# Static Analysis Results Interchange Format (SARIF) Version 2.0

## Committee Specification Draft 0102 /
## Public Review Draft 0102

## 15 June 2018

### Specification URIs

## 27 May 2019

**This version:**
https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/sarif-v2.0-csprd02.docx (Authoritative)
https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/sarif-v2.0-csprd02.html
https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/sarif-v2.0-csprd02.pdf

**Previous version:**
http://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/sarif-v2.0-csprd01.docx (Authoritative)
http://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/sarif-v2.0-csprd01.html
http://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/sarif-v2.0-csprd01.pdf

**Previous version:**
N/A

**Latest version:**
https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.docx (Authoritative)
https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html
https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.pdf

**Technical Committee:**
OASIS Static Analysis Results Interchange Format (SARIF) TC

**Chairs:**
David Keaton (dmk@dmk.com), Individual Member
Luke Cartey (luke@semmle.com), Semmle

**Editors:**
Michael C. Fanning (mikefan@microsoft.com), Microsoft
Laurence J. Golding (v-lgold@microsoft.com), Individual Member), Microsoft

**Additional artifacts:**
This prose specification is one component of a Work Product that also includes:
- JSON schemas: https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/schemas/

**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 document was last revised or approved by the OASIS Static Analysis Results Interchange Format (SARIF) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif#technical.

TC members should send comments on this specification to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at https://www.oasis-open.org/committees/sarif/.

This specification is provided under the RF on RAND Terms Mode of the OASIS IPR Policy, 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).

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any 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.

**Citation format:**

When referencing this specification the following citation format should be used:

**[SARIF-v2.0]**

*Static Analysis Results Interchange Format (SARIF) Version 2.0*. Edited by Michael Fanning and Laurence J. Golding. ~~15 June 2018~~27 May 2019. OASIS Committee Specification Draft ~~01/~~02 / Public Review Draft ~~01~~02. https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/sarif-v2.0-csprd02.html. Latest version: ~~http~~https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html.

  

# Notices

# Table of Contents

# 1 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, compliance with contractual or legal requirements, compliance with stylistic standards, understandability, and maintainability. To form an overall picture of program quality, developers often need to 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~~.~~, called the Static Analysis Results Interchange Format, or "SARIF"[1]. 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 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).~~

Although most static analysis tools analyze files on disk, SARIF can represent results detected in any URI-addressable artifact (for example, the text returned by an HTTP query). This specification uses the term "artifact" to refer to any item that a tool might analyze. It uses the more restrictive term "file" when referring specifically to a file on disk.

## 1.1 IPR Policy

This specification is ~~being developed~~provided under the RF on RAND Terms Mode of the OASIS IPR Policy, 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).

## 1.2 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in "Key words for use in RFCs to Indicate Requirement Levels" [BCP14] [RFC2119] and "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words" [RFC8174] when, and only when, they appear in all capitals, as shown here.

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

[1] Pronounced 'sæ-rɪf ("a" as in "cat", "i" as in "if", emphasis on the first syllable).

  

**analysis target**

artifact which aan analysis tool is instructed to analyze

**artifact**

see sequence of bytes addressable *via* a URI

Examples: A physical file in a file system such as a source file, an object file, a configuration file or a data file; a specific version of a file in a version control system; a database table accessed *via* an HTTP request; an arbitrary stream of bytes returned from an HTTP request.

**baseline**

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

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

**baseline run**

run that produces a baseline to which subsequent runs can be compared

**binary fileartifact**

artifact considered as a sequence of bytes

**binary region**

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

**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`.

**code flow**

set of one or more thread flows which together specify a pattern of code execution relevant to detecting a result

**column (number)**

1-based index of a character within a line

**configuration file**

file, typically textual, that configures the execution of an analysis tool or tool component

**converter**

SARIF producer that transforms the output of an analysis tool from its native output format into the SARIF format

**custom taxonomy**

**taxonomydeterministic producer**

which, given identical inputs, repeatedly produces an identical defined by and intended for use with a particular analysis tool

**direct producer**

analysis tool which acts as a SARIF producer

**driver**

  

tool component containing an analysis tool's or converter's primary executable, which controls the tool's or converter's execution, and which in the case of an analysis tool typically defines a set of analysis rules

**embedded link**

syntactic construct which enables a message string to refer to a location within aan artifact mentioned in a result

**~~embedded resource~~**

**~~that is contained within a~~ engineering system**

software development environment within which analysis tools execute

NOTE: An engineering system might include a build system, a source control system, a result management system, a bug tracking system, a test execution system, and so on.

**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

**(end) user**

person who uses the information in a log file to investigate, triage, or resolve results

**extension**

tool component other than the driver (for example, a plugin, a configuration file, or a taxonomy)

**external ~~resource~~property file**

file containing the values of one or more externalized properties

**externalizable property**

property that ~~is~~can be contained ~~within a~~in an ~~SARIF resource~~external property file

**externalized property**

property stored outside of the SARIF log file to which it logically belongs

**false positive**

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

**~~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.~~

**fingerprint**

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

**formatted message**

message string which contains formatting information such as Markdown formatting characters

**fully qualified logical name**

**~~fully qualified logical name~~**

string ~~that~~ that fully identifies the programmatic construct specified by a logical location, typically by means of a hierarchical identifier.

  

Example: The fully qualified logical name of the C# method `f(void)` in class `C` in namespace `N` is `"N.C.f(void)"`. Its logical name is `"f(void)"`.

**hierarchical string**

string in the format `<component>{/<component`~~`>}*`, for example, `"CWE/22"`~~`>}*`

**line**

contiguous sequence of characters, starting either at the beginning of ~~a~~an artifact 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~~artifact

**~~localization~~**

**~~process~~line (number)**

1-based index of ~~adapting~~ a ~~collection~~line within a file

NOTE: Abbreviated to "line" when there is no danger of ambiguity with "line~~ to a~~" in the sense of a sequence of characters.

**localizable**

subject to being translated from one natural language~~, region, or culture~~ to another

**log file**

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

**(log file) viewer**

SARIF consumer 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 artifact in which it occurs

**logical location**

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

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

**logical name**

string that partially identifies the programmatic construct specified by a logical location~~, typically~~ by specifying the most specific (often the rightmost) component of its fully qualified logical name.

Example: The logical name of the C# method `f(void)` in class `C` in namespace `N` is `"f(void)"`. Its fully qualified logical name is `"N.C.f(void)"`.

**message string**

human-readable string that conveys information relevant to an element in a SARIF file

**nested ~~file~~artifact**

artifact ~~which~~ that is contained within another ~~file~~artifact

**nested logical location**

logical location that is ~~nested~~contained within another logical location

Example: A method within a class in C++

**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.

**notification**

reporting item that describes a condition encountered by a tool during its execution

  

**~~opaque~~**

neither human-reable nor machine-parseable into constituent parts

**parent (~~file~~artifact)**

artifact which contains one or more nested artifacts~~nested files~~

**physical location**

location specified by reference to ~~a~~an artifact, possibly together with a region within that artifact

**plain text message**

message string which does not contain any formatting information

**plugin**

tool component that defines additional rules

**~~policy~~**

**set of rule configurations that specify how results that violate the rules defined by a particular tool component ~~(programming) artifact~~**

~~, 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.~~

are to be treated

**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 contractual or legal requirements, a deviation from stylistic standards.

**property ~~bag~~**

~~JSON~~attribute of an object consisting of a name and a value associated with the name

**property bag**

object consisting of an unordered set of ~~properties~~non-standardized properties with arbitrary camelCase names

**~~redaction-aware~~redactable property**

property~~property~~ that potentially contains sensitive information that a SARIF direct producer or a SARIF post-processor might wish to redact

**region**

contiguous portion of ~~a~~an artifact

**reporting item**

unit of output produced by a tool, either a result or a notification

**reporting configuration**

the subset of reporting metadata that a tool can configure at runtime, before performing its scan

Examples: severity level, rank

**reporting descriptor**

container for reporting metadata

**reporting metadata**

  

information that describes a class of related reporting items

Examples: id, description

**repository**

container for a related set of files in a version control system

**response file**

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

**result**

**reporting itemresource**

item that requires , such asdescribes a -or

**result**

condition present in a and reported by a an artifact

**result file**

artifact in which aan analysis tool detects a result

**result management system**

software system that consumes the log files produced by analysis tools, produces reports that enable engineering 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.

**rich text message**

message string which contains formatting information such as Markdown formatting characters

**result matching**

process of determining whether two results are reporting the same condition in the code

**root file**

SARIF log file to which one or more external property files logically belong

**rule**

specific criterion for correctness verified by aan 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.

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

**rule configuration information**

reporting configuration that a can modify at runtime, before executing its scan

that applies to a rule

**rule id**

stable value which aan 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**

reporting metadatainformation that describes a rule

~~Example: id, description, category, author~~

**run**

1. invocation of a specified 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

**SARIF consumer**

program that reads and interprets a SARIF log file

**SARIF log file**

log file in the format defined by the SARIF specification

**SARIF post-processor**

SARIF producer that transforms an existing SARIF log file into a new SARIF log file, for example, by removing or redacting security-sensitive elements.

**SARIF producer**

program that emits output in the SARIF format

~~**SARIF resource file**~~

~~file containing  for a single language, in the format defined by the SARIF specification~~

**stable value**

value which, once established, never changes over time

**standard taxonomy**

taxonomy defined without reference to a particular analysis tool

**(static analysis) tool**

program that examines artifacts to detect problems, without executing the program

Example: Lint

**taxon (pl. taxa)**

one of a set of categories which together comprise a taxonomy

**taxonomy**

classification of analysis results into a set of categories

**tag**

string that conveys additional information about the SARIF log file element to which it applies

~~**taint analysis**~~

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

~~**tainted data**~~

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

**text ~~file~~artifact**

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

**text region**

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

  

**thread flow**

temporally ordered set of code locations specifying a possible execution path through the code, which occur within a single thread of execution, such as an operating system thread or a fiber

**tool component**

component of an analysis tool or converter, either its driver or an extension, consisting of one or more files

**top-level ~~file~~artifact**

artifact which is not contained within any other ~~file~~artifact

~~Example: Category (for example, "Style" or "Security"), documentation URI.~~

**top-level logical location**

logical location that is not nested within another logical location

Example: A global function in C++

**translation**

rendering of a tool component's localizable strings into another language

**triage**

decide whether a result indicates a problem that needs to be corrected

**user**

see end user.

**VCS**

version control system

**viewer**

see log file viewer.

**web analysis tool**

analysis tool that models and analyzes the interaction between a web client and a server.

## 1.3 Normative References

**[BCP14]** Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997, https://tools.ietf.org/html/bcp14.

**[GFM]** "GitHub-Flavored Markdown spec", Version 0.28-gfm (2017-08-01), https://github.github.com/gfm/.

**[IANA-ENC]** Freed, Ned and Dürst, Martin, "Character Sets", 2017-12-20, https://www.iana.org/assignments/character-sets/character-sets.xhtml.

**[IANA-HASH]** "Hash Function Textual Names", https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xhtml, July 4, 2017.

**[ISO3166-1:2013]** "Codes for the representation of names of countries and their subdivisions – Part 1: Country codes", ISO 3166-1:2013, November, 2013, https://www.iso.org/standard/63545.html.

**[ISO639-1:2002]** "Codes for the representation of names of languages ~~--~~– Part 1: Alpha-2 code", ISO 639-1:2002, July 2002, https://www.iso.org/standard/22109.html.

~~**[ISO639-2]** "Codes for the representation of names of languages -- Part 2: Alpha-3 code", ISO 639-2:1998, October 1998, .~~

~~**[ISO639-3]** "Codes for the representation of names of languages -- Part 3: Alpha-3 code for comprehensive coverage of languages", ISO 639-3:2007, February 2007, .~~

**[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.

**[ISO14977:1996]** “Information technology – Syntactic metalanguage – Extended BNF”, ISO/IEC 14977:1996(E), December 1996, https://www.iso.org/standard/26153.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.

**[RFC3987]** Duerst, M. and Suignard, M., "Internationalized Resource Identifiers (IRIs)", RFC 3987, DOI 10.17487/RFC3987, January 2005, https://www.rfc-editor.org/info/rfc3987.

**[RFC4122]** Leach, P., Mealling, M., and Salz, R., "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, http://www.rfc-editor.org/info/rfc4122.

**[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.

**[~~RFC7763]~~** ~~Leonard, S., "The text/markdown Media Type~~**RFC6901]** Bryan, P., Ed., Zyp, K., and Nottingham, M., Ed., "JavaScript Object Notation (JSON) Pointer", RFC ~~7763~~6901, DOI 10.17487/~~RFC7763, March 2016~~RFC6901, April 2013, http://www.rfc-editor.org/info/~~rfc7763~~rfc6901.

~~**[RFC7764]**~~ ~~Leonard, S., "Guidance on Markdown: Design Philosophies, Stability Strategies, and Select Registrations~~**[RFC7230]** Fielding, R., Ed., and Reschke, J., Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC ~~7764~~7230, DOI 10.17487/~~RFC7764, March 2016~~RFC7230, June 2014, http://www.rfc-editor.org/info/rfc7230.

**[RFC8174]** Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, http://www.rfc-editor.org/info/rfc8174.

**[RFC8089]** Kerwin, M., "The "file" URI Scheme", RFC 8089, DOI 10.17487/RFC8089, February 2017, http://www.rfc-editor.org/info/rfc8089.

**[RFC8259]** Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 8259, DOI 10.17487/RFC8259, December 2017, http://www.rfc-editor.org/info/rfc8259.

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

**[~~UNICODE10~~UNICODE12]** Unicode 10.0, June 2017, http://www.unicode.org/versions/Unicode12.0.0.

## 1.4 Non-Normative References

**[CMARK]** “CommonMark Spec”, Version 0.28, (2017-08-01), http://spec.commonmark.org/0.28/.

**[CWE~~]~~™]** “Common Weakness Enumeration”, https://cwe.mitre.org.

  

**[GFMCMARK]** “GitHub's fork of cmark, a CommonMark parsing and rendering library and program in C”, https://github.com/github/cmark.

**[GFMENG]** “GitHub Engineering: A formal spec for GitHub Flavored Markdown”, https://githubengineering.com/a-formal-spec-for-github-markdown/.

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

**[ISO14882:~~2014~~2017]** “Information technology – Programming languages – C++”, ISO/IEC 14882, December ~~2014~~2017, https://www.iso.org/standard/68564.html.

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

**[PE]** “PE Format”, March 17, 2019, https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format.

**[TAR]** “GNU tar 1.32: Basic Tar Format”, http://www.gnu.org/software/tar/manual/html_node/Standard.html.

**[ZIP]** “.ZIP File Format Specification, Version 6.3.6, Revised April 26, 2019”, https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT.

## 1.5 Trademarks

CWE™ is the trademark of a product supplied by The MITRE Corporation.

JavaScript™ is the trademark of Oracle America, Inc.

Linux® is the registered trademark of a product supplied by The Linux Foundation.

Visual Basic™ is the trademark of a product supplied by Microsoft Corporation.

UNIX® is the registered trademark of a product supplied by The Open Group.

Windows® is the registered trademark of a product supplied by Microsoft Corporation.

This information is given for the convenience of users of this document and does not constitute an endorsement by OASIS of any of the products named. Equivalent products may be used if they can be shown to lead to the same results.

  

# 2 Conventions

## 2.1 General

The following conventions are used within this document.

## 2.2 Format examples

This document contains several partial examples of the JSON serialization of the SARIF format. The examples are formatted for clarity, as permitted by JSON [RFC8259], 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~~The examples also employ typographical conventions that are not part of the JSON or SARIF formats:

- 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.~~
- A '#' character introduces a comment that extends to the end of the line.
- When a JSON string is too long to fit on a line, it is broken into multiple lines.
- Some examples have italicized line numbers in the left margin.

## 2.3 Property notation

A ~~JSON~~SARIF object consists of a set of properties. The value of a property can 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 `charLength` property. For clarity, we can refer to the `charLength` property as `physicalLocation.region.charLength`.

## 2.4 Syntax notation

Where this specification describes a syntactic construct, it uses the extended Backus-Naur form (EBNF) ~~defined in~~ [ISO14977:1996].

In all EBNF definitions in this spec:

- The following syntax rules are assumed:

```
decimal digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

non negative integer =
    "0"
  | decimal digit - '0', { decimal digit };
```

- The following "special sequence" (see EBNF [ISO14977:1996], §4.19 and §5.11) refers to any character that can appear in a JSON string according to JSON [ECMA404]:

```
? JSON string character ?
```

## 2.5 Commonly used objects

This specification uses the following notation for certain commonly used objects:

| | |
|---|---|
| `theSarifLog` | The root object of the SARIF log file. |
| `theRun` | The `run` object (§3.14) containing the object under discussion. |

  

| | |
|---|---|
| `theTool` | The value of `theRun.tool` (§3.14.6) |
| `theDescriptor` | The `reportingDescriptor` object (§3.49) identified by the `reportingDescriptorReference` object (§3.52) under discussion. |
| `theComponent` | The `toolComponent` object (§3.19) identified by the `toolComponentReference` object (§3.54) under discussion. |
| `theResult` | The `result` object (§3.27) containing the object under discussion. |
| `thisObject` | The object containing the property under discussion. |

NOTE: Usually when the description of a property refers to another property of the same object, the other property is referred to by its unqualified name. When necessary to avoid confusion, the name of the other property is qualified with "`thisObject.`" to emphasize that it is a property of the object under discussion. For an example, see §3.27.7.

  

# 3 File format

## 3.1 General

A SARIF defines an object model, the top level of which is the `sarifLog` object (§3.13log file **SHALL** contain), which contains 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 requirementscontain a serialization of the SARIF object model into the JSON format.

NOTE 1: In the future, other serializations might be defined.

The top-level value in the log file, representing the `sarifLog` object, **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 inspecified by JSON [RFC8259]. We refer to the object represented by this top-level value as the `sarifLog` object (§).

A SARIF log file **SHALL** be encoded in UTF-8 [RFC3629].

NOTE: 2: JSON [RFC8259] requires this encoding for any JSON text "exchanged between systems that are not part of a closed ecosystem."

## ~~3.2 fileContent objects~~

## 3.2 SARIF file naming convention

The file name of a SARIF log file **SHOULD** end with the extension `".sarif"`.

EXAMPLE 1: `output.sarif`

The file name **MAY** end with the additional extension `".json"`.

EXAMPLE 2: `output.sarif.json`

## 3.3 artifactContent object

### 3.2.13.3.1 General

Certain properties in this specification represent the contents of portions of artifacts external filesto the log file, for example, filesartifacts that were scanned by an analysis tool. SARIF represents such file content with a `fileContent`an `artifactContent` object. Depending on the circumstances, the SARIF log file might need to represent this content as readable text, raw bytes, or both.

### 3.2.23.3.2 text property

If the external fileartifact is a text file, a `fileContent`artifact, an `artifactContent` object **SHOULD** contain a property named `text` whose value is a string containing the relevant text. Since SARIF log files are encoded in UTF-8 ([RFC3629]; see §3.1), this means that if the external fileartifact is a text fileartifact in any encoding other than UTF-8, the SARIF producer **SHALL** transcode the text to UTF-8 before assigning it to the `text` property. The SARIF producer **SHALL** escape any characters that JSON [RFC8259] requires to be escaped.

Notwithstanding any necessary transcoding and escaping, the SARIF producer **SHALL** preserve the text file'sartifact's line breaking convention (for example, `"\n"` or `"\r\n"`).

If the external fileartifact is a binary fileartifact, the `text` property **SHALL** be absent.

  

### ~~3.2.3~~3.3.3 binary property

If the external ~~file~~artifact is a binary ~~file~~artifact, or if the SARIF producer cannot determine whether the external ~~file~~artifact is a text ~~file~~artifact or a binary ~~file, a `fileContent`~~artifact, an `artifactContent` object **SHALL** contain a property named `binary` whose value is a string containing the MIME Base64 encoding [RFC2045] of the bytes in the relevant portion of the ~~file~~artifact.

If the external ~~file~~artifact is a text ~~file~~artifact in an encoding other than UTF-8, the `binary` property **MAY** be present, in which case it **SHALL** contain the MIME Base64 encoding of the bytes representing the relevant text in its original encoding.

If the external ~~file~~artifact is a UTF-8 text ~~file~~artifact, the `binary` property **SHOULD** be absent. If it is present, it **SHALL** contain the MIME Base64 encoding of the UTF-8 bytes representing the relevant text.

### 3.3.4 rendered property

An `artifactContent` object **MAY** contain a property named `rendered` whose value is a `multiformatMessageString` object (§3.12) that provides a rendered view of the contents.

> EXAMPLE: In this example, a `physicalLocation` object (§3.29) denotes a memory address. Its `region.snippet.rendered` property (§3.29.4, §3.30.13) offers a hex view of the relevant address range. The `markdown` property (§3.12.4) emphasizes a byte of particular interest.

```
{                                       # A physicalLocation object (§3.29).
  "address": {                          # See §3.29.6.
    "baseAddress": 4202880,             # See §3.32.6.
    "offset": 64                        # See §3.32.8.
  },

  "region": {                           # See §3.29.4.
    "snippet": {                        # An artifactContent object. See §3.30.13.
```

## ~~3.3~~ "rendered": { # A multiformatMessageString object (§3.12~~fileLocation objects~~

```
).
        "text": "00 00 01 00 00 00 00 00",
        "markdown": "00 00 **01** 00 00 00 00 00"
      }
    }
  }
}
```

## 3.4 artifactLocation object

### ~~3.3.1~~3.4.1 General

Certain properties in this specification specify the location of ~~a file~~an artifact. SARIF represents ~~a file~~an artifact's location with ~~a `fileLocation`~~an `artifactLocation` object. The most important member of ~~a `fileLocation`~~an `artifactLocation` object is its `uri` property (§3.4.3). If the `uri` property contains a relative reference (the term used in the URI standard [RFC 3986] for what is commonly called a "relative URI"), the `uriBaseId` property (§3.4.4) can sometimes be used to resolve the relative reference to an absolute URI.

