}" defined on line {1} is never used. Consider removing \"{0}\"."

There are two distinct placeholders, {0} and {1} (although {0} occurs twice). Therefore, the arguments array will have two elements, the first corresponding to {0} and the second corresponding to {1}.

## fix object

### General

A fix object represents a proposed fix for the problem indicated by the result object (§3.17) in which it occurs. It specifies a set of files to modify. For each file, it specifies which bytes to remove, and provides new bytes to be inserted.

EXAMPLE

{ # a result object (see §3.17)

"fix": {

"description": # see §3.29.2

"Private member names begin with '\_'",

"fileChanges": [ # see §3.29.3

{ # a fileChange object (see §3.30)

...

}

]

}

}

### description property

A fix object **SHOULD** contain a property named description whose value is a string describing the proposed fix.

NOTE: The purpose of the description property is to enable a result log file viewer to present the proposed fix to the end user.

EXAMPLE:

"fix": {

"description": "Combine declaration and initialization of variable x"

...

}

### fileChanges property

A fix object **SHALL** contain a property named fileChanges whose value is an array of one or more fileChange objects (§3.30).

## fileChange object

### General

A fileChange object represents a change to a single file.

EXAMPLE:

{ # a fix object (see §3.29)

"fileChanges": # see §3.29.3

[

{ # a fileChange object

"uri": "a.h", # see §3.30.2

"replacements": # see §3.30.4

[

{ # a replacement object (see §3.31)

...

},

{ # another replacement object.

...

}

]

}

]

}

### uri property

A fileChange object **SHALL** contain a property named uri whose value is a string value that represents the location of the file as a valid URI (§3.2).

### uriBaseId property

If the uri property (§3.30.2) contains a relative URI, then the fileChange object **MAY** contain a property named uriBaseId whose value is a string containing a URI base id (§3.3) which indirectly specifies the absolute URI with respect to which uri **SHALL** be interpreted.

If the uri property is absent or contains an absolute URI, then the uriBaseId property **SHALL** be absent.

### replacements property

A fileChange object **SHALL** contain a property named replacements whose value is an array of one or more replacement objects (§3.31), each of which represents the replacement of a single range of bytes in the file specified by the uri property (§3.30.2).

## replacement object

### General

A replacement object represents the replacement of a single range of bytes in a file. It specifies the location within the file where the replacement is to be made, the number of bytes to remove at that location, and a sequence of bytes to insert at that location.

If a replacement object specifies both the removal of a byte range by means of the deletedLength property (§3.31.4) and the insertion of a sequence of bytes by means of the insertedBytes property (§3.31.5), then the effect of the replacement **SHALL** be as if the removal were performed before the insertion.

If a single fileChange object (§3.30) specifies more than one replacement, then the effect of the replacements **SHALL** be as if they were performed in the order they appear in the replacements array (§3.30.4). The offset property (§3.31.3) of each replacement **SHALL** specify an offset in the unmodified file.

EXAMPLE: Suppose a fileChange object contains a fileChanges property whose value is the following array of two replacement objects:

"fileChanges": [

{

"offset": 12,

"deletedLength": 5,

"insertedBytes": "ZXhhbXBsZQ==" # The string "example"

},

{

"offset": 20,

"deletedLength": 3

}

]

The first replacement object removes 5 bytes starting at offset 12; that is, it removes bytes 12–16. Then it inserts 7 bytes (the UTF-8-encoded string example, itself encoded in MIME Base64) at the same offset.

The second replacement object removes 3 bytes starting at offset 20 *with respect to the unmodified file*. Since 5 bytes were removed and 7 bytes inserted *before* byte 20, the 3 bytes removed actually start at byte 22.

### Constraints

In any replacement object, either the deletedLength property (§3.31.4) **SHALL** be present and have a value greater than 0, or the insertedBytes property (§3.31.5) **SHALL** be present and have a string value whose length is greater than zero, or both.

NOTE: A replacement object in which the deletedLength property was absent or had a value of 0, and in which the insertedBytes property was absent or had a value equal to the empty string, would neither insert nor remove any bytes, and so would not be meaningful.

### offset property

A replacement object **SHALL** contain a property named offset whose value is a non-negative integer specifying the offset in bytes from the beginning of the file at which bytes are to be removed, inserted, or both. An offset of 0 shall denote the first byte in the file.

### deletedLength property

A replacement object **MAY** contain a property named deletedLength whose value is a non-negative integer specifying the number of bytes to delete, starting at the byte offset specified by the offset property (§3.31.3), measured from the beginning of the file.

If deletedLength is absent, or if its value is 0, no bytes **SHALL** be deleted.

### insertedBytes property

A replacement object **MAY** contain a property named insertedBytes whose value is a string that specifies the byte sequence to be inserted at the byte offset specified by the offset property (§5.31.3), measured from the beginning of the file.

If insertedBytes is absent, or if its value is the empty string, no bytes **SHALL** be inserted.

If the file into which the bytes are to be inserted is a binary file, the value of the insertedBytes string **SHALL** be the MIME Base64 encoding of the byte sequence to be inserted.