  

## 3.4.2 Constraints

At least one of the `uri` property (§3.4.3) or the `index` property (§3.4.5) **SHALL** be present. In certain circumstances (see §3.4.4 and §3.4.5), they **MAY** both be present.

> NOTE: Providing both `uri` and `index` makes the log file more readable at the expense of increased size. Providing only `index` reduces log file size but makes it less readable to an end user, who has to determine the URI by locating the `artifact` object (§3.24) at the index within `theRun.artifacts` (§3.14.15) specified by `index`.

## ~~3.3.2~~ If both `uri` and `index` are present, they SHALL both denote the same artifact. That is, let $URI_1$ be the fully resolved URI of the artifact specified by an `artifactLocation` object as determined by the `uriBaseId` resolution procedure described in §3.4.4. Let $URI_2$ be the fully resolved URI of the artifact specified by the `artifact` object indicated by `index`, determined in the same way. Then $URI_1$ and $URI_2$ SHALL be equivalent in the sense described in §3.10.1~~uri property~~

### ~~3.3.2.1 General~~

~~A `fileLocation` object **SHALL**~~.

## 3.4.3 uri property

Depending on the circumstances, an `artifactLocation` object either **SHALL**, **SHALL NOT**, or **MAY** contain a property named `uri` whose value is a string containing a URI ~~reference (the term used in~~ [RFC3986] ~~to describe either an absolute URI or a relative reference).~~

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

> ~~NOTE 1: A URI reference (even a relative reference) 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.~~

~~A URI reference~~ that specifies ~~a nested file **SHALL** consist of a URI reference to the outermost parent, together with a fragment that describes the nesting of the file within its parent or parents. The fragment **SHALL** begin with a forward slash character ("/") to emphasize that it represents the complete path to the nested file within its container. This requirement allows SARIF consumers to look up the URI of a nested file in the dictionary contained in `run.files` (§).~~the location of the artifact.

~~Two URI references **SHALL** be considered equivalent if their normalized forms are the same, as described in [].~~

> ~~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.~~

When two URI references are not equivalent in this sense (that is, when their normalized forms are not the same), we will say that they are "distinct."

Aside from normalization, SARIF producers **SHALL NOT** make any other changes to the text of a URI reference; for example, they **SHALL NOT** convert the 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 URI references in the log file would not match the path names on the destination system.

#### 3.3.2.2 URIs that use the "file" protocol

If a URI uses the `"file"` protocol [RFC8089] and the specified path is network-accessible, the SARIF producer **SHALL** include the host name.

EXAMPLE 1: A file-based URI that references a network share.

```
file://build.example.com/drops/Build-2018-04-19.01/src
```

If a URI uses the `"file"` protocol and the specified path is *not* network-accessible, the SARIF producer **SHOULD NOT** include the host name.

EXAMPLE 2: A file-based URI that references the local file system.

```
file:///C:/src
```

A SARIF post-processor **MAY** choose to remove the host name If `thisObject` describes a nested artifact whose location within its parent container can be expressed by a path from the root of the container, then if `uri` is present, it **SHALL** specify a relative reference [RFC3986such a URI, ] expressing that path. If the nested artifact is a member of an archive file (for example, zip [ZIP] or tar [TARfor security reasons. If it does so, then to maximize interoperability with previous version of the URI specification, it]), `uri` **SHOULD** specify the URI with leading `"//"`,member name or path as specified by the archive.

NOTE 1: `uri` does not have to begin with `"/"`. Archive formats such as zip and tar support both relative and absolute paths. For example, `"/a.txt"` and `"a.txt"` can both exist as distinct files in the same archive.

If `thisObject` occurs as the value of a "top-level" property in `theRun.originalBaseIds` (§3.14.14EXAMPLE 2.), then `uri` **MAY** be absent. See §3.14.14 for more information onan explanation and an example of this point. Otherwise:

If `index` (§3.4.5) is absent, `uri` **SHALL** be present.

NOTE 2: This ensures that there is a way to locate the artifact specified by the `artifactLocation` object.

If `thisObject` represents a nested artifact whose location within its parent container can be expressed only by means of a byte offset, then `uri` **SHALL NOT** be present.

NOTE 3: This implies that `index` will be present; see §3.4.5.

Otherwise, `uri` **MAY** be present.

### 3.3.33.4.4 uriBaseId property

IfIf this `artifactLocation` object describes a top-level artifact and the value of its `uri` property (§3.4.3) is a relative reference, a `fileLocation`the `artifactLocation` object **SHOULD** contain a property named `uriBaseId` whose value is a string which indirectly specifies the absolute URI with respect to which that relative reference is interpreted. If the `uri` property contains an absolute URI, the uriBaseId property **SHALL** be absent. If this artifactLocation object describes a nested artifact, uriBaseId **SHALL** be absent.

To avoid ambiguity in interpreting the property names (§) in run.files (§), the uriBaseId property **SHALL NOT** contain the character "#".

If a SARIF consumer requires an absolute URI (for example, to display the specified fileartifact to a user), then it needs to have the necessary information to resolve the uriBaseId property to an absolute URI, which it can then be combinedcombine with the relative reference stored in the uri property. One possibility is for the

A SARIF producer and consumers to agree on the meanings of any valuesconsumer **SHALL** use the following procedure to resolve a uriBaseId to an absolute URI:

1. If the end user has configured the SARIF consumer with a value for the uriBaseId property that appear in the log file. Another possibility is for the end user to supply those meanings to the consumer, either(for example, on the consumer's command line, or through a user interface prompt), then the consumer **SHALL** use the configured value.

   EXAMPLE 1: In this example the SARIF consumer's command line specifies that any uriBaseId property whose value is "SRCROOT" refers to the absolute URI "file:///C:/browser/src":/":

```
C:> SarifAnalyzer --input log.sarif --uriBaseId SRCROOT="file:///C:/browser/src"/"
```

2. If uriBaseId is not yet resolved and theRun.originalUriBaseIds (§3.14.14) is present, the consumer **SHALL** attempt to resolve the uriBaseId from the information in originalUriBaseIds, in the manner specified in §3.14.14.
3. If uriBaseId is not yet resolved, the consumer **MAY** use other information or heuristics to locate the artifact.

The uriBaseId property can 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 might. The SARIF producer and any SARIF consumers need to agree on the meanings of any values for the uriBaseId property that appear in the log file.

EXAMPLE 2: In this example, the analysis tool has set the uri property of a fileLocationan artifactLocation object (§3.4) to a relative reference. The tool has also set the uriBaseId property to "%srcroot%". The analysis tool and the SARIF consumers have agreed upon a convention whereby this indicates that the relative reference is expressed relative to the root of the source tree in which the file appears.

```
"fileLocation""artifactLocation": {
  "uri": "drivers/video/hidef/driver.c",
  "uriBaseId": "%srcroot%"
}
```

NOTE: There are various reasons for providing the uriBaseId property:

- Portability: A log file that contains relative references together with uriBaseId properties can be interpreted on a machine where the files are located at a different absolute location.
- Determinism: A log file that uses uriBaseId 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. For more information on this point, see Appendix FAppendix F, "Producing deterministic SARIF log files"..
- Security: The use of uriBaseId 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 `uriBaseId` property might be shorter than the absolute path it represents.~~

For more guidance on the intended use of the `uriBaseId` property, see §3.4.7.

### 3.4.5 index property

Depending on the circumstances, an `artifactLocation` object either **MAY**, **SHALL NOT**, **SHALL**, or **SHOULD** contain a property named `index` whose value is the array index (§3.7.4) within `theRun.artifacts` (§3.14.15) of the `artifact` object (§3.24), if any, that describes the artifact specified by this `artifactLocation` object.

If `thisObject` occurs as the `location` property (§3.24.2) of an `artifact` object in `theRun.artifacts`, then `index` **MAY** be present. If present, it **SHALL** equal the array index within `theRun.artifacts` of the containing `artifact` object.

Otherwise, if `theRun.artifacts` is absent or does not contain an element that describes the artifact specified by `thisObject`, then `index` **SHALL NOT** be present.

> NOTE 1: `index` cannot be present in this case because there is no array element for it to point to. But this implies that `uri` is present, because otherwise there would be no way to locate the artifact specified by `thisObject`.

Otherwise, if the `uri` property (§3.4.3) is absent, then `index` **SHALL** be present.

> NOTE 2: Again, this ensures that there is a way to locate the artifact specified by `thisObject`.

Otherwise (that is, if `uri` is present but there *is* a relevant `artifact` object in `theRun.artifacts`), `index` **SHOULD** be present.

> NOTE 3: If `index` is absent, the SARIF consumer will not be able to locate the additional information contained in the `artifact` object about the artifact specified by `thisObject`.

> EXAMPLE: In this example, `results[0].locations[0].physicalLocation.artifactLocation.index` specifies the `artifact` object located at `artifacts[0]`.

```
{                                    # A run object (§3.14).
  "artifacts": [
    {
      "location": {
        "uri": "file:///C:/Code/main.c"
      },
      "sourceLanguage": "c",
    }
  ],
  "results": [
    {
      "ruleId": "CA2101",
      "level": "error",
      "locations": [
        {
          "physicalLocation": {
            "artifactLocation": {
              "uri": "file:///C:/Code/main.c",
              "index": 0
            },
```

  

```
            "region: {
              "startLine": 24,
              "startColumn": 9
            }
          }
        }
      ]
    }
  ]
}
```

## 3.4.6 description property

An artifactLocation object **MAY** have a property named description whose value is a message object (§3.11) that describes this location.

> EXAMPLE: In this example, the property values in run.originalUriBaseIds (§3.14.14), which are artifactLocation objects, have description properties. This allows a SARIF viewer to display helpful information when prompting a user to supply values for the base id symbols.

```
{                                                # A run object (§3.14).
  "originalUriBaseIds": {                        # See §3.14.14.
    "PROJROOT": {
      "uri": "file:///C:/browser/",
      "description": {
        "text": "The project root directory."
      }
    },
    "SRCROOT": {
      "uri": "file:///C:/browser/src/",
      "description": {
        "text": "The root of the source code tree."
      }
    },
    "BINROOT": {
      "uri": "file:///C:/browser/src/",
      "description": {
        "text": "The build output directory."
      }
    }
  },
}
```

## ~~3.3.4~~3.4.7 Guidance on the use of ~~fileLocation~~artifactLocation objects

Some URIs are "deterministic" in the sense that they will be the same from one run to the next and are independent of machine-specific information such as volume names or drive letters. Internet addresses are typically deterministic.

In contrast, file system paths are typically non-deterministic. For example, a source code enlistment might exist at different paths on different machines.

~~fileLocation~~artifactLocation objects **MAY** represent both deterministic and non-deterministic URIs. ~~**The**~~ In either case, the uri property (§3.4.3) **SHOULD** ~~contain~~ be deterministic, either because it is a deterministic relative reference ~~that *is* deterministic,~~ (for example, the relative path to a file from the root of ~~a~~ the directory tree containing the analyzed source code ~~enlistment to the file. The~~) or because it is an absolute URI. If the URI is non-deterministic, the uriBaseId property (§3.4.4) **SHOULD** capture the non-deterministic portion of the URI, for example, the absolute path to the root of the directory tree containing the analyzed source code ~~enlistment~~.

  

EXAMPLE: In this example, the location of a result detected by a tool is specified by a relative reference together with a `uriBaseId` that specifies the root of the source code enlistment.

```
{                                                 # A run object (§3.14).
  "originalUriBaseIds": {                         # See §3.14.14.
    "SRCROOT": {
      "uri": "file:///C:/browser/src"/"
    }
  },

  "results": [                                    # See §3.14.23.
    {                                             # A result object (§3.27).
      "locations": [                              # See §3.27.12.
        {                                         # A location object (§3.28).
          "physicalLocation": {                   # See §3.28.3.
            "fileLocation": {                     # A
fileLocationartifactLocation": {                 # An artifactLocation object.
              "uri": "ui/window.cpp",
              "uriBaseId": "SRCROOT"
            }
          }
        }
      ]
    }
  ]
}
```

## 3.43.5 String properties

### 3.5.1 Localizable strings

### 3.4.1 Certain string-valued properties in this specification, for example, `toolComponent.name` (§3.19.8General

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

### 3.4.2 Redaction-aware string properties

), can be translated into other languages. We describe these properties as being "localizable." The description of every localizable property will state that it is localizable.

### 3.5.2 Redactable strings

Certain string-valued properties in this specification (for example, `invocation.commandLine` (§3.20.2)) might contain sensitive information that a SARIF producer or a SARIF post-processor might choose to redact. We describe these properties as being "redaction-aware.""redactable." The description of every redaction-awareredactable property will state that it is redaction-awareredactable.

If a SARIF producer or a SARIF post-processor chooses to redact sensitive information in a redaction-awareredactable property, it **SHALL** replace the sensitive information with thea string whose value is provided by the `run.redactionToken` propertyan element of `theRun.redactionTokens` (§3.14.28).

### 3.4.33.5.3 GUID-valued string propertiesstrings

Certain string-valued properties in this specification provide unique stable identifiers in the form of a GUID or UUID [RFC4122]. This document uses the term "GUID".

EXAMPLE: `"f81d4fae-7dec-11d0-a765-00a0c91e6bf6"`

  

NOTE: The UUID standard [RFC4122RFC4122] allows hex digits in either upper or lower case. It does not permit delimiters such as curly braces ("{", "}") around the value.

The description of every GUID-valued property will state that it is GUID-valued.

### 3.4.43.5.4 Hierarchical strings

#### 3.4.4.13.5.4.1 General

Certain string-valued properties and certain property names in this specification (for example, the value of the `run.automationLogicalId``runAutomationDetails.id` property (§3.17.3), and the property names in a property bag (§3.8)) are said to be "hierarchical"." This means that the string consists of a sequence of forward-slash-separated components, with this syntax:

```
hierarchical string = component, { "/", component };

component = { component character, { component character };

component character = ? JSON string character ? - "/";
```

NOTE 1: The grammar prohibits a `component` from containing a forward slash. There is no escape mechanism to allow a `component` to include a forward slash.

For examples, see §3.8.2 and §3.17.3.

The description of every hierarchical string will state that it is hierarchical.

A SARIF consumer **SHALL** interpret the values of a hierarchical string as forming a logical hierarchy. The first component represents the top level of the hierarchy, the second component represents the second level, and so on.

NOTE 2: A hierarchical string does not need to include any forward slashes. The syntax permits a single string of non-forward-slash characters. The purpose of this section is to define the semantics of the forward slash character in those properties that respect it.

In string-valued properties and property names that are *not* described as hierarchical, the forward slash character has no special meaning, and a SARIF consumer **SHALL NOT** interpret it as dividing the value into hierarchical components.

#### 3.4.4.23.5.4.2 Versioned hierarchical strings

Certain hierarchical strings in this specification (for example, the property names in `result.fingerprints` (§3.27.16) and `result.partialFingerprints` (§3.27.17)) are said to be "versioned." This means that if the last `component` of the string is of the form

```
version component = "v", non negative integer;
```

then a SARIF consumer **SHALL** consider that component to represent the version number of the entity specified by the string.

The description of every versioned hierarchical string will state that it is versioned.

In string-valued properties and property names that are described as hierarchical but *not* as versioned, a final `component` matching the syntax of `version component` has no special meaning, and a SARIF consumer **SHALL NOT** interpret it as a version number.

NOTE 1: A versioned hierarchical string does not need to include a version component. The syntax permits but does not require it.

A hierarchical string without a version component **SHALL** be considered older than any corresponding string with a version component.

EXAMPLE: In this example, the partial fingerprint whose property name is `"prohibitedWordHash"` is considered to have been computed with an older version of the “prohibited word hash” algorithm than the partial fingerprint whose property name is "prohibitedWordHash/v1".

```
{                                  # A result object (§3.27).
  "partialFingerprints": {         # See §3.27.17.
    "prohibitedWordHash": "4efcc21977b55",
    "prohibitedWordHash/v1v2": "097886bc876fe"
  }
}
```

NOTE 2: When a previously unversioned string is later versioned, as in the example above, it might be clearer to specify "v2" for the first explicitly versioned string.

## 3.53.6 Object properties

Certain properties in this specification are defined to be JSON objects whose property names satisfy certain conditions. Examples are run.filesoriginalUriBaseIds (§3.14.14) and rulereportingDescriptor.messageStrings (§3.49.11). 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.

## 3.63.7 Array properties

### 3.6.13.7.1 General

Certain properties in this specification are defined to be JSON arrays. Examples are the invocation.toolNotificationstoolExecutionNotifications property (§3.20.21) and the file.hashesproperty bag tags property (§3.8.2). 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.).

### 3.7.2 Default value

If an array-valued property is absent, it **SHALL** default to an empty array unless the property’s description specifies otherwise.

### 3.6.23.7.3 Array properties with unique values

Certain array-valued properties in this specification are described as having “unique” elements. When a property is so described, it means 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 the JSON Schema standard [JSCHEMA01], §4.3, “Instance equality”. In particular, two strings are considered equal when they consist of the same sequence of Unicode [UNICODE12] code points.

### 3.7.4 Array indices

If any property in this specification is described as an “array index,” it **SHALL** contain an integer that is a zero-based index into the specified array. If any such property is absent, it **SHALL** default to -1, which indicates that the value is unknown (not set), unless the property’s description specifies otherwise.

## 3.73.8 Property bags

### 3.7.13.8.1 General

Certain properties in this specification are defined to be “property bags”. A property bag is a JSONan object (§3.6) containing an arbitraryunordered set of properties with arbitrary names.

  

The property names are hierarchical strings (§3.5.4). The components of the property names **SHOULD** be camelCase strings, but see Appendix D for exceptions.

The property values **MAY** be of any JSON type, including strings, numbers, arrays, objects, Booleans, and null. If a property value is a string, it **MAY** be an empty string.

In addition to those properties that are explicitly documented, every object defined in this specification **MAY** contain a property named `properties` whose value is a property bag. This allows SARIF producers to include information about each object that is not explicitly specified in the SARIF format.

### ~~3.7.2~~3.8.2 Tags

#### ~~3.7.2.1~~3.8.2.1 General

If a property bag contains a property named `tags`, the property value **SHALL** be an array of zero or more unique (§3.7.3), hierarchical (§3.5.4) strings ~~(§).~~. Two strings **SHALL** be considered the same if they consist of the same sequence of Unicode [UNICODE12] code points.

Tags **SHOULD NOT** be used to label a result or a rule as belonging to a category in a classification system such as the Common Weakness Enumeration [CWE™] (for example, by adding a tag `"CWE/622"`). Instead, taxonomies (§3.19.3~~The strings in the `tags` array are hierarchical (§).~~

) **SHOULD** be used for this purpose.

Even when defining a custom classification system used within an engineering team, taxonomies **SHOULD** be used rather than tags when labeling a result or a rule.

> EXAMPLE 1: Rather than adding the tag `"shipBlocking"` to a result, consider defining a taxonomy such as "Shipping Impact". This enables metadata such as a description and a help URI to be associated with each taxonomic category.
>
> EXAMPLE 2: In this example, the SARIF producer ~~categorizes scan results according to the Common Weakness Enumeration taxonomy [].~~tags an artifact with the string `"openSource"`.

```
{                                    # A resultrun object (§3.14).
  "artifacts": [                     # See §3.14.15.
    {                                # An artifact object (§3.24).
      "location": {                  # See §3.24.2 "ruleId": "CA2124",
  ...
.
        "uri": "http://www.example.com/libraries/jsonParser.js"
      },
      "properties": {
        "tags": [
      "CWE/22    "openSource"
        ]
      }
}
    ]
  },
  ...
}
```

> NOTE: Anything a tag expresses can also be expressed with a named property bag entry, for example `"openSource": true`, but a tag is more concise.

#### ~~3.7.2.2~~3.8.2.2 Tag metadata

A SARIF log file **MAY** provide additional information about any tag value by including a property whose name is the same as that tag value~~,~~ and whose value is any JSON value. If present, this property **SHALL** be located ~~either~~by searching first in the ~~same~~ property bag that contains the tag, ~~or~~and then in the property bag of any SARIF element which lexically contains the element containing `run` object (§3.14the tag) `theRun`, if any.

EXAMPLE: Suppose a SARIF-producing tool classifies results according to 1: Continuing the example from §3.8.2.1Common Weakness Enumeration, using a tool-specific convention that the tag "`CWE/n`" denotes a ,result to which CWE *n* applies. Suppose this tool produces the following result:

```
{                               # A result object (§)
  "ruleId": "SEC0251",
  "message": {
    "text": "The path 'data/../bin' is not within the 'data' directory"
  },
  "properties": {
    "tags": [
      "security",
      "CWE/22"
    ]
  }
}
```

Now suppose the tool wishes to provide additional information about CWE 22.using open source code. It might provide that information within the property bag containing the tag (in this example, the property bag belonging to the `result``artifact` object):

```
{                               # A resultAn artifact object (§3.24)).
  "ruleId": "SEC0251",location": {
  "message": {
    "text": "The path 'data/../bin' is not within the 'data' directory"
    "uri": "http://www.example.com/libraries/jsonParser.js"
  },
  "properties": {
    "tags": [
      "security",openSource"
      "CWE/22"],
    },"openSource": {
    "CWE/22": {  "informationUri":
      "description": "Improper Limitation of a Pathname",
      "url": "https://cwe.mitre.org/data/definitions/22
"http://www.example.com/procedures/usingOpenSource.html"
    }
  }
}
However, there}
```

EXAMPLE 2: There might be several results associated with CWE 22.open source files. To avoid duplicating the metadatainformation, the tool might choose to place itthe tag metadata in the property bag belonging to the `theRun`:

```
{                               # A run object (§3.14) that lexically contains
the result object:).
{                               # A run  "artifacts": [
    {                           # An artifact object (§3.24(see §)).
  "results": [
    {
      "ruleId": "SEC0251",
      "message": {
        "text": "The path 'data/../bin' is not within the 'data' directory"
      },
        "location": {
        "uri": "http://www.example.com/libraries/jsonParser.js"
      },
      "properties": {
```

  

```
        "tags": [
          "openSource"
        ]
      }
    },
    ...
  ],
  ...
  "properties": {
        "tags": [
          "security",
          "CWE/22"
        ]
      }
    }
  ],
  "properties": {              # The run object's property bag of the
containing run.
    "CWE/22openSource": {
      "description": "Improper Limitation of a Pathname",
      "url": "https://cwe.mitre.org/data/definitions/22.html"
    }
  }
}
```

```
      "informationUri":
        "http://www.example.com/procedures/usingOpenSource.html"
    }
  }
}
```

## 3.83.9 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 the ISO standard for date and time formats [ISO8601:2004]:

```
date time = date, [ "T", time, "Z" ] (* UTC time *);

date = year, "-", month, "-", day;

year = 4 * decimal digit;

month = 2 * decimal digit (* from 01 to 12 *);

day = 2 * decimal digit (* from 01 to 31 *);

time = hour, ":", minute, [ ":", second, [ ".", fraction ] ];

hour = 2 * decimal digit (* from 00 to 1224, to represent midnight at the
                            end of a calendar day *);

minute = 2 * decimal digit (* from 00 to 59 *);

second = 2 * decimal digit (* from 00 to 60, to accommodate leap second *);

fraction = decimal digit, { decimal digit };
```

EXAMPLES:
2016-02-08T16:08:

  

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

The time component of every date/time-valued property **SHALL** be expressed in Coordinated Universal Time (UTC).

NOTE 1: The name of every date/time-valued property ends in “Utc” to emphasize that requirement.

The time components of date/time-valued properties in property bags (§3.8) **SHOULD** also be expressed in UTC.

NOTE 2: This might not always be possible if the property comes from a source that does not provide time zone information.

A SARIF producer **SHOULD** ~~base the number of~~ **NOT** provide more digits in `fraction` ~~on~~than warranted by the precision of the clock on the computer on which it runs.

A SARIF producer **SHOULD** express date/time properties, except for those that express product release dates, to a precision of at least whole seconds.

## 3.10 URI-valued properties

### 3.10.1 General

Certain properties in this specification specify either an absolute URI or a URI reference (the term used in the URI standard [RFC3986] to describe either an absolute URI or a relative reference). The value of every such property, if present, **SHALL** be a string in the format specified by the standard [RFC3986].

If a URI reference refers to a file stored in a version control system (VCS), its value **SHALL** include sufficient information (for example, a commit id) to enable the correct version of the target file to be retrieved from the VCS. If a URI reference refers to a file stored on a physical file system, it **MAY** be specified as a relative reference that omits root information details (such as hard drive letter and an arbitrarily named root directory associated with a source code enlistment).

NOTE 1: A URI reference (even a relative reference) 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.

The URI **SHALL** specify the location of the artifact at the time the analysis was performed.

Two URI references **SHALL** be considered equivalent if their normalized forms are the same, as described in the standard [RFC3986].

NOTE 2: Features of this normalized form include using upper-case hexadecimal digits for percent-encoded characters and expressing the scheme component in lower-case. For the full specification of the normalized URI form, see the standard [RFC3986].

For additional normalization requirements for URIs that use the `"file"` scheme, see §3.10.2.

When two URI references are not equivalent in this sense (that is, when their normalized forms are not the same), we will say that they are “distinct.”

Aside from normalization, SARIF producers **SHALL NOT** make any other changes to the text of a URI reference; for example, they **SHALL NOT** convert the 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 Microsoft 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 URI references in the log file would not match the path names on the destination system.

### 3.10.2 Normalizing file scheme URIs

If a URI uses the `"file"` scheme [RFC8089] and the specified path is network-accessible, the SARIF producer **SHALL** include the host name.

EXAMPLE 1: A file-based URI that references a network share.

```
file://build.example.com/drops/Build-2018-04-19.01/src
```

If a URI uses the `"file"` scheme and the specified path is *not* network-accessible, the SARIF producer **SHOULD NOT** include the host name.

EXAMPLE 2: A file-based URI that references the local file system.

```
file:///C:/src
```

A SARIF producer **MAY** choose to omit the hostname (authority) from a file URI, for example, for security reasons. If it does so, then to maximize interoperability with previous versions of the URI specification, the URI **SHOULD** start with `"file:///"`, as in EXAMPLE 2. See the standard [RFC8089] for more information on this point.

SARIF producers **SHALL** create `"file"` scheme URIs by means of the following procedure or any procedure with the same result:

1. In the case of a direct producer, preserve the file system's casing, even if the file system is case-insensitive. In the case of a converter (which might not know the file system's casing), preserve the casing specified in the analysis tool's native output file.
2. Remove `"."` path segments.
3. Remove empty path segments.
4. If the path contains `".."` path segments, then in the case of a direct producer, resolve the path to a canonical absolute path, using an appropriate algorithm for the operating system on which the tool ran.

   NOTE 1: This is necessary because, for example, the path `/d1/../f` naively converted to a URI is `file:///d1/../f`, which resolves to `file:///f` according to the URI standard [RFC3986]. But if `/d1` is a symbolic link to the directory `d2/d3`, then the correct URI is `file:///d2/f`.

   NOTE 2: A converter might not have the information necessary to remove `".."` segments. As a result, it might produce file scheme URIs that include `".."` segments.

5. Create a URI from the resulting path.
6. Optionally, divide the resulting URI into a base URI and a relative URI (preserving case in both parts), and create an entry for the base URI in `theRun.originalUriBaseIds` (§3.14.14).

   NOTE 3: URI and path manipulation are complex topics. Many operating systems, languages, and frameworks provide methods to perform these operations, which is preferable to having every SARIF producer reimplement them. For example, in C#, the operation can be performed as follows:

```
using System;
using System.IO;
...
string path = ...;

string fullPath = Path.GetFullPath(path);
var uri = new Uri(fullPath, UriKind.Absolute);
string uriString = uri.AbsoluteUri;
```

  

SARIF consumers **SHALL NOT** normalize `".."` segments out of a path. A consumer **SHALL** treat distinct portions of paths up to and including the rightmost `".."` segment as unique directories on the file system, even if [RFC3986] normalization would produce identical paths.

EXAMPLE 3: Consider the following three URIs:

- `file:///d1/../f1`
- `file:///d1/../f2`
- `file:///d1/d2/../../f3`

A consumer would treat `f1` and `f2` as residing in the same directory. So, for example, if a viewer prompted the user to supply the directory where `f1` resides, it could search for `f2` in the same directory, without prompting again. On the other hand, even though `f3` appears to reside in the same directory as `f1` and `f2`, the viewer would not assume that, and would prompt the user to supply the directory where `f3` resides.

### 3.10.3 URIs that use the sarif scheme

In certain circumstances, a URI can refer to an element of the current SARIF log file (for example, see §3.16.3). Such a URI uses the `sarif` scheme. The `sarif` URI scheme consists of only a scheme (with the value `sarif`) and a path component. The path component is interpreted as a JSON pointer [RFC6901] into the SARIF document containing the URI. The authority, query and fragment URI components **SHALL NOT** be present.

EXAMPLE: The URI `"sarif:/inlineExternalProperties/0"` refers to the 0th element of the array contained in the `inlineExternalProperties` property (§3.13.5) at the root of the log file.

### 3.10.4 Internationalized Resource Identifiers (IRIs)

If a URI-valued property refers to a resource identified by an Internationalized Resource Identifier (IRI) [RFC3987], the SARIF producer **SHALL** first transform the IRI into a URI, using the mapping mechanism specified in §3.1 of the standard [RFC3987], and then assign the transformed value to the property. The string value of a URI-valued property **SHALL NOT** include Unicode characters such as `"é"`; such characters are permitted in IRIs but are not permitted in URIs. §3.1 of the standard [RFC3987] describes how to replace such characters with "percent-encoded" equivalents to produce a valid URI.

EXAMPLE: Suppose a URI-valued property needs to refer to a resource identified by the string `"http://www.example.com/hu/sör.txt"`. This string contains the character `"ö"`, so it is a valid IRI but not a valid URI. Following the procedure in §3.1 of the standard [RFC3987], a SARIF producer would transform this string to the valid URI `"http://www.example.com/hu/s%C3%B6r.txt"` before assigning it to the property.

## ~~3.9~~3.11 message ~~objects~~object

### ~~3.9.1~~3.11.1 General

Certain objects in this specification define messages intended to be viewed by a user. SARIF represents such a message with a `message` object, which offers the following features:

- Message strings in plain text ("plain text messages") (§3.11.3~~")~~).
- Message strings that incorporate formatting information ("~~rich text~~formatted messages") in GitHub Flavored Markdown [GFM] (§3.11.4~~")~~).
- Message strings with placeholders for variable information (§3.11.5~~.~~).
- Message strings with embedded links (§3.11.6).

  

## 3.11.2 Constraints

- At least one of the `text` (§3.11.8) or `id` (§3.11.10~~Localized message strings.~~

) properties **SHALL** be present.

NOTE: This ensures that a SARIF consumer can locate the text of the message.

## ~~3.9.2~~3.11.3 Plain text messages

A plain text 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~~ **SHALL NOT** contain formatting information such as HTML tags.

The message **~~SHOULD NOT~~MAY** contain ~~JSON escaped~~ line breaks ~~("\~~(for example, `"\r"` or `"\n`~~").~~ ~~However,~~`"`, if ~~line~~the SARIF log file is serialized as JSON). Line breaks ~~are present, they~~ **MAY** follow any convention (for example, `"\n"` or `"\r\n"`). A SARIF post-processor **MAY** normalize line breaks to any desired convention, including escaping or removing the line breaks so that the entire message renders on a single line.

The message string **MAY** contain placeholders (§3.11.5) and embedded links (§3.11.6).

If the message consists of more than one sentence, its first sentence **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.

~~Rich text~~A SARIF post-processor **SHOULD NOT** modify line break sequences (except perhaps to adapt them to a particular viewing environment).

## ~~3.9.3~~3.11.4 Formatted messages

### ~~3.9.3.1~~3.11.4.1 General

~~Rich text~~Formatted messages **MAY** be of arbitrary length and **~~SHOULD~~MAY** contain formatting information. The message string **MAY** also contain placeholders (§3.11.5) and embedded links (§3.11.6).

~~Every rich text message in a given run **SHALL** be expressed in the same markup language, specified by the `run.richMessageMimeType` property (§). For maximum interoperability among SARIF log files produced by different tools, direct producers **SHALL** express rich text messages~~Formatted messages **SHALL** be expressed in GitHub-Flavored Markdown [GFM]. Since GFM is a superset of CommonMark [CMARK], any CommonMark Markdown syntax is acceptable.

~~If an analysis tool produces a custom output format that includes rich text messages in a format other than GFM, a converter which translates the output of that tool to SARIF **SHOULD NOT** attempt to translate the messages to GFM. Instead, it **SHOULD** set `run.richMessageMimeType` to a value appropriate to the analysis tool's output format.~~

### ~~3.9.3.2~~3.11.4.2 Security implications

~~If the rich text message format is any variant of Markdown, then for~~For security reasons, SARIF producers and ~~SARIF~~ consumers **SHALL** adhere to the following:

- SARIF producers **SHALL NOT** emit messages that contain HTML, even though all variants of Markdown permit it.

  

- Deeply nested markup can cause a stack overflow in the Markdown processor [GFMENG]. To reduce this risk, SARIF consumers **SHALL** use a Markdown processor that is hardened against such attacks.

  NOTE: One example is the GitHub fork of the cmark Markdown processor [GFMCMARK].

- To reduce the risk posed by possibly malicious SARIF files that do contain arbitrary HTML (including, for example, `javascript:` links), SARIF consumers **SHALL** either disable HTML processing (for example, by using an option such as the `--safe` option in the cmark Markdown processor) or run the resulting HTML through an HTML sanitizer.

SARIF consumers that are not prepared to deal with the security implications of ~~rich text~~formatted messages **SHALL NOT** attempt to render them and **SHALL** instead fall back to the corresponding plain text messages.

### ~~3.9.4~~3.11.5 Messages with placeholders

A message string **MAY** include one or more "placeholders." The syntax of a placeholder is:

```
placeholder = "{", index, "}""}";

index = non negative integer;
```

`index` represents a ~~0~~zero-based index into the array of strings contained in the `arguments` property (§3.11.11).

When a SARIF consumer displays the message, it **SHALL** replace every occurrence of the placeholder `{n}` with the string value at index `n` in the `arguments` array ~~(§)~~.. Within both plain text and ~~rich text~~formatted message strings, the characters "`{`" and "`}`" **SHALL** be represented by the character sequences "`{{`" and "`}}`" respectively.

Within a given `message` object:

- The plain text and ~~rich text~~formatted message strings **MAY** contain different numbers of placeholders.
- A given placeholder index **SHALL** have the same meaning ~~across all~~in the plain text and formatted message strings ~~in the object~~ (so ~~that~~ they can be replaced with the same element of the `arguments` array).

  EXAMPLE 1: Suppose a `message` object's `text` property (§3.11.8) contains this string:

  ```
  "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 at least two elements, the first corresponding to `{0}` and the second corresponding to `{1}`.

  EXAMPLE 2: In this example, the SARIF consumer will replace the placeholder `{0}` in `message.text` with the value `"pBuffer"` from the 0 element of `message.arguments`.

```
{                                                    # A run object (§3.14).
  "results": [                                       # See §3.14.23.
    {                                                # A result object (§3.27).
      "ruleId": "CA2101",                            # See §3.27.5.
      "message": {                                   # See §3.27.11.
        "text": "Variable '{0}' is uninitialized.",  # See §3.11.8.
        "arguments": [ "pBuffer" ]                   # See §3.11.11.
      }
    }
  ]
}
```

  

## 3.9.53.11.6 Messages with embedded links

A message string **MAY** include one or more links to locations within filesartifacts mentioned in the enclosing `result` object (§3.27). We refer to these links as "embedded links".

Within a rich textformatted message (§3.11.4), an embedded link **SHALL** conform to the syntax of a GitHub Flavored Markdown link (see [GFM], §6.6, "Links"), with the restriction that the "link destination" **SHALL** be a non-negative integer (whose interpretation is defined below).

NOTE 1: The GFM link syntax is very flexible. Since a SARIF viewer that renders rich textformatted messages will presumably rely on a full-featured GFM processor, there is no need to restrict the embedded link syntax in SARIF rich textformatted messages.

Within a plain text message (§3.11.3), an embedded link **SHALL** conform to the following syntax (which is a greatly restricted subset of the GFM link syntax) before JSON encoding:

```
escaped link character = "\" | "[" | "]""]";

normal link character = ? JSON string character ? – escaped link character;

link character = normal link character | ("\", escaped link character));

link text = { link character }};

link destination = non-negative integer;? Any valid URI ?;

embedded link = "[", link text, "](", link destination, ")";
```

`link text` is the message text visible to the user.

Literal square brackets ("`[`" and "`]`") in the link text of a plain text message **SHALL** be escaped with a backslash ("`\`"). Since JSON itself treats the backslash as an escape character, the backslash **SHALL** be doubled.

NOTE 2: When a SARIF log file is serialized as JSON, JSON encoding doubles the backslash.

EXAMPLE 1: Consider this embedded link whose link text contains square brackets and backslashes:

```
"message": {
  "text": "Prohibited term used in [para\\[0\\]\\\\spans\\[2\\](1)." # See §
}
```

A SARIF viewer would render it as follows:

Prohibited term used in para[0]\spans[2].

Literal square brackets and (doubled) backslashes **MAY** appear anywhere else in a plain text message without being escaped.

The `message` object's containing `result`If `link destination` is a non-negative integer, it **SHALL** refer to a `location` object (§3.28) **SHALL** contain exactly one `physicalLocation` object (§) whose `id` property (§3.28.2) equals the value of `link destination`.

NOTE: `link destination` is required to be an integer, rather than arbitrary string, to avoid confusion In this case, `theResult` **SHALL** contain exactly one `location` object with normal Markdown link syntax.that `id`.

NOTE 3: Negative values are forbidden because their use would suggest some non-obvious semantic difference between positive and negative values.

EXAMPLE 2: In this example, a plain text message contains an embedded link to a location with a file. There isThe `result` object contains exactly one

  

~~physicalLocation~~location object whose id property matches the link destination.

```
{                                  # A result object (§3.27{
  "version": "2.0.0",
  "runs": [
    {
      "results": [
        {
        )
  "ruleId": "TNT0001",
  "message": {
    "text": "Tainted data was used. The data came from [here](3)."
  },
  "locations": [
    {
      "physicalLocation": {
        "uri": "file:///C:/code/main.c",
        "region": {
          "startLine": 15,
          "startColumn": 9
        }
      }
    }
  ],
  "relatedLocations": [
    {
      "id": 3,
      "physicalLocation": {
      "id": 3,
        "uri": "file:///C:/code/input.c",
        "region": {
          "startLine": 1525,
          "startColumn": 919
        }
      }
    }
  ]
}
```

The link destination in embedded links in both plain text messages and formatted messages **MAY** use the sarif URI scheme (§3.10.3). This allows a message to refer to any content elsewhere in the SARIF log file.

EXAMPLE 1: A result.message (§3.27.11) can refer to another result in the same run (or, for that matter, in another run within the same log file) as follows:

"There was [another result](sarif:/runs/0/results/42) found by this code flow."

A SARIF viewer executing in an IDE might respond to a click on such a link by selecting the target result in an error list window and navigating the editor to that result's location.

Because the "sarif" URI scheme uses JSON pointer [RFC6901], which locates array elements by their array index, these URIs are potentially fragile if the SARIF log file is transformed by a post-processor.

EXAMPLE 2: If a post-processor concatenates two runs into a single log file, the links within the run at index 1 will be incorrect, and will need to be updated from "sarif:/runs/0/…" to "sarif:/runs/1/…".

EXAMPLE 3: If a post-processor removes results from a run, any links that refer to results at indices following the removed results will need to be adjusted. For example, sarif:/runs/0/results/54 might need to be adjusted to sarif:/runs/0/results/42.

  

When a tool displays on the console a result message containing an embedded link, it **MAY** reformat the link (for example, by removing the square brackets around the `link text`). If the `link destination` is an integer, and hence specifies a `location` object belonging to `theResult`, the tool **SHOULD** replace the integer with a string representation of the specified location.