If the file into which the bytes are to be inserted is a text file, the characters to be inserted **SHALL** first be encoded in UTF-8. The value of the insertedBytes string **SHALL** be the MIME Base64 encoding of the resulting UTF-8 byte sequence.

TODO: Relationship between original file encoding, offset, and UTF-8 byte sequence.

TODO: Explain responsibility of viewer/editor to match encoding.

## notification object

### General

A notification object describes a condition encountered in the course of running an analysis tool which is relevant to the operation of the tool itself, as opposed to being relevant to a file being analyzed by the tool. Conditions relevant to files being analyzed by a tool are represented by result objects (§3.17).

### id property

A notification object **MAY** contain a property named id whose value is a string containing an identifier for the condition that was encountered.

NOTE: In contrast to rule identifiers (see rule.id, §3.27.3), which must be stable and opaque, notification identifiers do need to be either stable or opaque, because the reasoning that leads to those requirements for rule ids does not apply to tool notifications. A tool notification with level "error" should always be treated as a failure, and tools should not allow them to be disabled. And tool authors are free to change the notification ids at any time, so there is no reason for them to be opaque; to the contrary, they are more useful if they convey information to the user.

### ruleId property

If the condition described by the notification object is relevant to a particular analysis rule, the notification object **SHOULD** contain a property named ruleId whose value is a string containing the stable, unique identifier of the rule (§3.27.3).

### ruleKey property

If there is more than one rule with the id specified by the ruleId property (§3.32.3), and if the run object in which this notification object occurs contains a rules property (§3.12.14), then the notification object **SHALL** contain a property named ruleKey whose value is a string that matches one of the property names in the run.rules object.

The value of the ruleId property on this notification object must match the id property (§3.27.3) of the rule object (§3.27) identified by ruleKey.

EXAMPLE: In this example, there is more than one rule with id CA1711. When the log file includes a notification with that rule id, it provides a value for ruleKey to specify which of the rules with that id is meant.

"runs": [

{

"configurationNotifications": [

{

"id": "CFG0001",

"message": "Rule configuration is missing.",

"ruleKey": "CA1711-1", # Specifies a property name within "rules".

"ruleId": "CA1711" # Matches the "id" value of the specified

# property value within "rules"

}

],

"rules": {

"CA1711-1": {

"id": "CA1711"

},

"CA1711-2": {

"id": "CA1711"

}

}

}

]

### physicalLocation property

If the condition described by the notification object is relevant to a particular file location, the notification object **SHOULD** contain a property named physicalLocation whose value is a physicalLocation object (§3.19) that identifies the relevant location.

### message property

A notification object **SHALL** contain a property named message whose value is a string that describes the condition that was encountered.

### level property

A notification object **MAY** contain a property named level whose value is one of a fixed set of strings that specify the severity level of the notification.

If present, the level property **SHALL** have one of the following values, with the specified meanings:

* "error": A serious problem was found. The condition encountered by the tool resulted in the analysis being halted, or caused the results to be incorrect or incomplete.
* "warning": A problem that is not considered to be serious was found. The condition encountered by the tool is such that it is uncertain whether a problem occurred, or is such that the analysis might be incomplete but the results that were generated are probably valid.
* "note": The notification is purely informational. There is no required action.

passlevel property is absent, it **SHALL** be considered equivalent to the value "warning".

### threadId property

A notification object **MAY** contain a property named threadId whose value is an integer which identifies the thread associated with this notification.

### time property

A notification object **MAY** contain a property named time whose value is a string specifying the date and time at which the analysis tool generated the notification. The string **SHALL** be in the format specified by (§3.8).

### exception property

If the notification is a result of a runtime exception, the notification object **MAY** contain a property named exception whose value is an exception object (§3.33).

If the notification is not the result of a runtime exception, the exception property **SHALL** be absent.

### properties property

A notification object **MAY** contain a property named properties whose value is a property bag (§3.7). This allows tools to include information about the encountered condition that is not explicitly specified in the SARIF format.

## exception object

### General

An exception object describes a runtime exception encountered in the course of executing an analysis tool. This includes signals in POSIX-conforming operating systems

### kind property

An exception object **SHOULD** contain a property named kind whose value is a string describing the exception.

If the exception represents a thrown object, kind **SHALL** be the fully qualified type name of the object that was thrown, if that information is available.

EXAMPLE 1: C#: "System.ArgumentNullException"

If the exception represents a POSIX signal, kind **SHALL** be the symbolic name of the signal as specified in <signal.h>.

EXAMPLE 2: POSIX: "SIGFPE"

If the tool does not have access to information about the object that was thrown, the kind property **SHALL** be absent.

### message property

An exception object **SHOULD** contain a property named message whose value is a string that describes the exception.

If the tool does not have access to an appropriate property of the thrown object, the message property **SHALL** be absent.

EXAMPLE 1: C++: The tool should populate message from the string returned from the what() method of any object derived from std::exception.

EXAMPLE 2: C#: The tool should populate message from the value of the Message property of any object derived from System.Exception.

### stack property

An exception object **MAY** contain a property named stack whose value is a stack object (§3.23) that describes the sequence of function calls leading to the exception.