```
EXAMPLE 4: Suppose a tool chooses to display the result message from Example 3, which
contains an integer-valued link destination, on the console. The output might be:

Tainted data was used. The data came from here: C:\code\input.c(25
}
      }
    }
  }
}
```

`, 19).`

Note that in addition to providing a string representation of the location, the tool removed the `[…](…)` link syntax and separated the link text from the location with a colon. Finally, the tool recognized that the location's URI used the `file` scheme and chose to display it as a file system path rather than a URI.

### 3.9.63.11.7 Message string resourceslookup

#### 3.9.6.1 General

A `message` object can directly contain message strings in its `text` (§3.11.8) and `richTextmarkdown` (§3.11.9) properties. It can also indirectly refer to message strings through its `messageIdid` (§3.11.10) and `richMessageId` (§) properties. We refer to these indirectly referenced message strings as "message string resources," and we refer to the contents of the `messageId` and `richMessageId` properties as "resource identifiers."

The resource identifiers used for the values of `messageId` and `richMessageId` properties **SHALL** be distinct. That is, any given resource identifier **SHALL NOT** appear both as the value of a `messageId`) property and the value of a `richMessageId` property in the same run.

Resources enable message strings to be localized into other languages. A SARIF `run` object (§) can optionally contain the message string resources for a single language, namely the language designated by its `tool.language` property (§). We refer to these message strings as "embedded resources." Embedded When a SARIF consumer needs to locate a message string from a `message` string resources are stored in the `run.resources.messageStrings` property (§).

If a SARIF consumer needs to access resources for a language other than the one specified by `tool.language`, it can attempt to locate the resources in an external file. We refer to such a file as a "SARIF resource file", and we refer to the message strings in such a file as "external resources." § defines the naming convention and file lookup procedure for SARIF resources files. § defines the SARIF resource file format.

#### 3.9.6.2 Embedded string resource lookup procedure

When a SARIF consumer needs to locate a message string for the run's declared objectlanguage, it **SHALL** follow the string lookup procedure specified in this section. The `run` object **SHALL** contain enough information for the string lookup procedure to succeed. This ensures that a SARIF consumer can always locate the message strings for the declared language without having to consult a SARIF resource file, which might not be available. The string lookup procedure depends on whether the consumer can render rich text messagesprocedure to succeed.

The lookup **SHALL** occur entirely within the context of a single `toolComponent` object (§3.19) which we refer to as `theComponent`. If the SARIF consumer is displaying messages in the language specified by `theRun.language` (§3.14.7), then `theComponent` is the tool component that defines the message. If the consumer is displaying messages in any other language – in which case a translation (§3.19.4If the consumer can render rich text messages, the string lookup procedure is:

1. If message.richText is present, use its value.

Otherwise, if ) is in use – then theComponent is the tool component that contains the translation.

In this procedure, we refer to the message.richMessageId object whose string is being looked up as theMessage.

At various points in this procedure, we state that the consumer uses an object's "text property or markdown property, as appropriate." This means that if the consumer can render formatted messages, it **MAY** use the markdown property, if present; otherwise it **SHALL** use the text property, but if the consumer cannot render formatted messages, it **SHALL** use the text property.

The procedure is:

IF theMessage.text is present, and the desired language is theRun.language THEN

Use the text or markdown property of theMessage as appropriate.

IF the string has not yet been found THEN

IF theMessage occurs as the value of result.message (§3.27.11) THEN

LET theRule be the reportingDescriptor object (§3.49), an element of theComponent.rules (§3.19.23), which defines the rule that was violated by this result.

IF theRule exists AND theRule.messageStrings (§3.49.11) is present AND contains a property whose name equals theMessage.id THEN

LET theMFMS be the multiformatMessageString object (§3.12) that is the value of that property.

Use the text or markdown property of theMFMS as appropriate.

ELSE IF theMessage occurs as the value of notification.message (§3.58.5) THEN

LET theDescriptor be the reportingDescriptor object (§3.49), an element of theComponent.notifications (§3.19.23run.resources), which describes this notification.

IF theDescriptor exists AND theDescriptor.messageStrings is present andAND contains a property whose name matches message.richMessageId, useequals theMessage.id THEN

LET theMFMS be the multiformatMessageString object that is the value of that property.

2. Otherwise, execute the lookup procedure for plain text messages, below.

If the consumer cannot render rich text messages, Use the text or markdown property of theMFMS as appropriate.

IF the string lookup procedure is:has not yet been found THEN

1. IF theComponent.globalMessageStrings (§3.19.22If message.text) is present, use its value.

Otherwise, if message.messageId is present, and run.resources.messageStrings is present and AND contains a property whose name matches message.messageId, use equals theMessage.id THEN

LET theMFMS be the multiformatMessageString object that is the value of that property.

Otherwise, Use the text or markdown property of theMFMS as appropriate.

IF the string has not yet been found THEN

The lookup procedure fails (which means that the SARIF log file is invalid).

  

#### 3.9.6.3 SARIF resource file lookup procedure

When a SARIF consumer needs to locate a message string for a language other than the tool's declared language, it **SHALL** follow the file lookup procedure specified in this section to locate a SARIF resource file.

SARIF resource file names **SHALL** follow the naming convention defined by the following syntax:

```
SARIF resource file name = language tag, ".resources.sarif"

language tag = ? RFC 5646 language tag ?
```

The file lookup procedure is:

1. Determine the "resource URI base" as follows:

   a. If the SARIF consumer is configured to obtain resources from a particular location (for example, by means of a configuration file or a command line argument), that is the resource URI base.

   b. If the resource URI base has not yet been determined, and if `run.tool.resourceLocation` (§) is present:

      i. If `run.tool.resourceLocation.uri` is an absolute URI, that is the resource URI base.

      ii. If the resource URI base has not yet been determined, then if `run.tool.resourceLocation.uriBaseId` is present and `run.originalUriBaseIds` is present and contains a matching property, then the resource URI base is the absolute URI obtained by combining `run.tool.resourceLocation.uri` with the matching property value from `run.originalUriBaseIds`

   c. If the resource URI base has not yet been determined, the SARIF consumer **MAY** use other means to determine it. (For example, it might prompt the user).

   d. If the resource URI base has not yet been determined, the file lookup procedure fails.

2. Locate a SARIF resource file under the resource URI base location as follows:

   a. Construct a file name using the full [] language tag specified by the user. (For example, this might be the operating system's current UI language, such as `fr-FR`. In this case, the file name would be `fr-FR.resources.sarif`.) If a file by that name is present, use it.

   b. Otherwise, if the first subtag is one of the two- or three-letter primary language subtags defined in [], [] or, [], construct a file name using only that subtag. (Continuing the previous example, the file name would be `fr.resources.sarif`.) If a file by that name is present, use it.

   c. If the SARIF resource file name has not yet been determined, the SARIF consumer **MAY** use other means to determine it. (For example, it might prompt the user.)

   d. If the SARIF resource file name has not yet been determined, the file lookup procedure fails.

  

If the file lookup procedure fails, the SARIF consumer **MAY** follow the string lookup procedure for embedded resources specified in §. In that case, the SARIF consumer might display messages in a language other than the one the end user requested. The SARIF consumer **MAY** notify the user if it was unable to locate resources for the requested language.

If the file lookup procedure succeeds, the SARIF consumer **SHALL** follow the string lookup procedure defined in § to extract the required message string from the SARIF resource file.

#### 3.9.6.4 SARIF resource file format

##### 3.9.6.4.1 General

A SARIF resource file contains only that subset of the elements of a SARIF log file that are necessary to describe resources. Some of the elements that are present in a SARIF resource file are constrained differently than they are in a SARIF log file, for example, by being required rather than optional, or by having a different number of array elements. All these differences are described in the sections that follow.

## 3.101.1 sarifLog object

The root element of a SARIF resource file is a `sarifLog` object (§). Its permitted properties, and their differences from the corresponding elements in a SARIF log file, are as follows:

| Property | Type | Required? | Difference from SARIF log file |
|---|---|---|---|
| `$schema` (§3.10.3) | `string` | No | Specifies the absolute URI from which the JSON schema for the SARIF resource file format (rather than the SARIF log file format) can be obtained. |
| `runs` (§3.10.4) | `run[]` (§3.11) | Yes | Array contains exactly one element, rather than one or more. That element contains only the properties specified in §3.9.6.4.3. |

##### 3.10.1.1.1 run object

The permitted properties on the `run` object, and their differences from the corresponding elements in a SARIF log file, are as follows:

| Property | Type | Required? | Difference from SARIF log file |
|---|---|---|---|
| `tool` (§3.11.8) | `tool` (§3.12) | Yes | Required rather than optional. Contains only the properties specified in §3.9.6.4.4. |
| `resources` (§3.11.17) | `resources` (§3.35) | Yes | Required rather than optional. |

##### 3.10.1.1.2 tool object

The permitted properties on the `tool` object, and their differences from the corresponding elements in a SARIF log file, are as follows:

| Property | Type | Required? | Difference from SARIF log file |
|---|---|---|---|
| `name` (§3.12.2) | `string` | Yes | None |
| `fullName` (§3.12.3) | `string` | No | None |

  

| | | | |
|---|---|---|---|
| ~~semanticVersion (§3.12.4)~~ | ~~string~~ | ~~Yes~~ | ~~None~~ |
| ~~version (§3.12.5)~~ | ~~string~~ | ~~No~~ | ~~None~~ |
| ~~fileVersion (§3.12.6)~~ | ~~string~~ | ~~No~~ | ~~None~~ |
| ~~language (§3.12.8)~~ | ~~string~~ | ~~Yes~~ | ~~Required rather than recommended. Just as in a SARIF log file, it specifies the language of the resources embedded in the file.~~ |

#### ~~3.10.1.1.3 resources object~~

~~The `resources` object in a SARIF resource file is identical to the `resources` object in a SARIF log file (§).~~

## ~~3.10.2~~3.11.8 text property

A `message` object **MAY** contain a property named `text` whose value is a non-empty string containing a plain text message (§3.11.3).

## ~~3.10.3~~3.11.9 ~~richText~~markdown property

A `message` object **MAY** contain a property named ~~`richText`~~`markdown` whose value is a non-empty string containing a ~~rich text~~formatted message (§3.11.4) expressed in GitHub-Flavored Markdown [GFM~~).~~].

If the ~~`richText`~~`markdown` property is present, the `text` property (§3.11.8) **SHALL** also be present.

> NOTE: This ensures that the message is viewable even in contexts that do not support the rendering of ~~rich~~formatted text.

SARIF consumers that cannot (or choose not to) render ~~rich~~formatted text **SHALL** ignore the ~~`richText`~~`markdown` property and use the `text` property instead.

## ~~3.10.4~~3.11.10 ~~messageId~~id property

A `message` object **MAY** contain a property named ~~`messageId`~~`id` whose value is a non-empty string containing the ~~resource~~ identifier ~~(§)~~ for the desired ~~plain text~~ message ~~(§).~~. See §3.11.7 ~~and §~~ for details of the ~~resource~~message string lookup procedure.

## ~~3.10.5 richMessageId property~~

~~A `message` object **MAY** contain a property named `richMessageId` whose value is a non-empty string containing the resource identifier (§) for the desired rich text message (§).~~

~~SARIF consumers that cannot (or choose not to) render rich text **SHALL** ignore the `richMessageId` property and use the `messageId` property instead. See § and § for details of the resource string lookup procedure.~~

## ~~3.10.6~~3.11.11 arguments property

If the message string specified by any of the properties `text` (§3.11.8~~), `richText`~~), `markdown` (§3.11.9~~), `messageId` (§~~), or ~~`richMessageId`~~`id` (§3.11.10) contains any placeholders (§3.11.5), the `message` object **SHALL** contain a property named `arguments` whose value is an array of strings. §3.11.5 specifies how a SARIF consumer combines the contents of the `arguments` array with the message string to construct the message that it presents to the end user, and provides an example.

  

If none of the properties `text`, ~~`richText`, `messageId`~~`markdown`, or ~~`richMessageId`~~`id` contains any placeholders, ~~the~~then `arguments` ~~property **SHALL**~~**MAY** be absent.

The `arguments` array **SHALL** contain as many elements as required by the maximum placeholder index among all the message strings specified by the `text`, ~~`richText`, `messageId`, or `richMessageId`~~`markdown`, and `id` properties.

> EXAMPLE: If the highest numbered placeholder in the `text` message string is `{3}` and the highest numbered placeholder in the ~~`richText`~~`markdown` message string is `{5}`, the `arguments` array must contain at least 6 elements.

## ~~3.11~~3.12 ~~sarifLog~~multiformatMessageString object

### ~~3.11.1~~3.12.1 General

A `multiformatMessageString` object groups together all available textual formats for a message string.

### 3.12.2 Localizable multiformatMessageStrings

Certain `multiformatMessageString`-valued properties in this specification, for example, `reportingDescriptor.shortDescription` (§3.49.9), can be translated into other languages. We describe these properties as being "localizable." The description of every localizable property will state that it is localizable.

### 3.12.3 text property

A `multiformatMessageString` object **SHALL** contain a property named `text` whose value is a non-empty string containing a plain text representation of the message.

> NOTE: This property is required to ensure that the message is viewable even in contexts that do not support the rendering of formatted text.

### 3.12.4 markdown property

A `multiformatMessageString` object **MAY** contain a property named `markdown` whose value is a non-empty string containing a formatted message (§3.11.4) expressed in GitHub-Flavored Markdown [GFM].

SARIF consumers that cannot (or choose not to) render formatted text **SHALL** ignore the `markdown` property and use the `text` property (§3.12.3) instead.

## 3.13 sarifLog object

### 3.13.1 General

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

EXAMPLE:

```
{
  "version": "2.~~0~~1.0", # See §3.13.2.
  "runs": [            # See §3.13.4.
    {
      ...              # A run object (§3.14)
    },
    ...
    {
      ...              # Another run object
```

  

```
    }
  ]
}
```

### ~~3.11.2~~3.13.2 version property

A `sarifLog` object **SHALL** contain a property named `version` whose value is a string designating the version of the SARIF ~~format~~specification to which this log file conforms. This string **SHALL** have the value `"2.~~0~~1.0"`.

Although the order in which properties 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.

### ~~3.11.3~~3.13.3 $schema property

A `sarifLog` object **MAY** contain a property named `$schema` whose value is a string containing an absolute URI from which a JSON schema document [JSCHEMA01] 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 **SHALL** describe the version of the SARIF format specified by the `version` property (§3.13.2).

> NOTE 1: 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.

> NOTE 2: The SARIF schema is available at https://raw.githubusercontent.com/oasis-tcs/sarif-spec/master/Schemata/sarif-schema-2.1.0.json.

### ~~3.11.4~~3.13.4 runs property

A `sarifLog` object **SHALL** contain a property named `runs` whose value is either `null` or an array of ~~one~~zero or more `run` objects (§3.14).

The value of `runs` **SHALL** be an array with at least one element except in the following circumstances:

- If a SARIF producer finds no data with which to populate `runs`, then its value **SHALL** be an empty array.

  NOTE 1: This would happen if, for example, the log file were the output of a query on a result management system, and the query did not match any runs stored in the result management system.

- If a SARIF producer tries to populate `runs` but fails, then its value **SHALL** be `null`.

  NOTE 2: This would happen if, for example, the log file were the output of a query on a result management system, and the query was malformed.

### 3.13.5 inlineExternalProperties property

A `sarifLog` object **MAY** contain a property named `inlineExternalProperties` whose value is an array of zero or more unique (§3.7.3) `externalProperties` objects (§4.3).

> NOTE: This property allows multiple runs to share large data sets in a single, self-contained log file.

> EXAMPLE: In this example, two tools analyze the same set of image files, stored in `sarifLog.inlineExternalProperties[0].artifacts`. The first tool locates the

inline `externalProperties` object by means of a URI with the `sarif` scheme (see §3.10.3). The second tool locates the object by means of its `guid` property (§4.3.4).

```
{
  "$schema": "https://raw.githubusercontent.com/oasis-tcs/sarif-spec/master/Schemata/sarif-schema-2.1.0.json",
  "version": "2.1.0",

  "inlineExternalProperties": [
    {
      "guid": "00001111-2222-3333-4444-555566667777", # See §4.3.4.

      "artifacts": [                                  # See §4.3.6.
        {
          "location": {
            "uri": "apple.png"
          },
          "mimeType": "image/png"
        },
        {
          "location": {
            "uri": "banana.png"
          },
          "mimeType": "image/png"
        }
      ]
    }
  ],

  "runs": [                                           # See §3.13.4.
    {                                                 # A run object (§3.14).
      "tool": {                                       # See §3.14.6.
        "driver": {
          "name": "ImageAccessibilityScanner"
        }
      },
      "externalPropertyFileReferences": {             # See §3.14.2.
        "artifacts": [
          {
            "location": {
              "uri": "sarif:/inlineExternalPropertyFiles/0"
            }
          }
        ]
      },
      "results": [
        ...
      ]
    },
    {
      "tool": {
        "driver": {
          "name": "ImageSuitabilityScanner"
        }
      },
      "externalPropertyFileReferences": {
        "artifacts": [
          {
            "guid": "00001111-2222-3333-4444-555566667777"
          }
        ]
      },
      "results": [
        ...
```

  

```
        ]
      }
    ]
}
```

## 3.123.14 run object

### 3.12.13.14.1 General

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

EXAMPLE:

```
{
  "tool": {        # See §3.14.6.
    ......                # A tool object (§3.18).
  },
  "results": [     # See §3.14.23.
    {
      ...          # A result object (§3.27).
    },
    ...
    {
      ...          # Another result object.
    }
  ]
}
```

### 3.12.23.14.2 instanceGuidexternalPropertyFileReferences property

A `run` object **MAY** contain a property named `instanceGuidexternalPropertyFileReferences` whose value is a GUID-valued string an `externalPropertyFileReferences` object (§3.15) which provides a unique, stable identifier for the run.

A result management system or other components) that specifies the locations of the engineering system **MAY** use `run.instanceGuid` to associate the information in the log with additional information not provided by the analysis tool that produced it.

logicalIdexternal property files (see §3.15.2) associated with this log file.

### 3.14.3 automationDetails property

A `run` object **MAY** contain a property named `logicalIdautomationDetails` whose value is a string containing a logical identifier for the run, that is, a string that serves to categorize the run. An engineering system **MAY** categorize runs using any desired classification system. Multiple runs in the same category **SHALL** have the same `logicalId`.

EXAMPLE 1:

```
{
  "logicalId": "Nightly security scanner run"
}
```

`logicalId` is hierarchical (§).

EXAMPLE 2:

```
{
  "logicalId": "Nightly security scanner run/x86/debug"
}
```

  

runAutomationDetailsAn engineering system **MAY** define any number of components and interpret them in any way desired. For example, it might use the components of logicalId to aggregate results from similar runs, such as "all 'Nightly security scanner' runs", or to display a set of runs in a tree view.

### 3.12.3 description property

An run object **MAY** contain a property named description whose value is a message object (§3.17) that describes this run.

For an example, see §3.17.1.

### 3.14.4 runAggregates property

A run object **MAY** contain a property named runAggregates whose value is an array of zero or more unique (§3.7.3) runAutomationDetails objects (§3.17) each of which describes an aggregate of runs to which this run belongs.

For an example, see §3.17.1If logicalId (§) is present, description **SHOULD** describe the type of run defined by logicalId.

EXAMPLE:

```
{                                                         # A run object (§).
  "logicalId": "Nightly security scanner run/x86/debug", # See §.
  "description": {
    "text": "This is the nightly run of the Security Scanner tool on all binaries
             except for test binaries. The scanned binaries are architecture '{0}'
             and build type '{1}'.",
    "arguments": [
      "x86",
      "debug"
    ]
  }
}
```

baselineInstanceGuid.

### 3.12.43.14.5 baselineGuid property

A run object **MAY** contain a property named baselineInstanceGuidbaselineGuid whose value is a GUID-valued string (§3.5.3) which **SHALL** equal the instanceGuidautomationDetails.guid property (§3.14.3, §3.17.4) of some previous run.

If the run object has a logicalId property (§), then the run identified by baselineInstanceGuid **SHALL** have the same value for logicalId.

NOTE: This ensures that only "similar" runs are compared.

If baselineInstanceGuidbaselineGuid is present, the result.baselineState property (§3.27.24) of every result object (§3.27) in the containing run objecttheRun **SHALL** be computed with respect to the run specified by baselineInstanceGuidbaselineGuid.

### 3.12.53.14.6 automationLogicalId tool property

A run object **SHALL** contain a property named tool whose value is a tool object (§3.18) that describes the analysis tool that was run.

### 3.12.63.14.7 language

A run object **MAY** contain a property named automationLogicalIdlanguage whose value is a string containing an identifier that allowsspecifying the language of the localizable strings (§3.5.1) in theRun (except for localizable strings that occur within theRun.translations (§3.14.9run to be correlated with other artifacts produced)), in the format specified by the language tags standard [RFC5646]. If this property is absent, it **SHALL** default to "en-US".

a larger automation process.

`automationLogicalId` is hierarchical (§).

> 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 `automationLogicalId`, allowing the tool run to be associated with other artifacts produced by the build. In this example, the build system takes advantage of the hierarchical nature of `automationLogicalId` to include the name of the build queue ("Nightly") in `automationLogicalId`.

```
{
  "automationLogicalId": "Nightly/14.0.1.2",
  ...
}
```

> architectureEXAMPLE 1: The language is region-neutral English:

```
"language": "en"
```

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

```
"language": "fr-FR"
```

### 3.12.73.14.8 taxonomies property

A `run` object **MAY** contain a property named `taxonomies` whose value is an array of zero or more unique (§3.7.3) `toolComponent` objects (§3.19) each of which represents a standard taxonomy (§3.19.3).

> NOTE: Analysis tools can define their own custom taxonomies; see §3.19.3 and §3.19.25.

### 3.14.9 translations property

A `run` object **MAY** contain a property named `translations` whose value is an array of zero or more unique (§3.7.3) `toolComponent` objects (§3.19`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 ) each of which represents a translation (§3.19.4the 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".

### 3.12.83.14.10 toolpolicies property

A `run` object **SHALLMAY** contain a property named `toolpolicies` whose value is a `tool` objectan array of zero or more unique (§3.7.3) `toolComponent` objects (§3.19) each of which represents a policy (§3.19.5) that describes the analysis tool that was run.).

  

## 3.14.11 invocations property

## ~~3.12.91.1.1 invocations property~~

A `run` object **MAY** contain a property named `invocations` whose value is an array of ~~unique (§)~~zero or more `invocation` objects (§3.20) that together describe ~~the invocation~~a single run of ~~the~~a single analysis tool ~~that was run~~.

> NOTE: Normally, an analysis tool runs as a single process, and the `invocations` array requires only one element. The `invocations` property is defined as an array, rather than as a single `invocation` object, to accommodate tools which execute a sequence of programs to produce results. For example, a tool might run one program to determine the set of ~~files~~artifacts to analyze and another program to analyze those ~~files~~artifacts.

The elements of the `invocations` array **SHOULD**, as far as possible, be arranged in chronological order according to the start time of each process. If some of the processes run in parallel, this might not be possible.

## ~~3.12.10~~3.14.12 conversion property

If a `run` object was produced by a converter, it **MAY** contain a property named `conversion` whose value is a `conversion` object (§3.22) that describes how the converter transformed the analysis tool's native output format into the SARIF format.

A direct producer **SHALL NOT** emit the `conversion` property.

## ~~3.12.11~~3.14.13 versionControlProvenance property

A `run` object **MAY** contain a property named `versionControlProvenance` whose value is an array of ~~one~~zero or  more unique (§3.7.3) `versionControlDetails` objects (§3.23). Each array entry specifies a revision in a repository containing files that were scanned during the run.

> NOTE 1: This property allows an engineering system to reproduce a scan by retrieving the specified revision of the required files from ~~of~~ each repository before repeating the analysis run.

> NOTE 2: This property is an array, rather than a single `versionControlDetails` object, to support scenarios where a tool scans files from multiple repositories in a single run.

> NOTE 3: This specification refers to a container for a related set of files in a VCS as a "repository." Different VCSs might use different terms~~; for example, Visual Studio Team Services Version Control calls it a "team project".~~.

> NOTE 4: This specification refers to a fixed revision of a set of files as a "revision". Different VCSs use different terms; for example, Git calls it a "commit".

> EXAMPLE: In this example, an analysis tool has scanned files from one repository: the GitHub repository `example/browser`.

```
{                                     # A run object.
  "versionControlProvenance": [
    {                                 # A versionControlDetails object (§3.23).
      "urirepositoryUri": "https://github.com/example/browser",    # See
§3.23.3.
      "revisionId": "fd3fbac"
1a0c6554caa37144459cb97cb15429b27831476e" # See §3.23.4.
      "branch": "master"              # See §3.23.5.
    }
  ]
}
```

  

### ~~3.12.12~~3.14.14 originalUriBaseIds property

A `run` object **MAY** contain a property named `originalUriBaseIds` whose value is ~~a JSON~~an object (§3.6) each of whose property names designates a URI base id (§3.4.4~~). The value of~~) and each of whose property values is an `artifactLocation` object (§3.4) that specifies (in the manner described below) the absolute URI [RFC3986] ~~which is the value~~ of that URI base id on the machine where the SARIF producer ran.

If the `artifactLocation` object's `uri` property (§3.4.3) is a relative reference, its `uriBaseId` property (§3.4.4) **SHALL** be present. Otherwise (that is, if `uri` is an absolute URI, or if it is absent), `uriBaseId` **SHALL** be absent.

If the actual value of `uri` would have been an absolute URI, `uri` **MAY** be omitted.

NOTE 1: A SARIF producer might omit such an absolute URI, or a SARIF postprocessor might remove it, for various reasons:

- To avoid revealing sensitive information such as a user name in a URI, for example, `file:///C:/Users/Mary/code/TheProject/`.
- To produce deterministic output (see Appendix F) by avoiding path names that differ depending on the machine where the analysis tool runs.

EXAMPLE 1: In this example, the "top-level" property `PROJECTROOT` specifies a URI containing a username:

```
"originalUriBaseIds": {
 "PROJECTROOT": {
    "uri": "file:///C:/Users/Mary/code/TheProject/",
    "description": "The root directory for all project files."
 },
  "SRCROOT": {
    "uri": "src",
    "uriBaseId": "PROJECTROOT",
    "description": "The root of the source tree."
  }
}
```

A post-processor might remove `uri` to avoid revealing a username. The advantage of this approach over removing the entire `PROJECTROOT` property is that it retains the `description` property:

```
"originalUriBaseIds": {
 "PROJECTROOT": {
    "description": "The root directory for all project files."
 },
  "SRCROOT": {
    "uri": "src",
    "uriBaseId": "PROJECTROOT",
    "description": "The root of the source tree."
  }
}
```

The values of the `uriBaseId` properties in the `artifactLocation` objects in `originalUriBaseIds` **SHALL NOT** form a loop, in the sense described in the URI base id resolution procedure below.

The values of the `uri` properties in the `artifactLocation` objects in `originalUriBaseIds`:

- **SHALL** end with a single forward slash .
- **SHALL NOT** include a query or fragment component as defined in URI Generic Syntax [RFC3986].
- **SHALL NOT** include `".."` path segments.

  

NOTE 2: The rationale for these restrictions is to allow the `uriBaseId` resolution procedure described below to work by simple concatenation of the `uri` properties in `originalUriBaseIds`. The prohibition of `".."` path segments ensures that the resolution procedure works with `file` scheme URIs, without concern for the presence of symbolic links. See §3.10.2 for more information on this point.

This property allows SARIF consumers to resolve any relative references which appear in any ~~`fileLocation`~~`artifactLocation` objects ~~(§)~~elsewhere in the run, as long as the consumer runs either on the same machine as the producer, or on a machine with an identical file system layout. This is useful for individual developers who wish to run analysis tools and examine the results in a viewer. It is also useful for teams which share a convention for their file system layout.

A SARIF consumer **SHALL** use the following procedure to resolve a URI base id from the information in `originalUriBaseIds`:

NOTE 3: This procedure is part of an overall URI base id resolution procedure described in §3.4.4~~When a SARIF consumer resolves a relative reference in a SARIF file, if the user has configured the consumer to use a particular value for the URI base id, the consumer **SHALL** use the configured value. If the file does not exist in that location, then the consumer **SHALL** use the value specified in the `originalBaseIds` property, if present. If the file does not exist at that location, the consumer **MAY** use other information or heuristics to locate the file.~~

.

NOTE 4: In this procedure, we refer to the resolved URI value by the variable name `resolvedUri`.

1. Set `resolvedUri` to an empty string.

2. Fetch the `artifactLocation` object whose property name within `originalUriBaseIds` is the value of `uriBaseId`. If there is no such property, the resolution procedure fails.

3. Prepend `artifactLocation.uri` to `resolvedUri`.

4. If `artifactLocation.uri` is an absolute URI, `resolvedUri` is the final resolved URI, and the procedure succeeds.

   Otherwise:

5. If `uriBaseId` is absent, the resolution procedure fails.

   NOTE 3: This would not occur in a valid SARIF file, but the file might not be valid.

6. If the value of `uriBaseId` has already been encountered during this resolution procedure (that is, if there is a loop in the sequence of URI base ids), the resolution procedure fails.

   NOTE 4: This would not occur in a valid SARIF file, but the file might not be valid.

7. Otherwise (that is, if `uriBaseId` is present and its value has not previously been encountered during this resolution), return to Step 2.

   EXAMPLE 2: In this example, the URI base id `"SRCROOT"` on the machine where the SARIF producer ran was `"file:///C:/code/MyProject/src~~"~~/"`. The producer detected a result in a file whose location relative to that URI base id was `"lib/memory.c"`. A viewer which wished to display that file would first attempt to locate it on the local file system at `"C:\code\MyProject\src\lib\memory.c"`. If the file did not exist at that location, the viewer might prompt the user for the location.

```
{                                          # A run object.
  "originalBaseIdsoriginalUriBaseIds": {
```

  

```
    "PROJECTROOT": {
      "uri": "file:///C:/code/TheProject/"
    },
    "SRCROOT": "file:///C:/{
      "uri": " src"/",
      "uriBaseId": "PROJECTROOT"
    }
  },

  "results": [
    {                                      # A result object (§3.27)—).
      "ruleId": "CA1001",
      "locations": [
        {                                  # A location object (§3.28)).
          "physicalLocation": {            # See §3.28.3.
            "fileLocation": {                        # A
fileLocationartifactLocation": {          # An artifactLocation object (§3.4)).
              "uri": "lib/memory.c",
              "uriBaseId": "SRCROOT"
            }
          }
        }
      ]
    }
  ]
}
```

~~The rules governing the inclusion of the host name in a URI that uses the `"file"` protocol are the same as for the `fileLocation.uri` property (see §).~~

## ~~3.12.13 files property~~

### ~~3.12.13.1 General~~

## 3.14.15 artifacts property

A `run` object ~~**SHOULD**~~**MAY** contain a property named ~~`files`~~`artifacts` whose value is an array of zero or more unique (§3.7.3~~a JSON object~~) `artifact` objects (§3.24) each of ~~whose properties~~ which represents ~~a file~~an artifact relevant to the run.

~~The object specified by the `files` property~~The array **SHOULD** contain ~~properties~~elements representing at least those ~~files~~artifacts in which results were detected, but it **MAY** contain ~~properties~~elements representing all ~~files~~artifacts examined by the tool (whether or not results were detected in those ~~files~~artifacts), or any subset of those ~~files~~artifacts. It **MAY** also include other ~~files~~artifacts relevant to the run, such as attachments (§3.27.26~~, §~~).

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

EXAMPLE:

```
"files": {
—"artifacts": [
  {
    "location": {
      "uri": "file:///C:/Code/main.c": {"
    "mimeType": "text/x-},
    "sourceLanguage": "c",
    "hashes": {[
      {
          "value"sha-256": "b13ce2678a8807ba0765ab94a0ecd394f869bc81",""
```

  

```
        "algorithm": "sha-256"}
   }
    }
  }
}
```

### 3.12.13.2 Property names

The property names in the `files` object are related to the file locations specified in `fileLocation` objects (§) within the run. The syntax for the property names is:

```
files object property name = absolute property name | relative property name

absolute property name = URI

relative property name = [ uri base id prefix ], relative-ref

URI = (? an absolute URI as defined by the URI construct in RFC 3986 ?)

relative-ref = (? a relative URI as defined by the relative-ref construct in RFC 3986 ?)

uri base id prefix = "#", uri base id, "#"

uri base id = (? the value of a uriBaseId property in a fileLocation object ?)
```

If the `fileLocation.uri` property (§) contains an absolute URI, the corresponding property name in the `files` object **SHALL** be an `absolute property name` containing an absolute URI equivalent to the value of `fileLocation.uri` in the sense described in §.

> EXAMPLE 1: In this example, a `fileLocation` object in the run has a `uri` property whose value is an absolute URI. The name of the corresponding property in the `files` object matches that URI.

```
{                                     # A run object (§).
  "results": [
    {                                 # A result object (§).
      "relatedLocations": [
        {                             # A location object (§).
}
```

```
          "physicalLocation": {       # A physicalLocation object (§).
            "fileLocation": {         # A fileLocation object (§).
              "uri": "file:///C:/source/input.c"
            }
          }
        }
      ]
    }
  ],

  "files": {
    "file:///C:/source/input.c": {   # Property name matches absolute URI from
      ...                             # fileLocation object
    }
  }
}
```

If the `fileLocation.uri` property contains a relative reference, the corresponding property name in the `files` object **SHALL** be a `relative property name` whose `relative-ref` portion is a relative reference equivalent to the value of `fileLocation.uri` in the sense described in §.

> EXAMPLE 2: In this example, a `fileLocation` object in the run has a `uri` property whose value is a relative reference. The name of the corresponding property in the `files` object matches that relative reference.

```
{                                   # A run object (§).
  "results": [
    {                               # A result object (§).
      "relatedLocations": [
        {                           # A location object (§).
          "physicalLocation": {     # A physicalLocation object (§).
            "fileLocation": {       # A fileLocation object (§).
              "uri": "input.c",
              "uriBaseId": "SRCROOT"
            }
          }
        }
      ]
    }
  ],

  "files": {
    "input.c": {                     # Property name matches relative reference from
      ...                            # fileLocation object
    }
  }
}
```

If two or more properties in the `files` object correspond to `fileLocation` objects with equivalent relative reference-valued `uri` properties but different `uriBaseId` properties (§), then each of the conflicting property names **SHALL** have a `uri base id prefix`. This avoids a situation where two properties would otherwise have the same property name.

NOTE 1: Since no valid URI reference starts with a `"#"` character, there is no danger of a property name that starts with a `uri base id prefix` colliding with another property name that represents a URI reference with no prefix.

EXAMPLE 3: In this example, two `fileLocation` objects have the same relative reference-valued `uri` property but different `uriBaseId` properties. The names of the corresponding properties in the `files` object include a `uri base id prefix` to avoid a property name collision.

```
{                                   # A run object (§).
  "results": [
    {                               # A result object (§).
      "relatedLocations": [
        {                           # A location object (§).
          "physicalLocation": {     # A physicalLocation object (§).
            "fileLocation": {       # A fileLocation object (§).
              "uri": "utilities.c",
              "uriBaseId": "SRCROOT"
            }
          },
          "physicalLocation": {
            "fileLocation": {
              "uri": "utilities.c",
              "uriBaseId": "TESTSRCROOT"
            }
          }
        }
      ]
    }
  ],

  "files": {
    "#SRCROOT#utilities.c": {       # Property name includes uri base id prefix
      ...
    },
    "#TESTSRCROOT#utilities.c": {
      ...
    }
  }
}
```

  

If a `relative property name` does *not* conflict with any other property name in the `files` object, the `uri base id prefix` portion of the property name **SHOULD** be absent (see EXAMPLE 2).

> NOTE 2: This recommendation improves the readability of the SARIF log file. It is a recommendation, rather than a requirement, to accommodate SARIF producers which do not wish to include the extra logic necessary to keep track of property name collisions.

Regardless of whether the property name represents an absolute URI, a relative reference, or a relative reference with a `uri base id prefix`, the URI reference portion of the property name **SHOULD** be normalized as described in [].

> EXAMPLE 4: In this example, the `uri` property of the `fileLocation` object is not normalized, but the name of the corresponding property in the `files` object *is* normalized.

```
{                                   # A run object (§)
  "results": [
    {                               # A result object (§)
      "relatedLocations": [
        {                           # A location object (§)
          "physicalLocation": {     # A physicalLocation object (§)
            "fileLocation": {       # A fileLocation object (§)
              "uri": "FILE:///C:/source/input.c"  # scheme is not normalized
            }
          }
        }
      }
  ],

  "files": {
    "file:///C:/source/input.c": {   # Property name matches absolute URI after
      ...                            # normalization (scheme has been normalized).
    }
  }
}
```

Every pair of absolute URI-valued property names **SHALL** be distinct (that is, they **SHALL** differ after normalization) as described in §. Similarly, every pair of relative reference-valued property names which lack a `uri base id prefix` **SHALL** be distinct.

> NOTE 3: This restriction ensures that there is only one property in the `files` object that describes any given physical file.

> EXAMPLE 5: This example represents invalid SARIF because the names of two properties in the files object are not distinct; that is, they would be the same if both were normalized.

```
"files": {
  "FILE:///C:/source/input.c": {
    ...
  },
  "file:///C:/source/input.c": {  # INVALID: the property names are not distinct.
    ...
  }
}
```

### 3.12.13.3 Property values

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

In some cases, ~~a file~~an artifact might be nested within another ~~file~~artifact (for example, a compressed container), referred to as its "parent." ~~A file~~An artifact that is not nested within another ~~file~~artifact is referred to as a "top-level file". A fileartifact". An artifact that is nested within another fileartifact is referred to as a "nested file".

If the file is a nested file, then the property name **SHALL** specify a URI reference to the outermost parent, together with a fragment that describes the nesting of the file within its parent or parents. The fragment **SHALL** begin with a forward slash character ("/"), to emphasize that it represents the complete path to the nested file within its container.

EXAMPLE 1: Valid: The fragment begins with a forward slash:

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

EXAMPLE 2: Invalid: The fragment does not begin with a forward slash:

```
"files": {
  "file:///C:/bin/archive.zip#images/grape.jpg": { # INVALID
    ...
  }
}
```

If the file is nested more than one level deep in the outermost parent, the fragments artifact". Within the `artifacts` array, an `artifact` object representing a nested artifact is linked to its parent *via* its `parentIndex` property (§3.24.3each 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 §). For an example, see §3.24.3.

NOTE: 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 `fileLocation` property (§), or in the `offset` (§) and `length` (§) properties, of each `file` object.

EXAMPLE 3: 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 [], those character **SHALL** be percent-encoded as specified in [].

EXAMPLE 4: 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 5: 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 §.

```
"files": {
  "file:///C:/Code/app.zip": {
    "mimeType": "application/zip",
  },
  "file:///C:/Code/app.zip#/docs/intro.docx": {
    "fileLocation": {
      "uri": "/docs/intro.docx",
    },
```

  

```
    "mimeType": "application/vnd.openxmlformats-officedocument.wordprocessingml.document",
    "parentKey": "file:///C:/Code/app.zip"    # See §
  },
  "file:///C:/Code/app.zip#/docs/intro.docx/Flash1": {
    "offset": 17522,
    "length": 4050,
```

If a nested artifact appears in the `artifacts` array, then the `artifacts` array **SHALL** also contain elements describing each of its parents, up to and including the top-level artifact.

## 3.14.16 specialLocations property

```
A run object MAY contain a property named specialLocations whose value is a specialLocations object (§3.25    "mimeType": "application/x-shockwave-flash",
    "parentKey": "file:///C:/Code/app.zip#/docs/intro.docx"
  }
}
```

) that defines locations of special significance to SARIF consumers.

## 3.12.143.14.17 logicalLocations property

Depending on the circumstances, aA `run` object either **MAY** or **SHOULD** contain a property named `logicalLocations` whose value is a JSON objectan array of zero or more unique (§3.7.3) `logicalLocation` objects (§3.33) each of whose propertieswhich represents a logical location relevant to one or more results detected during the run.

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

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

With one rare exception described in §, each property name in the `logicalLocations` object **SHALL** be the fully qualified name of the logical location. See § for examples. The property names **SHALL** follow the naming rules for fully qualified logical names described in §.

Each property value in the `logicalLocations` object **SHALL** be a `logicalLocation` object (§).

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". Within the `logicalLocations` array, a `logicalLocation` object representing a nested logical location is linked to its parent *via* its `parentIndex` property (§3.33.8).

If a nested logical location appears in the `logicalLocations` objectarray, then the `logicalLocations` objectarray **SHALL** also contain propertieselements 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` objectarray contains not only a propertyan element describing the class, but also propertieselements describing its containing namespaces.

```
"logicalLocations": {[
  {
    "name": "classC",
    "fullyQualifiedName": "namespaceA::namespaceB::classC": {",
    "name": "classC",
    "kind": "type",
```

  

```
    "parentKeyparentIndex": 1
  },
  {
    "name": "namespaceB",
    "fullyQualifiedName": "namespaceA::namespaceB"",
  },
"namespaceA::namespaceB": {
    "name": "namespaceB",
    "kind": "namespace"
    "parentKey": "namespaceA"parentIndex": 2
  },
  "namespaceA": {{
    "namefullyQualifiedName": "namespaceA",
    "kind": "namespace"
  }
}
]
```

NOTE: The detailed information in `logicalLocations` is useful, even though much of it is captured in ~~`location.fullyQualifiedLogicalName`~~ ~~(§~~`logicalLocation.fullyQualifiedName` (§3.33.5), because it allows results management systems and other SARIF consumers 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`~~`fullyQualifiedName` string.

### 3.14.18 addresses property

A `run` object **MAY** contain a property named `addresses` whose value is an array of zero or more unique (§3.7.3) `address` objects (§3.32) representing addresses that appear in `physicalLocation` objects (§3.29) within `theRun`.

In some cases, an address might be nested within another address (for example, an offset within a table within a section). An address that is nested within another address is referred to as a "nested address". Within the `addresses` array, an `address` object representing a nested address is linked to its parent *via* its `parentIndex` property (§3.32.13).

If a nested address appears in the `addresses` array, then `addresses` **SHALL** also contain elements describing each of its parents, up to and including the top-level address.

### 3.14.19 threadFlowLocations property

A `run` object **MAY** contain a property named `threadFlowLocations` whose value is an array of zero or more unique (§3.7.3) `threadFlowLocation` objects (§0) representing locations that appear in `threadFlow` objects (§3.37) within `theRun`.

The `threadFlowLocations` array may contain all or any subset of the `threadFlowLocation` objects in the run.

NOTE: Defining `threadFlowLocation` objects within `run.threadFlowLocations` can reduce the size of the log file if certain locations occur frequently, either within a single thread flow (for example, if the thread flow represents a loop) or across thread flows (for example, if all thread flows start at the program entry point and share their first few locations).

### ~~3.12.15~~3.14.20 graphs property

A `run` object **MAY** contain a property named `graphs` whose value is an array of ~~one~~zero or more unique (§3.7.3) `graph` objects (§3.39~~) each of which~~). A `graph` object represents a directed graph~~. A directed~~

  

~~graph is~~: a network of nodes and directed edges that describes some aspect of the structure of the code (for example, a call graph).

A `graph` object defined at the `run` level **MAY** be referenced by a `graphTraversal` object (§3.42) defined in the `graphTraversals` property (§3.27.20) of any `result` object (§3.27) in ~~the `run`~~`theRun`.

### 3.14.21 webRequests property

A `run` object **MAY** contain a property named `webRequests` whose value is an array of zero or more unique (§3.7.3) `webRequest` objects (§3.46) representing HTTP requests that appear in `result` objects (§3.27) within `theRun`.

NOTE: This property is primarily useful to web analysis tools.

### 3.14.22 webResponses property

A `run` object **MAY** contain a property named `webResponses` whose value is an array of zero or more unique (§3.7.3) `webResponse` objects (§3.47) representing HTTP responses that appear in `result` objects (§3.27) within `theRun`.

NOTE: This property is primarily useful to web analysis tools.

### ~~3.12.16~~3.14.23 results property

~~A~~Depending on the circumstances, a `run` object either **SHALL** or **MAY** contain a property named `results` whose value ~~is~~, again depending on circumstances, is either `null` or an array of zero or more `result` objects (§3.27)~~,~~) each of which represents a single result detected in the course of the run.

NOTE: The `results` array is not defined to contain unique (§3.7.3) elements because some tools report a line number but not a column number for a result's location. Such a tool might report the same result twice on the same line, in some cases producing multiple identical `result` objects.

~~The~~If the tool failed to start, and if the engineering system responsible for running the tool synthesized a SARIF file to record the failure, then `results` ~~array~~ **MAY** be present. If it is present, its value **SHALL** be `null`. See §3.20.13~~empty if the tool~~, `invocation` ~~that produced the `run` object~~`.processStartFailureMessage`, for more about this scenario.

If the tool started but failed to begin its analysis (for example, because its command line was invalid), then again `results` **MAY** be present, and if present **SHALL** be `null`.

In all other circumstances, `results` **SHALL** be present and **SHALL** contain all results detected by the tool. If the tool did not detect any results, `results` **SHALL** be an empty array.

~~resources~~If `results` is absent, it **SHALL** default to `null`.

### ~~3.12.17~~3.14.24 defaultEncoding property

~~A `run` object **MAY** contain a property named `resources` whose value is a `resources` object (§). A `resources` object represents items that can be localized, such as resource strings and rule metadata.~~

### ~~3.12.18 defaultFileEncoding~~

A `run` object **MAY** contain a property named ~~`defaultFileEncoding`~~`defaultEncoding` whose value is a case-sensitive string that provides a default for the `encoding` property (§3.24.9) of any ~~`file`~~`artifact` object (§3.24) in ~~`run.files`~~`theRun.artifacts` (§3.14.15) that refers to a text ~~file~~artifact. The string **SHALL** be one of the character set names ~~specified in~~defined by IANA [IANA-ENC]. ~~The property value **SHALL** be case-insensitive.~~

  

If this property is absent, it **SHALL** be interpreted as meaning that there is no default file encoding. In that case, the encoding of any ~~file~~artifact object that does not contain an `encoding` property **SHALL** be taken to be unknown.

For an example, see §3.24.9.

### 3.14.25 defaultSourceLanguage property

A `run` object **MAY** contain a property named `defaultSourceLanguage` whose value is a hierarchical string (§3.5.4) that provides a default value for the `sourceLanguage` property (§3.24.10) of any `artifact` object (§3.24) in `theRun.artifacts` (§3.14.15) which refers to a text artifact that contains source code.

If `defaultSourceLanguage` is present, its value **SHOULD** conform to the conventions defined in §3.24.10.2.

If `defaultSourceLanguage` is absent, it **SHALL** be taken to mean that there is no default source language. In that case, the source language of any `artifact` object that does not contain a `sourceLanguage` property **SHALL** be taken to be unknown. In that case, a SARIF viewer **MAY** use any method or heuristic to determine the source language of each file, for example by examining the file's file name extension or MIME type, or by prompting the user.

### 3.14.26 newlineSequences property

A `run` object **MAY** contain a property named `newlineSequences` whose value is an array of one or more unique (§3.7.3) strings each of which specifies a character sequence that the tool treated as a line break during this run.

If this property is absent, it **SHALL** default to the array `[ "\r\n", "\n" ]`.

The order of the elements in the array is significant. It **SHALL** mean that at potential line breaks, the tool "greedily" attempted to match each element of the array in order.

> EXAMPLE 1: If `newlineSequences` has the value `[ "\r\n", "\r", "\n" ]`, the character sequence `"\r\n"` counts as one line break, not two.

> NOTE: This property is useful for SARIF consumers that are sensitive to the value of the line number properties `startLine` (§3.30.5) and `endLine` (§3.30.7) in `region` objects (§3.30). It ensures that the consumer counts lines in the same way as the producer. A SARIF viewer might use this property when highlighting a region to ensure that it highlights the correct lines. More critically, a tool that applies fixes (see §3.55), especially one that applies them automatically, can use this property to ensure that it inserts and removes content on the correct lines.

> EXAMPLE 2: In this example, the SARIF producer accepts the Unicode characters NEXT LINE (U+0085) and LINE SEPARATOR (U+2028) as line separators in addition to the usual values.