### innerExceptions property

An exception object **MAY** contain a property named innerExceptions whose value is an array of one or more exception objects, each of which is considered to be a cause of the containing exception.

NOTE: There is commonly no more than one inner exception. This property is an array to accommodate platforms that provide a mechanism for aggregating exceptions, such as the System.AggregateException class from the .NET Framework.

# Conformance

(**Note**: The [OASIS TC Process](https://www.oasis-open.org/policies-guidelines/tc-process#wpComponentsConfClause) requires that a specification approved by the TC at the Committee Specification Public Review Draft, Committee Specification or OASIS Standard level must include a separate section, listing a set of numbered conformance clauses, to which any implementation of the specification must adhere in order to claim conformance to the specification (or any optional portion thereof). This is done by listing the conformance clauses here.

For the definition of ‘conformance clause,’ see [OASIS Defined Terms](https://www.oasis-open.org/policies-guidelines/oasis-defined-terms-2017-05-26#dConformanceClause).

See “Guidelines to Writing Conformance Clauses”:   
<http://docs.oasis-open.org/templates/TCHandbook/ConformanceGuidelines.html>.

Remove this note before submitting for publication.)

1. Acknowledgments

(**Note:** A Work Product approved by the TC must include a list of people who participated in the development of the Work Product. This is generally done by collecting the list of names in this appendix. This list shall be initially compiled by the Chair, and any Member of the TC may add or remove their names from the list by request.

Remove this note before submitting for publication.)

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

[Participant Name, Affiliation | Individual Member]

[Participant Name, Affiliation | Individual Member]

1. Use of fingerprints by result management systems

On large software projects, a single run of a set of analysis tools can produce hundreds of thousands of results or more. To deal with such a large number of results, some software development teams adopt a strategy whereby they first prevent the introduction of new problems into their code, and then work to address the existing problems.

To prevent the introduction of new problems, it is necessary first to record the results from a designated run. We refer to this as a baseline. It is then necessary to compare the results from a subsequent run with the baseline.

To determine whether a result from a subsequent run is the same as a result from the baseline, there must be a way to use information contained in the result to construct a stable identifier for the result. We refer to this identifier as a fingerprint.

A result management system can construct a fingerprint by using information contained in the SARIF file such as

* the name of the tool that produced the result.
* the rule id.
* the file system path to the analysis target.

There are situations where information that would be helpful in uniquely identifying a result is not easily detectable by the result management system. For example, consider a tool which checks documentation for words that are culturally or politically sensitive. The word would most likely occur only in the fullMessage property, for example: "The word xxx should not be used in documentation."

The SARIF format provides the toolFingerprintContribution property to allow analysis tools to provide additional information which a result management system can incorporate into the fingerprint that it constructs for each result. In this example, the tool might set the value of toolFingerprintContribution to the prohibited word.

Some information contained in the result is not useful in constructing a fingerprint. For example, suppose the fingerprint were to include the line number where the result was located, and suppose that after the baseline was constructed, a developer inserted additional lines of code above that location. Then in the next run, the result would occur on a different line, the computed fingerprint would change, and the result management system would erroneously report it as a new result.

It is difficult to devise an algorithm that constructs a truly stable fingerprint for a result. Fortunately, for practical purposes, the fingerprint does need to be absolutely stable; it only needs to be stable enough to reduce the number of results that are erroneously reported as “new” to a low enough level that the development team can manage the erroneously reported results without too much effort.

1. Use of SARIF by log file viewers

It is frequently useful for an end user to view the results produced by an analysis tool in the context of the programming artifacts in which they occur. A log file viewer is a program that allows an end user to do this.

Typically, the user opens a log file in the viewer, which presents a list of the results in the log file. When the user selects a result from the list, the viewer displays the source code from the file specified in the result, and displays information about the result in the vicinity of the region where the result occurred. For example, the viewer might interleave result information between lines of source code.

There are various reasons why a viewer might need to know the type of information contained in a source file that it displays:

* If the viewer knows the programming language, it can provide services such as syntax highlighting.
* If the result occurs in a source file that is nested within (for example) a compressed container file, then the viewer needs to know the file type of the container so that it can extract the source file.

There are various ways that a viewer might obtain file type information. In the SARIF format, the mimeType property of the file object provides this information. In the absence of the mimeType property, a viewer can fall back to examining the filename extension, for example “.zip”. It is recommended that the analysis tool provide the mimeType property (which it must know, because it was able to interpret the file in which it detected the result), rather than forcing the viewer to rely on a file name extension.

1. Production of SARIF by converters

There are two broad categories of tools that can produce output in the SARIF format. Analysis tools produce SARIF as a result of performing a scan on a set of analysis targets. Converters translate existing data from a non-SARIF format into the SARIF format. That data might come from an analysis tool that produces output in a non-SARIF format, from a bug database, or from any other source.

Converters should populate those elements of the SARIF format for which a direct equivalent exists in the input data.