```
{          # A run object (§3.14).
  ...
  "newlineSequences": [ "\r\n", "\n", "\u0085", "\u2028" ],
  ...
}
```

### ~~3.12.19~~3.14.27 columnKind property

If a SARIF producer processes text artifacts and `theRun.results` (§3.14.23~~files~~) is non-empty, the `run` object **SHALL** contain a property named `columnKind` whose value is a string that specifies the unit in which the analysis tool measures columns. If a SARIF producer processes text artifacts and `theRun.results` is empty, `columnKind` **MAY** be present.

`columnKind` **SHALL** have one of the following values, with the specified meanings:

  

- "utf16CodeUnits": Each UTF-16 code unit is considered to occupy one column. This means that a surrogate pair is considered to occupy two columns.
- "unicodeCodePoints": Each Unicode code point (abstract character) is considered to occupy one column. This means that even a character that is represented in UTF-16 by a surrogate pair is considered to occupy one column.

If the SARIF producer does not process text filesartifacts, columnKind **SHALL** be absent.

If a SARIF consumer uses a column measurement unit other than that specified by columnKind, and if the consumer is required to interact with the fileartifact's contents (for example, by displaying the fileartifact in an editor and highlighting a region), the consumer **SHALL** recompute column numbers in its (the consumer's) native measurement unit.

### 3.12.203.14.28 richMessageMimeTyperedactionsToken property

If the value of any redactable property (§3.5.2A run object **MAY** ) in theRun has been redacted, theRun **SHALL** contain a property named richMessageMimeTyperedactionTokens whose value is a string that specifies the MIME type [] of all rich text message properties (§) in the run. If this property is absent, it **SHALL** default to "text/markdown;variant=GFM". [] defines the "text/markdown" media type, and [] registers "GFM" as the valuean array of zero or more unique (§3.7.3the variant parameter ) strings any of which specifies GitHub-Flavored Markdown [].

For a discussion of the security implications of expressing rich text messages in GFM, see §.

### 3.12.21 redactionToken property

If the value of any redaction-aware property (§) in the run has been redacted, the run object **SHALL** contain a property named redactionToken whose value is the string can be used to replace the redacted text. If no text in the runtheRun has been redacted, the redactionToken propertyredactionTokens **SHALL** be absent.

The value of redactionTokenIf redactionTokens contains a single element, that element **SHOULD** be the string "[REDACTED] ". ]"; if it contains more than one, each additional element **SHOULD** be of the form "[REDACTED-*n*]" where *n* is a positive integer.

> NOTE 1: The rationale for recommending the alternate form only for the second and subsequent tokens is that a tool might create one token and only later discover that additional tokens are required. With this recommendation, the tool does not have to rename the token it has already created.

> NOTE 2: Redaction tokens have no special meaning in properties not specified as "redactable."

If for any reason a different value isvalues are used, itthey **MAY** be any readily identifiable stringstrings. An example of a situation where a SARIF producer might choose a different redaction token is if the string "[REDACTED]" occurs in the value of any redaction-awarea redactable property in the runtheRun.

> EXAMPLE 1: In this example, the leading portion of a full path name has been redacted from the redaction-awareredactable property invocation.commandLine to avoid revealing information about the machinemachine's directory layout.

```
{                          # A run object (§3.14.).
  "redactionToken":redactionTokens": [
    "[REDACTED]",]"
  ],

  "invocation": {
    "commandLine": "SourceScanner --input [REDACTED]/src/ui"
  }
  ...
}
```

  

## 3.15 externalPropertyFileReferences object

### 3.15.1 General

#### 3.12.22 An `externalPropertyFileReferences` object contains information that enables a SARIF consumer to locate the external property files (see §3.15.2properties property

A ) that contain the values of all externalized properties associated with `theRun`.

### 3.15.2 Rationale

In some engineering environments, a single tool run object **MAY** contain a might analyze hundreds of thousands of files and produce millions of results. This causes problems for both producers and consumers of such large SARIF log files:

- The log file might be too large for a consumer to hold in memory and might take several minutes to read.
- During production, some information (such as the complete set of artifacts that were analyzed, the complete set of rules that were violated, or the end time of the run) cannot be known until the run is complete. Therefore, it is likely to be serialized at the end of the log file. However, consumers might need to access some of that information before reading the entire file. For example, a SARIF viewer might need to display rule metadata along with each result it displays, or to display the start and end times of a set of tool runs.

To mitigate these problems, SARIF allows certain properties of a `run` object and its sub-objects to be stored in separate files. We refer to these files as "external property files", and we refer to the file containing the `run` object itself as the "root file". We refer to a property that can be stored in an external property file as an "externalizable property." We refer to a property that *has* been stored in an external property file as an "externalized property."

The format of an external property file is described in §4.

A SARIF consumer **SHALL** treat the value of an object-valued property stored in an external property file exactly as if it had appeared inline in the root file as the value of the corresponding property.

A SARIF consumer **SHALL** treat the value of an array-valued property stored in an external property file exactly as if its elements had appeared inline in the root file, appended to the existing value, if any, of that property.

> NOTE: This allows a SARIF producer to begin writing the elements of an array-valued property to the root file, and then, if the file grows too large, to "spill" the additional elements into one or more external property files.

### 3.15.3 Properties

The following table lists all the externalizable properties together with their corresponding property names in the `externalPropertyFileReferences` object:

| Externalizable property | Property name | Type |
|---|---|---|
| `run.addresses` | `addresses` | array |
| `run.artifacts` | `artifacts` | array |
| `run.conversion` | `conversion` | object |
| `run.graphs` | `graphs` | array |
| `run.invocations` | `invocations` | array |

  

| | | |
|---|---|---|
| `run.logicalLocations` | `logicalLocations` | array |
| `run.policies` | `policies` | array |
| `run.properties` | `externalizedProperties` | object |
| `run.webRequests` | `webRequests` | array |
| `run.webResponses` | `webResponses` | array |
| `run.results` | `results` | array |
| `run.taxonomies` | `taxonomies` | array |
| `run.threadFlowLocations` | `threadFlowLocations` | array |
| `run.translations` | `translations` | array |
| `run.tool.driver` | `driver` | object |
| `run.tool.extensions` | `extensions` | array |

NOTE 1: `run.properties` is externalized under the property name `externalizedProperties` to allow this object to have a property bag named `properties`, consistent with all other objects in this specification.

NOTE 2: Note that `run.conversion.tool.driver` and `run.conversion.tool.extensions` are not separately externalizable. Rather, the `run.conversion` property as a whole is externalizable.

Every externalizable property whose type is shown in the table as "object" **SHALL**, if externalized, be stored in a single external property file. In that case, the value of the corresponding property in `externalPropertyFileReferences` **SHALL** be an `externalPropertyFileReference` object (§3.16) specifying the location of the external property file.

Every externalizable property whose type is shown in the table as "array" **SHALL**, if externalized, be stored in one or more external property files. In that case, the value of the corresponding property in `externalPropertyFileReferences` **SHALL** be an array of zero or more `externalPropertyFileReference` objects specifying the locations of those external property files.

EXAMPLE 1: In this example, `run.conversion` is stored in the file `C:\logs\scantool.conversion.sarif-external-properties` and `run.results` is divided into the files `C:\logs\scantools.results-1.sarif-external-properties` and `C:\logs\scantools.results-2.sarif-external-properties`.

```
{                               # A run object.
  "originalUriBaseIds": {       # See §3.14.14.
    "LOGSDIR": {
      "uri": "file:///C:/logs/"
    }
  },
  "externalPropertyFileReferences": {
    "conversion": {             # An externalPropertyFileReference object (§3.16).
      "location": {             # See §3.16.3.
        "uri": "scantool.conversion.sarif-external-properties",
        "uriBaseId": "LOGSDIR"
      },
      "guid": "11111111-1111-1111-1111-111111111111" # See §3.16.4.
    },
    "results": [
      {
        "location": {
```

  

```
          "uri": "scantool.results-1.sarif-external-properties",
          "uriBaseId": "LOGSDIR"
        },
        "guid": "22222222-2222-2222-2222-222222222222",
        "itemCount": 10000
      },
      {
        "location": {
          "uri": "scantool.results-2.sarif-external-properties",
          "uriBaseId": "LOGSDIR"
        },
        "guid": "33333333333-3333-3333-3333-333333333333",
        "itemCount": 4277
      }
    ]
  }
  ...
}
```

With one exception described below, if a property appears inline in the root file, its name **SHALL NOT** appear as one of the property names in `externalPropertyFileReferences`. Since an external property file can contain multiple externalized properties, `externalPropertyFileReference` objects belonging to distinct properties **MAY** denote the same external property file. However, if an array-valued externalizable property is divided among multiple external property files, the `externalPropertyFileReference` objects belonging to that property **SHALL** denote distinct external property files.

EXAMPLE 2: In this example, `theRun.conversion` and `theRun.properties` are stored in the same external property file.

```
{                              # A run object (§3.14).
  "originalUriBaseIds": {     # See §3.14.14.
    "LOGSDIR": {
      "uri": "file:///C:/logs/"
    }
  },
  "externalPropertyFileReferences": {
    "conversion": {      # An externalPropertyFileReference object (see §3.16).
      "location": {            # See §3.16.3.
        "uri": "scantool.sarif-external-properties",
        "uriBaseId": "LOGSDIR",
        "index": 0
      },
      "guid": "11111111-1111-1111-1111-111111111111" # See §3.16.4.
    },
    "externalizedProperties": {
      "location": {
        "uri": "scantool.sarif-external-properties",
        "uriBaseId": "LOGSDIR",
        "index": 0
      },
      "guid": "11111111-1111-1111-1111-111111111111"
    }
  }
  ...
}
```

EXAMPLE 3: This example represents invalid SARIF because both elements of the array belonging to the `results` property denote the same external property file.

```
{                              # A run object (§3.14).
  "originalUriBaseIds": {     # See §3.14.14.
    "LOGSDIR": {
```

```
      "uri": "file:///C:/logs/"
    }
  },
  "externalPropertyFileReferences": {
    "results": [
      {                  # An externalPropertyFileReference object (see §3.16).
        "location": {
          "uri": "scantool.results.sarif-external-properties",
          "uriBaseId": "LOGSDIR",
          "index": 0
        },
        "guid": "22222222-2222-2222-2222-222222222222"
      },
      {               # INVALID: The two external property files are the same.
        "location": {
          "uri": "scantool.results.sarif-external-properties",
          "uriBaseId": "LOGSDIR",
          "index": 0
        },
        "guid": "22222222-2222-2222-2222-222222222222"
      }
    ]
  }
  ...
}
```

The exception is that if `run.tool.driver` is externalized, it **SHALL** still occur inline in the root file. The inline `driver` property **SHOULD** contain only properties that identify the tool, such as `name` (§3.19.8) and `semanticVersion` (§3.19.12); it **SHOULD NOT** contain properties such as `globalMessageStrings` (§3.19.22), `rules` (§3.19.23), `notifications` (§3.19.24), and `taxa` (§3.19.25), which take up a large among of space.

> NOTE 3: This makes it possible to identify the tool that produced the log file without locating and opening the external property file, while still getting the benefit of externalizing those properties that take up a large amount of space.

## 3.16 externalPropertyFileReference object

### 3.16.1 General

An `externalPropertyFileReference` object contains information that enables a SARIF consumer to locate the external property file (see §3.15.2) that contains the value of an externalized property associated with `theRun`.

### 3.16.2 Constraints

At least one of the `location` property (§3.16.3) or the `guid` property (§3.16.4) **SHALL** be present. If both are present, they **SHALL** identify the same set of externalized properties (possibly located inline; see §3.13.5).

> NOTE: This constraint ensures that it is possible to locate the externalized properties.

### 3.16.3 location property

Depending on the circumstances, an `externalPropertyFileReference` object either **SHALL** or **MAY** contain a property named `location` whose value is an `artifactLocation` object (§3.4) that specifies the location of the external property file.

If the externalized properties are persisted in a separate file, `location` **SHALL** be present. In that case, if the `artifactLocation` object's `uri` property (§3.4.3) specifies a relative reference and its uriBaseId property (§3.4.4) is absent, then uri **SHALL** be interpreted relative to the location of the root file.

Otherwise (that is, if the externalized properties are persisted as an element of theSarifLog.inlineExternalProperties (§3.13.5)), then location **MAY** be present. If location is present, its uri property **SHALL** resolve to an absolute URI using the sarif scheme (§3.10.3). If location is absent, then a SARIF consumer that needs to locate the externalized properties **SHALL** do so using the guid property (§3.16.4).

### 3.16.4 guid property

Depending on the circumstances, an externalPropertyFileReference object either **SHALL** or **MAY** contain a property named guid whose value is a GUID-valued string (§3.5.3) which provides a unique, stable identifier for the external property file.

If the externalized properties are persisted in an element of theSarifLog.inlineExternalProperties (§3.13.5) and location (§3.16.3) is absent, then guid **SHALL** be present.

Otherwise (that is, if the extermalized properties are persisted in a separate file, in which case location is required, or if the externalized properties are persisted in an element of theSarifLog.inlineExternalProperties but location is present), guid **MAY** be present.

> NOTE: The rationale for these constraints is to ensure that there is enough information to locate the external properties. If the properties are in an external file, then location is necessary but guid can still be present; if the properties are inline, either location or guid suffices but both can be present.

If guid is present, it **SHALL** equal the guid property (§4.3.4) of the externalProperties object (§4.3~~value is a property bag (§). This allows tools to include information about the run that is not explicitly~~) identified by guid and/or location.

### 3.16.5 itemCount property

If an externalPropertyFileReference object specifies an external property file that contains all or a portion of an array-valued property, it **MAY** contain a property named itemCount whose value is a non-negative integer that specifies the number of items in the externalized property array in that file. If the externalPropertyFileReference object specifies an external property file that contains an object-valued property, itemCount **SHALL** be absent.

If itemCount is absent, it **SHALL** default to -1, which indicates that the value is unknown (not set).

> NOTE: This information is useful to a SARIF consumer that needs to locate the item at a specified ~~in the SARIF format~~array index in an externalized array-valued property. Without this information, the consumer would have to open in turn each external property file belonging to that property, counting the number of array elements in each, until it reached the file containing the desired element.

> EXAMPLE: In EXAMPLE 1 in §3.15.3~~tool~~, the array-valued property results is divided into two files, the first containing 10,000 elements and the second containing 4,277 elements. A SARIF consumer that needs to access element 12,000 knows immediately that it is contained in the second file, at index 2,000.

## ~~3.13~~3.17 runAutomationDetails object

### ~~3.13.1~~3.17.1 General

A ~~tool~~runAutomationDetails object contains information that specifies theRun's identity and role within an engineering system.

  

EXAMPLE: In this example, a run contains the results from one nightly execution of a single security tool over a specified set of binaries. `theRun.automationDetails` describes the run. Its `id` and `guid` properties both identify the run; the former in human-readable form, the latter in a form that might be more useful in an engineering system's database. Its `correlationGuid` property specifies the set of runs identified by *all but the last component* of `id`'s hierarchical string; that is, it identifies the set of runs `"Nightly CredScan run for sarif-sdk/master/x86/debug"`.

The run in this example is part of an aggregate of runs which together comprise the nightly execution of the engineering system's full suite of security tools. `theRun.runAggregates[0]` describes that aggregate. Its `id` and `guid` properties both identify the aggregate. Its `correlationGuid` property specifies the collection of such aggregates identified by *all but the last component* of `id`'s hierarchical string; that is, it identifies the collection of aggregates `"Nightly security tools run for sarif-sdk/master/x86/debug"`.