If the input data includes information for which there is no SARIF equivalent, converters may use it to populate the various property bags and tag lists defined by the SARIF format, or they may simply omit it from the output. When populating a property bag with such information, converters should use a property name that matches the name of that piece of information in the native tool format, even if that name does not conform to the camelCase convention used in the rest of this specification. This makes it easier to match these properties with the source data in the native tool format.

NOTE: The converter must replace any characters that cannot occur in a JSON string with the appropriate escape sequence.

If the input data does not include an equivalent for any SARIF element, the converter should not attempt to synthesize that element. For example, a converter should not attempt to heuristically extract a rule id from the text of an unstructured error message.

If a converter were to synthesize values, it would potentially introduce additional complexity in the implementation of SARIF viewers. The reason is that the viewer itself might examine the analysis tool and its version in the tool object, and attempt to synthesize missing elements.

Now suppose a converter made a bad choice in synthesizing a missing element, and then fixed the problem in an update. As a result, two log files claiming to have been produced by the same version of the same analysis tools might have different elements filled in, or the same elements filled in differently. For that matter, two different converters might make different choices in how to synthesize missing elements. As a result, the viewer would have to take into account both the analysis tool (and its version) and the converter (and its version) in deciding how to synthesize any remaining elements.

By design, to avoid this added complexity, the SARIF standard does not define an element to hold the converter version. This, together with the guidance that converter implementers should not attempt to synthesize missing elements, allows viewer implementers to assume that all files from the same version of the same tool are identical in structure.

This general guidance is embodied in various sections of the specification. For example:

* A converter should not attempt to synthesize a ruleId for a result if the tool does not provide one.
* A converter that knows which file a result was detected in, but not which file the analysis tool was originally instructed to scan, should populate the location.resultFile property, but should not attempt to populate location.analysisTarget (see §3.18.2).
* A converter should not attempt to guess whether the analysis tool's version string is intended to be interpreted as a Semantic Version 2.0.0 version string (see §3.13.4).

1. Locating rule metadata

The SARIF format allows rule metadata to be included in a SARIF log file (see §3.12.14 and §3.27). A SARIF log file does need to include any rule metadata. This raises the questions of when rule metadata should be included in a log file, and how to locate the rule metadata if it is not included in the log file.

Rule metadata should be included in a log file in the following circumstances:

* The log file is intended to be viewed in a tool such as a log file viewer that needs to display rule metadata related to each result even when the tool is not connected to a network.
* The log file is intended to be uploaded to a result management system which requires information about every rule specified by every result, and which might not have prior knowledge of the rules specified by the results in this log file.
* Neither of the above applies, but the increased log file size due to the rule metadata is not considered significant.

If rule metadata is not included in the log file, this specification does not specify a mechanism for locating the metadata. If the SARIF log file is produced in the context of an engineering system that provides a service from which rule metadata can be obtained (for example, a result management system, or a web service dedicated to rule metadata), then tooling can be created to merge a log file with the relevant metadata when required (for example, when presenting the results in a log file viewer).

1. Producing deterministic SARIF log files
   1. General

In certain circumstances, it is desirable for an analysis tool to produce deterministic output; that is, for it to produce identical output when run repeatedly over identical inputs.

Certain build systems provide an example of when this is desirable. Consider a build system that caches the results of each build step. If the build is rerun, and the inputs to the step are identical (which the build system might determine, for example, by comparing timestamps, or by computing a hash of the inputs to the step and storing it along with the output from the step), then the build system can save time by not re-running the step, and simply using the existing outputs.

In the case of SARIF, one could imagine a sequence of build steps where Steps A B, and C each run an analysis tool on a different set of targets, producing log files A.sarif, B.sarif, and C.sarif, and then build Step D performs an analysis on the aggregate of those log files. If the targets analyzed in Step B change but the targets analyzed in steps A and C do not, and if the contents of the SARIF log file are deterministic, then when the build is re-run, only Steps B and D need to be performed.

Authors of analysis tools are encouraged to provide a mechanism (for example, a command line option such as --deterministic) which instructs the tool to produce deterministic output.

There are several issues to consider when producing deterministic output:

* Avoiding elements of the SARIF file format whose values are non-deterministic.
* Emitting array and dictionary elements in a deterministic order.
* Avoiding absolute paths.
* Handling baseline information
  1. Non-deterministic file format elements

For a tool to produce deterministic output, it should not emit the following elements of the SARIF format. All of these elements are **OPTIONAL**.

Not all of these elements are non-deterministic in all cases. For example, some build systems might run all builds on the same machine or under the same account. However, avoiding these elements, in conjunction with the techniques described in subsequent sections of this Appendix, guarantees deterministic output.

* invocation.startTime
* invocation.endTime
* invocation.processId
* invocation.machine
* invocation.account
* invocation.fileName (because fileName is specified as being an absolute path, and tools might be stored in different directories on different machines)
* invocation.workingDirectory
* invocation.environmentVariables
* The use of absolute file paths in invocation.commandLine (because builds performed on different machines might use a different root directory)
* annotatedCodeLocation.threadId
* notification.threadId
* notification.time
* run.id
* run.automationId
* run.baselineId
* stackFrame.threadId
* stackFrame.address (because security measures such as address space layout randomization (ASLR) might place identical code at different addresses from run to run)
* The presence of any non-deterministic elements in a property bag property
  1. Array and dictionary element ordering

For a tool to produce deterministic output, it must emit array and dictionary elements in a deterministic order.