```
{                               # A run object (§3.14).
  "automationDetails": {        # See §3.14.3.
    "description": {
      "text": "This is the {0} nightly run of the Credential Scanner tool on
             all product binaries in the '{1}' branch of the '{2}' repo. The
             scanned binaries are architecture '{3}' and build type '{4}'.",
      "arguments": [
        "October 10, 2018",
        "master",
        "sarif-sdk",
        "x86",
        "debug"
      ]
    },
    "id": "Nightly CredScan run for sarif-sdk/master/x86/debug/2018-10-05",
    "guid": "11111111-...",
    "correlationGuid": "22222222-...."
  },
  "runAggregates": [            # See §3.14.4.
    {
      "id":
        "Nightly security tools run for sarif-sdk/master/x86/debug/2018-10-05"
      "guid": "33333333-...",
      "correlationGuid": "44444444-...."
    }
  ]
}
```

## 3.17.2 description property

A `runAutomationDetails` object **MAY** contain a property named `description` whose value is a `message` object (§3.11) that describes the role played within the engineering system by `theRun`.

## 3.17.3 id property

A `runAutomationDetails` object **MAY** contain a property named `id` whose value is a hierarchical string (§3.5.4~~describing~~) that uniquely identifies `theRun` within the engineering system.

A result management system or other components of the engineering system **MAY** use `run.automationDetails.id` to associate the information in the log with additional information not provided by the analysis tool that produced it.

An engineering system **MAY** define any number of components and interpret them in any way desired.

NOTE: The intent is to use the components of `id` to group results from similar runs, such as "all nightly Credential Scanner runs." A SARIF viewer might display a set of runs in a tree view, grouped by the components of `id`.

EXAMPLE 1: A run whose `id` is `"My Nightly Run/Debug/x64/2018-10-10"` belongs to the category `"My Nightly Run/Debug/x64"`. Presumably, this is the run from October 10, 2018.

The trailing component of `id` **MAY** be empty; note that the grammar for a hierarchical identifier (§3.5.4.1) permits any component to be empty. This **SHALL** be taken to signify that the run belongs to the specified category, but that the run itself has no unique identifier.

EXAMPLE 2: A run whose `id` is `"My Nightly Run/Debug/x64/"` belongs to the category `"My Nightly Run/Debug/x64"` but is not distinguished from other runs in that category.

`id` **MAY** consist of a single component. This **SHALL** be taken to specify a unique identifier for the run, withough specifying any category that the run belongs to.

EXAMPLE 3: A run whose `id` is `"My Nightly Run Debug x64 2018-10-10"` has a unique identifier but cannot be inferred to belong to any category.

### 3.17.4 guid property

A `runAutomationDetails` object **MAY** contain a property named `guid` whose value is a GUID-valued string (§3.5.3) that provides a unique, stable identifier for `theRun`.

A result management system or other components of the engineering system **MAY** use `run.automationDetails.guid` to associate the information in the log with additional information not provided by the analysis tool that produced it.

### 3.17.5 correlationGuid property

A `runAutomationDetails` object **MAY** contain a property named `correlationGuid` whose value is a GUID-valued string (§3.5.3) which is shared by all such runs of the same type, and differs between any two runs of different types.

If `id` (§3.17.3) is present, `correlationGuid` **SHALL** identify the category of runs specified by all but the last hierarchical component (which **MAY** be empty according to the grammar (§3.5.4.1) for hierarchical strings) of `id`.

NOTE: Consider an engineering system that allows engineers to define "build definitions", and that assigns a GUID to each build definition. In such a system, the build definition's GUID could serve as `run.automationDetails.correlationGuid`. It would be the same for all runs produced by the same build definition, and different between any two runs produced by different build definitions.

## 3.18 tool object

### 3.18.1 General

A `tool` object describes the analysis tool or converter that was run. The `tool` object in `run.tool` (§3.14.6) describes an analysis tool; the `tool` object in `run.conversion.tool` (§3.14.12, §3.22.2) describes a converter.

A tool consists of one or more "tool components," each of which consists of one or more files. We refer to the component that contains the tool's primary executable file as the "driver." It controls the tool's execution and typically defines a set of analysis rules. We refer to all other tool components as "extensions." Extensions can include:

- Libraries of additional rules, which we refer to as "plugins."

  

- Files that affect the behavior of the tool, which we refer to as "configuration files."

  NOTE: Configuration files that affect the analysis output are of particular interest in compliance scenarios, where, for example, it is necessary to demonstrate that a particular set of rules has been evaluated.

Each tool component is represented by a `toolComponent` object (§3.19).

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:

```
{                                      # A tool object.
  "driver": {                    # See §3.18.2{
.
    "name": "CodeScanner",                                              # see §
    "fullName": "CodeScanner 1.1, Developer Preview (en-US)",           # see §
    "semanticVersion": "1.1.2-beta.12",                                 # see §
    "version": "1.1.2b12,                                               # see §",
    ...
  },
  "extensions": [                # See §3.18.3 "fileVersion": "1.1.1502.2"
# see §
}
.
    {
      "name": "CodeScanner Security Rules",
      "version": "3.1",
      ...
    }
  ]
}
```

## ~~3.13.2~~3.18.2 driver property

A `tool` object **SHALL** contain a property named ~~`name`~~`driver` whose value is a `toolComponent` object (§3.19~~string~~) that describes the component containing the ~~name of the tool that produced the log~~ tool's primary executable file.

~~EXAMPLE: "CodeScanner"~~

## ~~3.13.3~~3.18.3 ~~fullName~~extensions property

~~A~~If the tool used any extensions during the run, the `tool` object ~~**MAY**~~**SHOULD** contain a property named ~~`fullName`~~`extensions` whose value is an array of one or more unique (§3.7.3) `toolComponent` objects (§3.19) that describe those extensions. If the tool did not use any extensions during the run, then `extensions` **SHALL** either be absent or an empty array.

# 3.19 toolComponent object

## 3.19.1 General

A `toolComponent` object represents one of the components which comprise an analysis tool or a converter, either its driver or one of its extensions. For more information, see §3.18.1.

SARIF also uses `toolComponent` objects to represent other components that participate in the analysis, including:

- Taxonomies (§3.19.3)
- Translations (§3.19.4)

- Policies (§3.19.5)

  NOTE: SARIF makes this design choice because `toolComponent` objects contain properties that are useful in all of these other types of components: properties that represent the component's identity, localizable properties (§3.5.1) that label the component and describe its purpose, and properties that define rules and similar items that participate in the analysis. Not every property is useful in every component type; for example, `translationMetadata` (§3.19.27) is useful only in `toolComponent` objects that represent translations.

### 3.19.2 Constraints

At least one of `version` (§3.19.13) and `semanticVersion` (§3.19.12) **SHOULD** be present.

### 3.19.3 Taxonomies

A taxonomy is a classification of results into a set of categories. Some taxonomies are defined publicly, without reference to any particular tool; we refer to these as "standard taxonomies." An example is the Common Weakness Enumeration [CWE™]. A tool can also define its own classification (in addition to the classification implied by its rule definitions); we refer to this as a "custom taxonomy." We refer to a category within a taxonomy as a "taxon" (*pl.* "taxa").

A taxonomy is represented by a `toolComponent` object. Its taxa are stored in the `taxa` property (§3.19.25).

A taxon is represented by a `reportingDescriptor` object (§3.49); hence `toolComponent.taxa` is an array of `reportingDescriptor` objects. This is the same object that represents rules and notifications, so a taxon can specify identity properties such as `id` (§3.49.3) and `guid` (§3.49.5), localizable (§3.5.1) descriptive properties such as `name` (§3.49.7) and `fullDescription` (§3.49.10), and configuration properties in `defaultConfiguration` (§3.49.14).

Standard taxonomies **SHALL** be stored in the `run.taxonomies` array (§3.14.8). Every `toolComponent` object in this array **SHALL** contain a `taxa` property (§3.19.25), and **SHALL NOT** contain `rules` (§3.19.23) or `notifications` (§3.19.24) properties.

A custom taxonomy is represented by providing a `toolComponent` object in `tool.driver` (§3.18.2) or `tool.extensions` (§3.18.3) with a `taxa` property. Such a `toolComponent` object **MAY** still contain `rules` and/or `notifications` as usual.

> EXAMPLE: In this example, the tool driver supports the CWE™ taxonomy, and also supports a custom taxonomy that it defines. Any result that violates the driver's rule `"CA2101"` falls into the `"MemoryManagement"` taxon of its custom taxonomy, as shown by the `"superset"` relationship from the `"MemoryManagement"` taxon to the rule (which is interpreted as "The `MemoryManagement` taxon is a superset of rule `CA2101`"). For more information on relationships, see §3.49.15 and §3.53.

```
{                                   # A run object (§3.14).
  "tool": {                         # See §3.14.6.
    "driver": {                     # See §3.18.2.
      "name": "CodeScanner",
      "semanticVersion": "3.3",     # See §3.19.12.
      "guid": "11111111-1111-1111-1111-111111111111",
      ...
      "rules": [
        {
          "id": "CA2101",
          "shortDescription": {
            "text": "Failed to release dynamic memory."
          },
          "relationships": [        # See §3.49.15.
            {               # A reportingDescriptorRelationship object (§3.53).
```

  

```
              "target": {          # See §3.53.2
                "id": "MemoryManagement",
                "guid": "66666666-6666-6666-6666-666666666666",
                "toolComponent": {
                  "name": "CodeScanner",
                  "guid": "11111111-1111-1111-1111-111111111111"
                }
              },
              "kinds": [           # See §3.53.3.
                "superset"
              ]
            }
          ]
        },
        ...
      ],
      "taxa": [
        {
          "id": "MemoryManagement",
          "guid": "66666666-6666-6666-6666-666666666666",
          "shortDescription": {
            "text": "Improper usage of dynamic memory."
          }
        },
        {
          "id": "Cryptography",
          "guid": "77777777-7777-7777-7777-777777777777",
          "shortDescription": {
            "text": "Insecure use of cryptography."
          }
        }
      ],
      "supportedTaxonomies": [
        {
          "name": "CodeScanner",
          "guid": "11111111-1111-1111-1111-111111111111"
        },
        {
          "name": "CWE",
          "index": 1,
          "guid": "33333333-0000-0000-0000-000000000000"
        }
      ]
    }
  },

  "taxonomies": [
    {
      "name": "CWE",
      "version": "3.2",
      "releaseDateUtc": "2019-01-03",
      "guid": "33333333-0000-0000-0000-000000000000",
      "informationUri": "https://cwe.mitre.org/data/published/cwe_v3.2.pdf/",
      "downloadUri": "https://cwe.mitre.org/data/xml/cwec_v3.2.xml.zip",
      "organization": "MITRE",
      "shortDescription": {
        "text": "The MITRE Common Weakness Enumeration"
      },
      "contents": [
        "localizedData",
        "nonLocalizedData"
      ],
      "isComprehensive": true,
      "minimumRequiredLocalizedDataSemanticVersion": "3.2",
```

  

```
      "taxa": [
        {
          "id": "327",
          "guid": "33333333-0000-0000-0000-111111111111",
          "name": "BrokenOrRiskyCryptographicAlgorithm",
          "shortDescription": {
            "text": "Use of a Broken or Risky Cryptographic Algorithm."
          },
          "defaultConfiguration": {
            "level": "warning"
          }
        },
        {
          "id": "924",
          "guid": "33333333-0000-0000-0000-222222222222",
          "name": "TransmittedMessageIntegrity",
          "shortDescription": {
            "text": "Improper Enforcement of Message Integrity ..."
          },
          "defaultConfiguration": {
            "level": "warning"
          }
        },
        ...
      ]
    }
  ],

  ...
}
```

### 3.19.4 Translations

A translation is the rendering of a `toolComponent` object's localizable strings (§3.5.1) into another language.

A translation is itself represented by a `toolComponent` object whose localizable properties are the translated versions of the corresponding properties in the component being translated. A translation specifies the tool component to which it applies by way of its `associatedComponent` property (§3.19.33).

Translations **SHALL** be stored in the `run.translations` array (§3.14.9).

A translation **SHALL** specify the component that it translates by way of its `associatedComponent` property (§3.19.33). `associatedComponent` **SHALL NOT** refer to another translation.

A translation component **SHALL** contain the translations of every localizable string in the translated component, even if the translated string is identical to the original string. It **MAY** contain additional strings that do not appear in the translated component.

To some degree, translations and the components they translate can version independently. The versioning relationship between a translation and the translated component is explained in the sections describing `localizedDataSemanticVersion` (§3.19.31), populated by translations, and `requiredMinimumLocalizedDataSemanticVersion` (§3.19.32), populated by translated components.

A translation **SHOULD** include the value `"localizedData"` in its `contents` array (§3.19.29). It **MAY** also include the value `"nonLocalizedData"`.

To facilitate the identification of translations that are associated with a given component, a `toolComponent` **SHOULD** populate its `guid` property (§3.19.6), and a translation for that component **SHOULD** set its `guid` property to the same value.

  

In many cases, a new version of a `toolComponent` defines new localizable strings or requires changes to existing ones (for example, when the tool defines new analysis rules). But in some cases, a new version of a `toolComponent` can use existing translations (for example, in the case of a bug fix release). To ensure that new translations are created only when necessary, a translation component **SHOULD** populate `localizedDataSemanticVersion` (§3.19.31), and a translatable component **SHOULD** populate `minimumRequiredLocalizedDataSemanticVersion` (§3.19.32). See the descriptions of those two properties for an explanation of the interaction between them.

> EXAMPLE: In this example, a French translation is available. It translates localizable component-level properties such as `toolComponent.name` (§3.19.8), as well as rule-level properties such as `reportingDescriptor.shortDescription` (§3.49.9). The translation can be used because its `localizedDataSemanticVersion` property (§3.19.31) is compatible with the translated component's `minimumRequiredLocalizedDataSemantic` version property (§3.19.32).

```
{                                   # A run object (§3.14).
  "tool": {                         # See §3.14.6.
    "driver": {                     # See §3.18.2.
      "name": "CodeScanner",
      "semanticVersion": "3.3",     # See §3.19.12.
      "minimumRequiredLocalizedDataSemanticVersion": "3.1",
      ...
      "rules": [
        {
          "id": "CA2101",
          "shortDescription": {
            "text": "Do not do dangerous things."
          }
        }
      ]
    }
  },
  "translations": [
    {                               # A toolComponent object.
      "language": "fr-FR",
      "semanticVersion": "3.1.3",
      "localizedDataSemanticVersion": "3.1.2",
      "contents": [
        "localizedData"
      ],
      "translationMetadata": {
        "name": "French translation for CodeScanner"
      },
      "name": "<The tool name 'CodeScanner' translated into French>",
      ...
      "rules": [
        {
          "id": "CA2101",
          "shortDescription": {
            "text": "<'Do not do dangerous things.' Translated into French>"
          }
        }
      ]
    }
  ],
  ...
}
```

### 3.19.5 Policies

A policy is a set of rule configurations that specify how results that violate the rules defined by a particular tool component are to be treated.

A policy is represented by a `toolComponent` object. A policy specifies the tool component to which it applies by way of its `associatedComponent` property (§3.19.33).

A policy **SHALL** contain a `rules` property (§3.19.23), each `reportingDescriptor`-valued (§3.49) element of which in turn contains a `defaultConfiguration` property (§3.49.14). Each element of the `rules` array **SHALL** correspond to a rule defined by the associated component. The `rules` array **MAY** contain elements describing any or all of the rules defined by the associated component. The elements of the `rules` array **MAY** alter rule properties such as `level` (§3.50.3), and **MAY** enable or disable rules. In this way, the policy defines the code analysis standard that is expected of the engineering team.

Policies **SHALL** be stored in the `run.policies` array (§3.14.10).

A SARIF consumer **MAY** offer the user the option of treating results according to the associated component's default rule configuration (possibly modified by command line options stored in `theInvocation.ruleConfigurationOverrides` (§3.20.5), by configuration files, by environment variables, or by any other means), or according to the configuration defined by a selected element of `run.policies`. If the user selects a policy, then for any result that violates a rule covered by that policy, the SARIF consumer **SHALL** treat the result according to the policy, regardless of the associated component's default configuration, regardless of any configuration overrides, and regardless of whether the `result` object (§3.27) itself specifies a configuration property such as `level` (§3.27.10).

> NOTE: The rationale is that when a user asks to see how a policy views a set of results, they want to see exactly what the policy has to say, regardless of any configuration options that might have been selected when the log was created.

> EXAMPLE: In this example, the tool driver defines rule `CA2101` to be a warning and disables rule `CA2551` by default. However, the corporate security policy specifies that a violation of rule `CA2101` is an error and requires rule `CA2551` to be run. The presence of `run.policies` allows a SARIF viewer to display the results according to the tool's view or the policy's view.

```
{                                   # A run object (§3.14).
  "tool": {                         # See §3.14.6.
    "driver": {                     # See §3.18.2.
      "name": "CodeScanner",
      "rules": [                    # See §3.19.23.
        {                           # A reportingDescriptor object (§3.49).
          "id": "CA2101",
          "defaultConfiguration" {  # See §3.49.14.
            "level": "warning"
          }
        },
        {
          "id": "CA2551",
          "defaultConfiguration": {
            "level": "warning",
            "enabled": false
          }
        }
      ]
    }
  },
  "policies": [
    {                               # A toolComponent object (§3.19).
      "name": "Example Corp. Security Policy",
      "semanticVersion": "7.0",
      "rules": [
        {
```

  

```
          "id": "CA2101",
          "defaultConfiguration" {
            "level": "error"
          }
        },
        {
          "id": "CA2551",
          "defaultConfiguration" {
            "enabled": true
          }
        }
      ]
    }
  ]
}
```

### 3.19.6 guid property

A `toolComponent` object **MAY** contain a property named `guid` whose value is a GUID-valued string (§3.5.3) that provides a unique, stable identifier for the component. `guid` **SHALL NOT** vary between versions of a given component.

### 3.19.7 Product hierarchy properties

The `name` (§3.19.8) or `fullName` (§3.19.9), `product` (§3.19.10), and `productSuite` (§3.19.11) properties establish a hierarchy of related software: the tool component identified by `name` and/or `fullName` is part of the product named by `product`, which in turn is part of the product suite identified by `productSuite`.

### 3.19.8 name property

A `toolComponent` object **SHALL** contain a property named `name` whose value is a localizable string (§3.5.1) containing the name of the tool component.

EXAMPLE 1: `"CodeScanner"`

EXAMPLE 2: `"CodeScanner Security Rules Plugin"`

EXAMPLE 3: `"CodeScanner configuration file"`

### 3.19.9 fullName property

A `toolComponent` object **MAY** contain a property named `fullName` whose value is a localizable string (§3.5.1) containing the name of the tool component along with its version and any other useful identifying information, such as its locale.

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

### 3.19.10 product property

A `toolComponent` object **MAY** contain a property named `product` whose value is a localizable string (§3.5.1) containing the name of the product to which the tool component belongs.

EXAMPLE: `"product": "Example Software Corp. Security Scanner"`

### 3.19.11 productSuite property

A `toolComponent` object **MAY** contain a property named `productSuite` whose value is a localizable string (§3.5.1) containing the name of the suite of products to which the tool component belongs.

EXAMPLE: `"productSuite": "Example Software Corp. Quality Tools"`

  

## ~~3.13.4~~3.19.12 semanticVersion property

~~In a log file produced by an analysis tool, a tool~~A `toolComponent` object **MAY** contain a property named `semanticVersion` whose value is a string containing the tool component's version in a format that conforms to the syntax and semantics specified by Semantic Versioning [SEMVER].

EXAMPLE~~ 1:~~

```
~~"tool": {~~
  : "semanticVersion": "1.1.2-beta.12"
~~}~~
```

NOTE 1: Semantic versions are 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.

Unless the author of the converter knows that the version number of the tool from which it converts is intended to be interpreted according to Semantic Versioning [SEMVER~~A~~], the converter **SHALL NOT** emit the `semanticVersion` property in `run.tool` (§3.14.6~~.~~

~~NOTE 2: The rationale is that an analysis tool knows whether~~ ), although of course it may emit 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~~own `semanticVersion` property (the one in `run.conversion.tool` (§3.22.2)).

## 3.19.13 ~~In a log file produced by an analysis tool, a `tool`~~version property

A `toolComponent` object **MAY** contain a property named `version` whose value is a string containing the tool component's version in whatever format the ~~tool~~component natively provides.

~~A converter **SHALL** emit the `version` property.~~

~~fileVersion~~NOTE: Plugins are often binary files whose version can be determined; configuration files are typically text files with no embedded version information.

## ~~3.13.5~~3.19.14 dottedQuadFileVersion property

If the operating system on which the tool runs provides a value for the file version of the ~~tool's~~tool component's primary executable file, and if that value logically consists of an ordered set of four non-negative integers, then the ~~`tool`~~`toolComponent` object **MAY** contain a property named ~~`fileVersion`~~`dottedQuadFileVersion` whose value is a string representation of that file version in this syntax:

```
dottedQuadFileVersion = non negative integer, 3*(".", non negative integer);
```

where the `non negative integer`s follow the logical order of the components of the file version.

If the operating system does not provide such a value, the ~~`fileVersion`~~`dottedQuadFileVersion` property **SHALL** be absent.

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

### 3.19.15 releaseDateUtc property

A `toolComponent` object **MAY** contain a property named `releaseDateUtc` whose value is a string in the format specified in §3.9, specifying the UTC date (and optionally, the time) of the component's release.

### 3.13.63.19.16 downloadUri property

A tooltoolComponent object **MAY** contain a property named `downloadUri` whose value is a localizable string (§3.5.1) containing the absolute URI [RFC3986] from which this version of the tool component can be downloaded.

#### 3.13.71.1.1 language property

A NOTE: This property is localizable to allow different language versions of a tool to be downloaded from their own URIs.

### 3.19.17 informationUri property

A `toolComponent` object **SHOULDMAY** contain a property named languageinformationUri whose value is a localizable string (§3.5.1specifying) containing the absolute URI [RFC3986] at which information about this version of the tool component can be found.

> NOTE: This property is localizable to allow tool information in different languages to be found at different URIs.

### 3.19.18 organization property

A `toolComponent` object **MAY** contain a property named `organization` whose value is a localizable string (§3.5.1) containing the name of the company or organization that produced the tool component.

> EXAMPLE: `"organization": "Example Software Corp."`

### 3.19.19 shortDescription property

A `toolComponent` object **MAY** contain a property named `shortDescription` whose value is a localizable `multiformatMessageString` object (§3.12, §3.12.2) containing a brief description of the tool component.

The `shortDescription` property **SHOULD** be a single sentence that is understandable when visible space is limited to a single line of text.

### 3.19.20 fullDescription property

A `toolComponent` object **MAY** contain a property named `fullDescription` whose value is a localizable `multiformatMessageString` object (§3.12, §3.12.2) containing a comprehensive description of the tool component.

The beginning of `fullDescription` (for example, its first sentence) **SHOULD** provide a concise description of the tool component, 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 `shortDescription` (§3.19.19). Tools that do not construct `fullDescription` in this way **SHOULD** provide a value for `shortDescription`.

> NOTE: The rationale for this guidance is that in the absence of `shortDescription`, a viewer with limited display space might display a truncated version of `fullDescription`, for example, the first sentence (if a sentence is identifiable), the first paragraph, or the first 100 characters. If this guidance is not followed, that truncated description might not be understandable.

  

## 3.19.21 language property

Depending on the circumstances, a `toolComponent` object either **SHALL** or **MAY** contain a property named `language` whose value is a string specifying the language of the localizable strings (§3.5.1) contained in the component (except for those in the `translationMetadata` property (§3.19.27~~of the messages produced by the tool, in~~)), in a subset of the format specified by the language tags standard [RFC5646]. ~~If this property is absent, it **SHALL** default to "en-US".~~

The subset consists of strings conforming to the syntax

```
language value = language code, "-", country code;

language code = ? ISO 2-character language name [ISO639-1:2002] ?;

country code = ? ISO country code [ISO3166-1:2013] ?;
```

If this object represents a translation (see §3.19.4), `language` **SHALL** be present; otherwise it **MAY** be present.

If this property is absent, it **SHALL** default to `"en-US"`.

EXAMPLE 1: The ~~tool~~ language is region-neutral English:

```
"tool": {
  "language": "en"
}
```

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

```
"tool": {
  "language": "fr-FR"
}
```

## ~~3.13.8~~3.19.22 The ~~language~~globalMessageStrings property ~~specifies:~~

A `toolComponent` object **MAY** contain a property named `globalMessageStrings` whose value is an object (§3.6) each of whose property values is a localizable `multiformatMessageString` object (§3.12, §3.12.2). The property names correspond to `id` properties (§3.11.10) within `message` objects (§3.11).

EXAMPLE:

```
"driver": {                          # A toolComponent object (§3.19).
  "globalMessageStrings": {
    "call": {                        # A multiformatMessageString object (§3.12).
      "text": "Function call",
      "markdown": "Function **call**"
    },
    "return": {
      "text": "Function return",
      "markdown": "Function **return**"
    }
  }
}
```

NOTE: The ~~language of the~~ message strings ~~contained~~in this property are not associated with a single rule (hence the "global" in the property name.

## 3.19.23 rules property

A `toolComponent` object **MAY** contain a property named `rules` whose value is an array of zero or more unique (§3.7.3) `reportingDescriptor` objects (§3.49) each of which provides information about an analysis rule supported by the tool component.

Some tools use the same identifier to refer to multiple distinct (although logically related) rules. Therefore, the `id` properties (§3.49.3) of the `reportingDescriptor` objects do not need to be unique within the array.

> EXAMPLE: In this example, two distinct but related rules have the same rule id. They are distinguished by their message strings.