For some arrays, the SARIF format requires a specific ordering. For example, within the stack.Frames property, SARIF requires the annotatedCodeLocation object representing the most deeply nested function call to appear first.

For other arrays, the SARIF format does not require a specific ordering. For example, within the file.hashes property, SARIF does not require the hash objects to appear in any particular order. For such arrays, a tool can ensure the order by sorting the array elements before writing them to the log file. For example, it might sort the hash objects alphabetically by the string value of the hash.algorithm property.

A tool might similarly choose to emit the string elements of a properties.tags array in locale-insensitive alphabetical order.

The array of result objects in the run.results array presents more of a problem. A multi-threaded analysis tool analyzing multiple files in parallel might produce results in any order, and there is no natural order for the results. A tool might choose to order them, for example, first alphabetically by analysis target URI, then numerically by line number, then by column number, then alphabetically by rule id.

For dictionaries such as the run.rules object or the run.files object, a tool might order the property names alphabetically, using a locale-insensitive ordering.

* 1. Absolute paths

The use of absolute file paths in URI-valued properties such as physicalLocation.uri makes it difficult to produce deterministic output. For example:

* Different build machines might be configured to use different source directories.
* A single build machine might use a different directory for each build.

For a tool to produce deterministic output, it must avoid the use of absolute file paths. Tools can achieve this by emitting URIs that are relative to one or more root directories (for example, a source root directory and an output root directory), and accompanying each URI-valued property with a URI base id property (§3.3).

* 1. Compensating for non-deterministic output

If an analysis tool does not produce deterministic output, a build system can add additional processing steps to compensate.

There are two scenarios to consider:

* Log equality is determined by a simple comparison of file contents, or by comparing file hashes.
* Log equality is determined by an “intelligent” comparison.

In the first scenario, a post-processing step could produce deterministic output by creating a new file that omits non-deterministic elements, reorders array elements and object properties, removes file path prefixes, and introduces uriBaseId properties.

In the second scenario, a post-processing step could intelligently compare the newly produced log to the log from a previous build by ignoring non-deterministic elements, ensuring that arrays have the same elements regardless of order, and ignoring file path prefixes.

* 1. Interaction between determinism and baselining

SARIF's baselining feature poses a particular challenge for determinism. We illustrate the problem with the following scenario:

On a particular date, a project's nightly build runs an analysis tool ToolX, which produces a log file, say, log\_20170914.sarif. The next day, a developer modifies one of the files scanned by the tool in a way that introduces a new problem. That night, the nightly build tool runs again, this time producing a log file which compares the current set of results to those that appeared in the previous run:

ToolX --input a.c b.c --baseline log\_20170914.sarif --output log\_20170915.sarif

Because a new problem has been introduced, log\_20170614.sarif will contain a result object whose baselineState is "new". The next night, without any further changes to the source files, the tool is run yet again:

ToolX --input a.c b.c --baseline log\_20170915.sarif --output log\_20170916.sarif

The result object that first appeared in log\_20160615.sarif still appears in log\_20160616.sarif, but since it existed in the baseline, its baselineState will now be "existing".

The result is that even though none of the analysis target files have changed, the log file has changed, or at least, a simple file comparison (such as comparing the hash of the new log with the hash of the baseline) will report that is has changed.

Strictly speaking, this does not violate determinism. After all, the baseline file has changed, and the baseline file is one of the inputs to the analysis. But from a practical standpoint, this is still a problem, albeit a small one.

If the build uses a simple mechanism such as hash value comparison to determine if a file has changed, then on those occasions when the only difference between the newest log and the baseline is that some results that were previously "new" are now "existing", subsequent build steps which consume the SARIF log file will run, even if they might not actually be necessary. For example, a build step which automatically files bugs for new results will run, even though the log contains no new results. Or a build step which tracks the number of open issues will run, even though the number of open issues has not actually changed.

If the build engineers for a project wish to absolutely minimize the execution of unnecessary build steps, they have various options. They might perform an “intelligent” comparison between the baseline and the new log, treating "new" results in the baseline as equivalent to "existing" results. Or they might rewrite the baseline (marking all "new" results as "existing") before performing the comparison. Of course, there is no guarantee that such an “intelligent” comparison or baseline rewriting process will actually take less time than the unnecessary build steps it is intended to avoid.

1. Guidance on fixes

Tools that produce SARIF files which include fix objects should take care to structure those fixes in such a way as to affect a minimal range of bytes. This maximizes the likelihood that an automated tool can safely apply multiple fixes to the same file.

The following example will clarify what this means and why it is important. Consider an XML file containing the following element:

<lineItem partNumber=A3101 />

Suppose that a (domain-specific) XML scanning tool reported two results:

* The value of the partNumber attribute is not enclosed in quotes.
* The part numbering scheme has changed, and part numbers beginning with “A” now begin with “AA”.

Fixing only result #1 would produce the element

<lineItem partNumber="A3101" />

Fixing only result #2 would produce the element

<lineItem partNumber=AA3101 />

Fixing both results should produce the element

<lineItem partNumber="AA3101" />

The fix for result #1 might be specified in various ways, for example:

1. As a single replacement:
   * Replace the characters A3101 with the characters "A3101".
2. As a sequence of two replacements:
   1. Insert a quotation mark before A3101.
   2. Insert a quotation mark after A3101.

The fix for result #2 is most simply specified as a single replacement:

* Replace the characters A3101 with the characters AA3101.

Suppose there exists an automated tool which reads a SARIF file containing fix objects and applies as many of the specified fixes as possible to the source files.

If the fix for result #1 were structured as a single replacement, then after applying the fix, the tool would not be able to fix result #2, because the range of characters specified by the fix for result #2 would have been replaced. On the other hand, if the fix for result #1 were structured as two replacements (with a separate insertion for each quotation mark), the tool would still be able to apply the fix for result #2, because the targeted range of characters would still exist.

Therefore, structuring fixes as sequences of minimal, disjoint byte range replacements maximizes the amount of work that can be done by automated fixup tools.

1. Examples

This Appendix contains examples of complete, valid SARIF files, to complement the fragments shown in examples throughout this document.

* 1. Minimal valid SARIF file resulting from a scan

This is a minimal valid SARIF file for the case where the analysis tool was run with the intent of scanning files and producing results (see §3.12.11). The file contains only those elements required by the specification (that is, those elements which the specification states **SHALL** be present).

The file contains a single run object (§3.12) with an empty results array (§3.12.11), as would happen if the tool detected no issues in any of the files it scanned.

{

"version": "1.0.0",

"runs": [

{

"tool": {

"name": "CodeScanner",

"semanticVersion": "2.1.0"

},

"results": [

]

}

]

}

* 1. Minimal recommended SARIF file with source information

This is a minimal recommended SARIF file for the case where

1. The analysis tool was run with the intent of scanning files and producing results (see §3.12.11), and
2. The analysis tool has source location information available.

The file contains those elements recommended by the specification (that is, those elements which the specification states “**SHOULD**” be present), in addition to the required elements.

The file contains a single run object (§3.12) with a results array (§3.12.11). The results array contains a single result object (§3.17) so the recommended elements of the result object can be shown.

It contains a run.files property (§3.12.9) specifying only those files in which the tool detected a result.

It does not contain a run.logicalLocations property (§3.12.10), because when physical location information is available, that property is optional (it “**MAY**” be present).

This example also includes a run.rules property (§3.12.14) containing rule metadata, even though rule metadata is optional, to show how a SARIF log file can be self-contained, in the sense of containing all the information necessary to interpret the results.

{

"version": "1.0.0",

"runs": [

{

"tool": {

"name": "CodeScanner",

"semanticVersion": "2.1.0"

},

"files": {

"file:///user/builder/work/src/collections/list.cpp": {

"mimeType": "text/x-c"

}

},

"results": [

{

"ruleId": "C2001",

"message": "Variable \"count\" was used without being initialized.",

"locations": [

{

"analysisTarget": {

"uri": "file:///user/builder/work/src/collections/list.cpp",

"region": {

"startLine": 15

}

},

"fullyQualifiedLogicalName": "collections::list:add"

}

]

}

],

"rules": {

"C2001": {

"id": "C2001",

"fullDescription": "A variable was used without being initialized.

This can result in runtime errors such as

null reference exceptions."

}

}

}

]

}

* 1. Minimal recommended SARIF file without source information

This is a minimal recommended SARIF file for the case where

1. The analysis tool was run with the intent of scanning files and producing results (see §3.12.11), but
2. The analysis tool does not have source location information available.

The file contains those elements recommended by the specification (that is, those elements which the specification states “**SHOULD**” be present), in addition to the required elements.

The file contains a single run object (§3.12) with a results array (§3.12.11). The results array contains a single result object (§3.17) so the recommended elements of the result object can be shown.

It contains a run.files property (§3.12.9) specifying only those files in which the tool detected a result.

It contains a run.logicalLocations property (§3.12.10), because when physical location information is not available, that property is recommended (it “**SHOULD**” be present).

{

"version": "1.0.0",

"runs": [

{

"tool": {

"name": "BinaryScanner",

"semanticVersion": "1.0.1"

},

"files": {

"file:///user/builder/work/bin/example": {

"mimeType": "application/vnd.microsoft.portable-executable"

}

},

"logicalLocations": {

"Example": {

"name": "Example",

"kind": "namespace"

},

"Example.Worker": {

"name": "Worker",

"kind": "type",

"parentKey": "Example"

},

"Example.Worker.DoWork": {

"name": "DoWork",

"kind": "function",

"parentKey": "Example.Worker"

}

},

"results": [

{

"ruleId": "B6412",

"message": "The insecure method \"Crypto.Sha1.Encrypt\"

should not be used.",

"level": "warning",

"locations": [

{

"fullyQualifiedLogicalName": "Example.Worker.DoWork"

}

]

}

]

}

]

}

* 1. SARIF file for exporting rule metadata

This sample demonstrates the use of SARIF for exporting a tool's rule metadata. The file contains a single run object (§3.12) with no results array, but with a rules object (§3.12.14) containing rule metadata.

{

"version": "1.0.0",

"runs": [

{

"tool": {

"name": "BinaryAnalyzer",

"semanticVersion": "2.1.0"

},

"rules": {

"BA2006": {

"id": "BA2006",

"name": "BuildWithSecureTools",

"shortDescription": "Application code should be compiled with

the most up-to-date tool sets.",

"fullDescription": "Application code should be compiled with

the most up-to-date tool sets. The latest

version is 2.2.",

"messageFormats": {

"Error\_BadModule": "built with {0} compiler version {1}

(Front end version {2})",

"Pass": "{0} was built with tools that satisfy

configured policy.",

"Error": "{0} was compiled with one or tools that do not

satisfy configured policy.",

"NotApplicable\_InvalidMetadata": "{0} was not evaluated for

check '{1}'."

},

"defaultLevel": "warning",

"helpUri":

"http://www.example.com/tools/BinaryAnalyzer/rules/BA2006"

}

}

}

]

}

* 1. Comprehensive SARIF file

The purpose of this example is to demonstrate the usage of as many SARIF elements as possible. Not all elements are shown, because some are mutually exclusive.

Because the purpose is to present as many elements as possibly, the file as a whole does not represent best practices for SARIF usage, nor does it represent the output of a single, coherent analysis. For example, the result presented in the file involves a runtime exception, but at the same time it is marked as suppressedExternally (to demonstrate the result.suppressionStates property), which is unrealistic.

{

"version": "1.0.0",

"$schema": "http://json.schemastore.org/sarif-1.0.0",

"runs": [

{

"id": "BC650830-A9FE-44CB-8818-AD6C387279A0",

"stableId": "Nightly code scan",

"baselineId": "0A106451-C9B1-4309-A7EE-06988B95F723",

"automationId": "Build-14.0.1.2-Release-20160716-13:22:18",

"architecture": "x86",

"tool": {

"name": "CodeScanner",

"fullName": "CodeScanner 1.1 for Unix (en-US)",

"version": "2.1",

"semanticVersion": "2.1.0",

"fileVersion": "2.1.0.0",

"language": "en-US",

"sarifLoggerVersion": "1.25.0",

"properties": {

"copyright": "Copyright (c) 2017 by Example Corporation.

All rights reserved."

}

},

"invocation": {

"commandLine": "CodeScanner @collections.rsp",

"responseFiles": {

"collections.rsp": "-input src/collections/\*.cpp -log out/collections.sarif -rules all -disable C9999"

},

"startTime": "2016-07-16T14:18:25Z",

"endTime": "2016-07-16T14:19:01Z",

"machine": "BLD01",

"account": "buildAgent",

"processId": 1218,

"fileName": "/bin/tools/CodeScanner",

"workingDirectory": "/home/buildAgent/src",

"environmentVariables": {

"PATH": "/usr/local/bin:/bin:/bin/tools:/home/buildAgent/bin",

"HOME": "/home/buildAgent",

"TZ": "EST"

}

},

"files": {

"file:///home/buildAgent/src/collections/list.cpp": {

"mimeType": "text/x-c",

"length": 980,

"hashes": [

{

"algorithm": "sha256",

"value": "b13ce2678a8807ba0765ab94a0ecd394f869bc81"

}

]

},

"file:///home/buildAgent/bin/app.zip": {

"mimeType": "application/zip"

},

"file:///home/buildAgent/bin/app.zip#/docs/intro.docx": {

"uri": "/docs/intro.docx",

"mimeType": "application/vnd.openxmlformats-officedocument.wordprocessingml.document",

"parentKey": "file:///home/buildAgent/bin/app.zip",

"offset": 17522,

"length": 4050

}

},

"logicalLocations": {

"collections::list::add": {

"name": "add",

"kind": "function",

"parentKey": "collections::list"

},

"collections::list": {

"name": "list",

"kind": "type",

"parentKey": "collections"

},

"collections": {

"name": "collections",

"kind": "namespace"

}

},

"results": [

{

"ruleId": "C2001",

"formattedRuleMessage": {

"formatId": "default",

"arguments": [

"ptr"

]

},

"suppressionStates": [ "suppressedExternally" ],

"baselineState": "existing",

"level": "error",

"snippet": "add\_core(ptr, offset, val);\n return;",

"locations": [

{

"analysisTarget": {

"uri": "file:///home/buildAgent/src/collections/list.cpp"

},

"resultFile": {

"uri": "file:///home/buildAgent/src/collections/list.h",

"region": {

"startLine": 15,

"startColumn": 9,

"endLine": 15,

"endColumn": 10,

"length": 1,

"offset": 254

}

},

"fullyQualifiedLogicalName": "collections::list:add",

"decoratedName": "?add@list@collections@@QAEXH@Z"

}

],

"relatedLocations": [

{

"message": "\"count\" was declared here.",

"physicalLocation": {

"uri": "file:///home/buildAgent/src/collections/list.h",

"region": {

"startLine": 8,

"startColumn": 5

}

},

"fullyQualifiedLogicalName": "collections::list:add"

}

],

"codeFlows": [

{

"message": "Path from declaration to usage",

"locations": [

{

"step": 1,

"kind": "declaration",

"target": "ptr",

"importance": "essential",

"message": "Variable \"ptr\" declared.",

"snippet": "int \*ptr;",

"physicalLocation": {

"uri": "file:///home/buildAgent/src/collections/list.h",

"region": {

"startLine": 15

}

},

"fullyQualifiedLogicalName": "collections::list:add",

"module": "platform",

"threadId": 52

},

{

"step": 2,

"kind": "assignment",

"target": "offset",

"values": [

"43"

],

"state": {

"y": "2",

"z": "4",

"y + z": "6",

"q": "7"

},

"importance": "unimportant",

"snippet": "offset = (y + z) \* q + 1;",

"physicalLocation": {

"uri": "file:///home/buildAgent/src/collections/list.h",

"region": {

"startLine": 15

}

},

"annotations": [

{

"message": "(y + z) = 42",

"locations": [

{

"region": {

"startLine": 15,

"startColumn": 13,

"endColumn": 19

}

}

]

}

],

"fullyQualifiedLogicalName": "collections::list:add",

"module": "platform",

"threadId": 52

},

{

"step": 3,

"kind": "call",

"importance": "essential",

"message": "Uninitialized variable \"ptr\" passed to

method \"add\_core\".",

"snippet": "add\_core(ptr, offset, val)",

"target": "collections::list:add\_core",

"physicalLocation": {

"uri": "file:///home/buildAgent/src/collections/list.h",

"region": {

"startLine": 25

}

},

"fullyQualifiedLogicalName": "collections::list:add",

"module": "platform",

"threadId": 52

}

]

}

],

"stacks": [

{

"message": "Call stack resulting from usage of

uninitialized variable.",

"frames": [

{

"message": "Exception thrown.",

"uri": "file:///home/buildAgent/src/collections/list.h",

"line": 110,

"column": 15,

"module": "platform",

"threadId": 52,

"fullyQualifiedLogicalName": "collections::list:add\_core",

"address": 10092852,

"offset": 16,

"parameters": [ "null", "0", "14" ]

},

{

"uri": "file:///home/buildAgent/src/collections/list.h",

"line": 43,

"column": 15,

"module": "platform",

"threadId": 52,

"fullyQualifiedLogicalName": "collections::list:add",

"address": 10092176,

"offset": 84,

"parameters": [ "14" ]

},

{

"uri": "file:///home/buildAgent/src/application/main.cpp",

"line": 28,

"column": 9,

"module": "application",

"threadId": 52,

"fullyQualifiedLogicalName": "main",

"address": 10091200,

"offset": 156

}

]

}

],

"fixes": [

{

"description": "Initialize the variable to null",

"fileChanges": [

{

"uri": "file:///home/buildAgent/src/collections/list.h",

"replacements": [

{

"offset": 109,

"insertedBytes": "PSBudWxs"

}

]

}

]

}

]

}

],

"configurationNotifications": [

{

"id": "UnknownRule",

"ruleId": "ABC0001",

"level": "warning",

"message": "Could not disable rule \"ABC0001\" because

there is no rule with that id."

}

],

"toolNotifications": [

{

"id": "CTN0001",

"level": "note",

"message": "Run started."

},

{

"id": "CTN9999",

"ruleId": "C2152",

"level": "error",

"message": "Exception evaluating rule \"C2152\". Rule disabled;

run continues.",

"physicalLocation": {

"uri": "file:///home/buildAgent/src/crypto/hash.cpp"

},

"threadId": 52,

"time": "2016-07-16T14:18:43.119Z",

"exception": {

"kind": "ExecutionEngine.RuleFailureException",

"message": "Unhandled exception during rule evaluation.",

"stack": {

"frames": [

{

"message": "Exception thrown",

"module": "RuleLibrary",

"threadId": 52,

"fullyQualifiedLogicalName":

"Rules.SecureHashAlgorithmRule.Evaluate",

"address": 10092852

},

{

"module": "ExecutionEngine",

"threadId": 52,

"fullyQualifiedLogicalName":

"ExecutionEngine.Engine.EvaluateRule",

"address": 10073356

}

]

},

"innerExceptions": [

{

"kind": "System.ArgumentException",

"message": "length is < 0"

}

]

}

},

{

"id": "CTN0002",

"level": "note",

"message": "Run ended."

}

],

"rules": {

"C2001": {

"id": "C2001",

"shortDescription": "A variable was used without being

initialized.",

"fullDescription": "A variable was used without being initialized.

This can result in runtime errors such as

null reference exceptions.",

"messageFormats": {

"default": "Variable \"{0}\" was used without being initialized."

}

}

}

}

]

}

1. Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Editor** | **Changes Made** |
| 01 | 2017/09/22 | Laurence J. Golding | Initial version, transcribed from contribution with minor corrections. |