```
"driver": {                          # A toolComponent object (§3.19).
  "name": "CodeScaner",
  "rules": [
    {                                # A reportingDescriptor object (§3.49).
      "id": "CA1711",
      "shortDescription": {
        "text": "Certain type name suffixes should not be used."
      },
      "messageStrings": {
        "default": {
          "text": "Rename type name {0} so that it does not end in '{1}'."
        }
      }
    },
    {
      "id": "CA1711",
      "shortDescription": {
        "text": "Certain type name suffixes have preferred alternatives."
      },
      "messageStrings": {
        "default": {
          "text": "Either replace the suffix '{0}' in member name '{1}' with
                  the suggested numeric alternate or provide
                  a more meaningful suffix."
        }
      }
    }
  ]
}
```

## 3.19.24 notifications property

A `toolComponent` object **MAY** contain a property named `notifications` whose value is an array of zero or more unique (§3.7.3) `reportingDescriptor` objects (§3.49) each of which provides information about a notification provided by the tool component.

A tool might use the same identifier to refer to multiple distinct (although logically related) notifications. Therefore, the `id` properties (§3.49.3~~text (§)~~) of the `reportingDescriptor` objects do not need to be unique within the array.

> EXAMPLE: In this example, two distinct but related notifications have the same id. They are distinguished by their descriptions and message strings.

```
"driver":                            # A toolComponent object (§3.19).
  "notifications": [
    {                                # A reportingDescriptor object (§3.49).
      "id": "ERR0001",
      "level": "error",
      "shortDescription": {
        "text": "A plugin could not be loaded because it does not exist."
      },
      "messageStrings": {
        "default": "Cannot load plugin '{0}' because it was not found."
      }
    },
    {
```

  

```
      "id": "ERR0001",
      "level": "error",
      "shortDescription": {
        "text": "A plugin could not be loaded because it is not signed."
      },
      "messageStrings": {
        "default": "Cannot load plugin '{0}' because it is not signed."
      }
    }
  ]
}
```

### 3.19.25 taxa property

A `toolComponent` object **MAY** contain a property named `taxa` whose value is an array of zero or more unique (§3.7.3) `reportingDescriptor` objects (§3.49) each of which provides information about a taxon defined by the component.

If the `toolComponent` describes a standard taxonomy (for example, the Common Weakness Enumeration [CWE™]), it **SHALL NOT** contain `rules` (§3.19.23) or `notifications` (§3.19.24).

> NOTE: Tool components representing standard taxonomies are stored in `run.taxonomies` (§3.14.8), but will typically be persisted to external property files (see §3.15.2).

If the `toolComponent` describes a tool driver or plugin that defines its own custom taxonomy, it **MAY** contain all of `rules`, `notifications`, and `taxa`.

> EXAMPLE: In this example, a `toolComponent` object represents the Common Weakness Enumeration.

```
{                                    # A toolComponent object.
  "name": "CWE",
  "version": "3.2",
  "guid": "11111111-1111-1111-1111-111111111111",
  "releaseDateUtc": "2019-01-03",
  "informationUri": "https://cwe.mitre.org/data/published/cwe_v3.2.pdf/",
  "downloadUri": "https://cwe.mitre.org/data/xml/cwec_v3.2.xml.zip",
  "organization": "MITRE",
  "shortDescription": {
    "text": "The MITRE Common Weakness Enumeration"
  },
  "taxa": [
    {
      "id": "327",
      "name": "BrokenOrRiskyCryptographicAlgorithm",
      "shortDescription": {
        "text": "Use of a broken or risky cryptographic algorithm."
      },
      "defaultConfiguration": {
        "level": "warning"
      }
    },
    ...
  ]
}
```

### 3.19.26 supportedTaxonomies property

A `toolComponent` object **MAY** contain a property named `supportedTaxonomies` whose value is an array of zero or more unique (§3.7.3) `toolComponentReference` objects (§3.54) each of which refers to a taxonomy (§3.19.3~~richText (§) properties of~~) that the component uses to classify results.

  

A toolComponent object that contains a supportedTaxonomies property **SHALL** declare which taxa (if any) each of its rules falls into by providing the relationships property (§3.49.15) as appropriate on each reportingDescriptor object (§3.49) in its rules array (§3.19.23).

> NOTE: A SARIF consumer could infer the set of taxonomies that a component supports by examining the set of relationships properties of each element of toolComponent.rules. The supportedTaxonomies property is a convenience, intended to enable consumers to see this information at a glance.

If a toolComponent supports a custom taxonomy, it **SHOULD** include a reference to itself in supportedTaxonomies.

> EXAMPLE: In this example, a toolComponent claims to support the Common Weakness Enumeration [CWE message object (§) in the containing™], and also supports a custom taxonomy.

```
{                                   # A run object (§3.14).)
```

1. The language of any embedded resources (§) contained in the resources property (§) of the containing run object.

### 3.13.9 resourceLocation property

```
If a SARIF producer provides external resources (§) for languages other than
the tool's declared language (§), the   "tool": {                       # See
§3.14.6.
    "driver": {                   # See §3.18.2.
      "name": "CodeScanner",
      "guid": "22222222-2222-2222-222222222222",
      "rules": [                  # See §3.19.23.
        ...
      ],
      "taxa": [                   # See §3.19.25. Here, defines a custom
        ...                       #  taxonomy.
      ]
      "supportedTaxonomies": [
        {                         # A toolComponentReference object (§3.54).
          "name": "CWE",          # Declares support for CWE.
          "index": 0,
          "guid": "11111111-1111-1111-1111-111111111111"
        },
        {
          "name": "CodeScanner",  # Declares support for its custom taxonomy.
          "guid": "22222222-2222-2222-222222222222"
        }
      ]
    }
  },
  "taxonomies": [
    {                           # A toolComponentReference object.
      "name": "CWE",
      "version": "3.2",
      "guid": "11111111-1111-1111-1111-111111111111",
      ...
      "taxa": [
        ...
      ]
    }
  ],
  ...
}
```

  

## 3.19.27 translationMetadata property

If a `toolComponent` object represents a translation (§3.19.4 ~~object~~), it **SHALL** contain a property named ~~`resourceLocation`~~`translationMetadata` whose value is a `translationMetadata` object (§3.26) that contains descriptive information about the translation itself, as opposed to describing the component whose localizable strings (§3.5.1) it translates. Otherwise, `translationMetadata` **SHALL** be absent.

## 3.19.28 locations property

A `toolComponent` object **MAY** contain a property named `locations` whose value is an array of zero or more unique (§3.7.3) `artifactLocation` objects (§3.4~~`fileLocation` object (§)~~) each of which specifies the location of ~~a directory containing~~one of the ~~tool's SARIF resource~~ files comprising this tool component.

## 3.19.29 contents property

A `toolComponent` object **SHOULD** contain a property named `contents` whose value is an array of zero or more unique (§3.7.3) strings each of which is one of the following values with the specified meanings:

- `"localizedData"`: The component includes localizable strings (§3.5.1) such as rule messages.
- `"nonLocalizedData"`: The component includes non-localizable properties such as rule severity levels.

If `contents` is absent, it **SHALL** default to `[ "localizedData", "nonLocalizedData" ]`.

> NOTE: The purpose of this property is to help protect components from misuse. Within a SARIF file, the component types are all stored in their own properties, so there is no danger of mistaking, for example, a translation (stored in `run.translations` (§3.14.9)) for a policy (stored in `run.policies` (§3.14.10)). But components such as translations and policies are typically authored independently from a tool and stored separately from its log files. The author of a translation (which contains only `"localizedData"`) can help prevent its misuse as a policy (which requires `"nonLocalizedData"`) by setting `contents` to `[ "localizedData" ]`.
>
> For example, a user might specify the path to a policy file on a tool's command line. If the specified file does not claim to contain `"nonLocalizedData"`, the tool could conclude that the file does not contain a policy and warn the user.

## 3.19.30 isComprehensive property

A `toolComponent` object **SHOULD** contain a property named `isComprehensive` whose value is a Boolean that is `true` if the component contains complete information for the content types specified by `contents` (§3.19.29) and `false` otherwise.

If `isComprehensive` is absent, it **SHALL** default to `false`.

> NOTE: This property is useful because tools are permitted to emit `rules` (§3.19.23), `notifications` (§3.19.24), or `taxa` (§3.19.25) properties that contain only those items relevant to the current run. For example, a tool might define hundreds of rules, but if a scan detects violations of only two of them, then the `rules` property (if it is present at all, which it does not need to be) need only contain metadata for those two rules.
>
> So, for example, the author of a translation (§3.19.4) would want to work from a log file whose `contents` array includes `"localizedData"` and whose `isComprehensive` property is set to `true`. Similarly, the author of a policy (§3.19.5) would want to work from a log file whose `contents` array contains `"nonLocalizedData"` and whose `isComprehensive` property is set to `true`.

  

### 3.19.31 localizedDataSemanticVersion property

If a `toolComponent` object represents a translation (§3.19.4), it **SHOULD** contain a property named `localizedDataSemanticVersion` whose value is a string that specifies the semantic version [SEMVER] of the translated strings. Otherwise, `localizedDataSemanticVersion` **MAY** be present, in which case it represents the semantic version of the localizable strings (§3.5.1) that are present in this component.

If `localizedDataSemanticVersion` is absent, it **SHALL** default to `thisObject.semanticVersion` (§3.19.12).

> NOTE 1: See the description of `minimumRequiredLocalizedDataSemanticVersion` (§3.19.32) for an explanation of how these two properties interact.

> NOTE 2: In a translation, `localizedDataSemanticVersion` will usually be the same as `semanticVersion`. They will differ only if it is necessary to revise the translation component to correct an error unrelated to the translated strings, for example, an error in its `translationMetadata` (§3.19.27). In that case, `semanticVersion` would be incremented but `localizedDataSemanticVersion` would not.

### 3.19.32 minimumRequiredLocalizedDataSemanticVersion property

If a `toolComponent` object does not represent a translation (§3.19.4), it **SHOULD** contain a property named `minimumRequiredLocalizedDataSemanticVersion` whose value is a string that specifies the minumum semantic version [SEMVER] of the translated strings that it requires. Otherwise, `minimumRequiredLocalizedDataSemanticVersion` **SHALL** be absent.

If `minimumRequiredLocalizedDataSemanticVersion` is absent, it **SHALL** default to `thisObject.semanticVersion` (§3.19.12).

When a SARIF consumer is seeking a translation for this object, it **SHALL** only accept one whose `localizedDataSemanticVersion` (§3.19.31) is greater than or equal to (in the SEMVER sense) but has the same major version component as `thisObject.minimumRequiredLocalizedDataSemanticVersion`.

> NOTE: `minimumRequiredocalizedDataSemanticVersion` can differ from `semanticVersion` for two reasons. First, successive versions of a translated component (even versions whose minor version component is incremented) might be able to use the same set of translated strings. Second, the translation itself might be versioned if, for example, the translation author discovers a typo or decides to clarify a message string.

EXAMPLE: In this example, the tool is at version 3.3, but it only requires strings at version 3.1, because tool versions 3.2 and 3.3 didn't affect any user-facing localizable strings. Therefore, the translation at index 0 in `theRun.translations` (§3.14.9~~If a SARIF producer does not provide external resources, the `resourceLocation` property **SHALL** be absent.~~

~~If the `fileLocation` object's `uri` property (§) specifies a relative reference, then its `uriBaseId` property (§) **SHOULD** be present, and the `run` object's `originalUriBaseIds` property (§) **SHOULD** contain a property corresponding to the `uriBaseId` property.~~

> ~~EXAMPLE 1: In this example, a subdirectory of the analysis tool's installation directory contains the SARIF resource files.~~

> ~~{~~) is acceptable.

```
{                                    # A run object (§3.14).
  "tool": {                          # See §3.14.6.
    "driver": {                      # See §3.18.2.
      "name": "SecurityScannerCodeScanner",
      "semanticVersion": "3.3",      # See §3.19.12"version": "2.0.
```

  

```
      "minimumRequiredLocalizedDataSemanticVersion": "3.1",
    "resourceLocation": {  ...
    }
  },
  "translations": [
    {                                # A fileLocationtoolComponent object (§).
      "uri": "resourceslanguage": "fr-FR",
      "uriBaseId": "TOOLINSTALLDIR"
    }
  },

  "originalUriBaseIds": {                         # See §.
    "TOOLINSTALLDIR":
"file:///C:/Program%20Files/SecurityScanner/2.0localizedDataSemanticVersion":
"3.1".2",
  }      ...
    }
  ],
  ...
}
```

## 3.19.33 associatedComponent property

If this `toolComponent` object represents a plugin (see §3.18.1), a taxonomy (§3.19.3), a translation (§3.19.4), or a policy (§3.19.5), it **MAY** contain a property named `associatedComponent` whose value is a `toolComponentReference` object (§3.54) which identifies the component (either `theTool.driver` (§3.18.2) or an element of `theTool.extensions` (§3.18.3)) to which this plugin, translation, or policy applies. If `associatedComponent` is absent, it **SHALL** default to a reference to `theTool.driver`.

> NOTE: The scenario for a taxonomy component to have an `associatedComponent` property is when a party other than the tool vendor defines a custom taxonomy to categorize the rules defined by a specific tool. In this case, `associatedComponent` would specify the tool's driver. A custom taxonomy defined by the tool vendor would be defined in in the `taxa` property (§3.19.25) of the driver itself, so `associatedComponent` would not be necessary.

The associated `toolComponent` object **MAY** itself contain an `associatedComponent` property; for example, a translation might be associated with a plugin which in turn is associated with the driver (see §3.18.1}

~~EXAMPLE 2: In this example, the SARIF resource files are available on the analysis tool's web site.~~

```
{                                               # A run object (§).
  "tool": {
    "name": "SecurityScanner",
    "version": "2.0.1",
    "resourceLocation": {                       # A fileLocation object (§).
      "uri": ".",
      "uriBaseId": "RESOURCES"
    }
  },

  "originalUriBaseIds": {                       # See §.
    "RESOURCES": "https://www.example.com/tools/security-scanner/resources/2.0.1"
  }
}
```

~~If a SARIF producer provides web-based external resources, it **SHOULD** structure its resources directory with subdirectories for each program version, as in EXAMPLE 2 above.~~

  

### ~~3.13.10 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.~~

### ~~3.13.11 properties property~~

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

).

## ~~3.14~~3.20 invocation object

### ~~3.14.1~~3.20.1 General

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

### 3.20.2 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.

### ~~3.14.2~~1.1.1 ~~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~~helps 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.

The `commandLine` property is ~~redaction-aware~~redactable (§3.5.2) because it might contain information which it is not appropriate to disclose, such as passwords, tokens, database connection strings, or in some circumstances even the fully qualified path to the tool's executable or script file.

> 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 `commandLine` might contain the redacted string

```
[REDACTED]\DbScanner.exe /connectionString=[REDACTED] /input=*.sql
```

The `commandLine` property might describe a command that would be harmful if it were executed. For this reason, a SARIF consumer that receives ~~of~~ a SARIF log file from an untrusted source **SHOULD NOT** execute the command line without first examining it carefully. In particular, an automated SARIF consumer **SHALL NOT** execute a command line in a SARIF log file from an untrusted source.

EXAMPLE 2: An example of a harmful command line:

```
{                              # An invocation object
  "commandLine": "rm -rf /"
}
```

### ~~3.14.3~~3.20.3 arguments property

An `invocation` object **MAY** contain a property named `arguments` whose value is either `null` or an array of zero or more strings, containing in order the command line arguments passed to the tool from the operating system.

If `arguments` is absent, it **SHALL** default to `null`.

An empty array **SHALL** mean that the tool was invoked with no command line arguments. `null` **SHALL** mean that the command line arguments, if any, are not known.

EXAMPLE: If the tool is implemented as a C# or Java program, `arguments` would contain the contents of the `args` array passed to the entry point method.

NOTE: Although the `commandLine` property (§3.20.2) contains the same information, parsing it is error prone even if one understands the command shell's quoting and escaping conventions. SARIF consumers might find the pre-parsed `arguments` property easier to use.

### ~~3.14.4~~3.20.4 responseFiles property

An `invocation` object **MAY** contain a property named `responseFiles` whose value is either `null` or an array of zero or more unique (§3.7.3~~fileLocation~~) `artifactLocation` objects (§3.4~~),~~) each of which represents a response file specified on the tool's command line.

If `responseFiles` is absent, it **SHALL** default to `null`.

An empty array **SHALL** mean that the tool was invoked with no command line arguments that specified response files. `null` **SHALL** mean that it is not known whether any command line arguments specified a response file.

A SARIF producer **MAY** embed the contents of a response file in the SARIF log file by mentioning the response file in ~~`run.files`~~`theRun.artifacts` (§3.14.15) and providing a value for ~~`file`~~`artifact.contents` (§3.24.8).

EXAMPLE:

```
{                        # An invocation object.
    "commandLine": "/quiet @analyzer.rsp @strict.rsp" @options.rsp,

    "responseFiles": [
      {                  # A fileLocationAn artifactLocation object (§3.4).. ).
        "uri": "analyzer.rsp",
        "uriBaseId": "RESPONSEFILEDIR"
      },
      {
        "uri": "strict.rsp",
        "uriBaseId": "RESPONSEFILEDIR"
      },
      {
        "uri": "options.rsp",
        "uriBaseId": "RESPONSEFILEDIR"
```

  

```
      }
    }
    ...
}
```

### 3.14.53.20.5 attachmentsruleConfigurationOverrides property

An invocation object **MAY** contain a property named attachmentsruleConfigurationOverrides whose value is an array of onezero or more unique (§3.7.3) attachment) configurationOverride objects (§3.51). Each attachment object **SHALL** describe a file relevant to) each of which overrides the invocation of the tool. Typically, these would be files specified on the tool's command line, and therefore mentioned in the commandLinedefaultConfiguration property (§3.49.14) of a reportingDescriptor object (§3.48.7) or the arguments) that describes a rule (that is, a reportingDescriptor object that is an array element of the rules property (§3.19.23) of some toolComponent object (§3.19), if present. They might also be files implicitly consumed by the tool, such as a configuration file.)).

For an example, see EXAMPLE 1 in §.

### 3.14.63.20.6 startTimenotificationConfigurationOverrides property

An invocation object **MAY** contain a property named notificationConfigurationOverrides whose value is an array of zero or more unique (§3.7.3) configurationOverride objects (§3.51) each of which overrides the defaultConfiguration property (§3.49.14) of a reportingDescriptor object (§3.49) that describes a notification (that is, a reportingDescriptor object that is an array element of the notifications property (§3.19.24) of some toolComponent object (§3.19startTime)).

### 3.20.7 startTimeUtc property

An invocation object **MAY** contain a property named startTimeUtc whose value is a string in the format specified in §3.9, specifying the UTC date and time at which the runtool's execution started. The string **SHALL** be in the format specified in §.

### 3.14.73.20.8 endTimeendTimeUtc property

An invocation object **MAY** contain a property named endTimeendTimeUtc whose value is a string in the format specified in §3.9, specifying the UTC date and time at which the runtool's execution ended. The string **SHALL** be in the format specified in §.

### 3.14.83.20.9 exitCode property

If the SARIF producer process did not exit due to a signal, an invocation object **SHOULD** contain a property named exitCode whose value is an integer specifying the process exit code.

If the SARIF producer process exited due to a signal, the exitCode property **SHALL** be absent.

For examples, see §3.20.10.

### 3.14.93.20.10 exitCodeDescription property

If the SARIF producer process did not exit due to a signal, an invocation object **MAY** contain a property named exitCodeDescription whose value is a string describing the reason for the process exit.

EXAMPLE 1:

```
{                         # An invocation object
  "exitCode": 0,
```

  

```
  "exitCodeDescription": "Normal successful completion"
}
```

EXAMPLE 2:

```
{                        # An invocation object
  "exitCode": 2,
  "exitCodeDescription": "File not found"
}
```

### ~~3.14.10~~3.20.11 exitSignalName property

If the SARIF producer process exited due to a signal, an `invocation` object **SHOULD** contain a property named `exitSignalName` whose value is a string containing the name of the signal that caused the process to exit.

If the SARIF producer process did not exit due to a signal, the `exitSignalName` property **SHALL** be absent.

For an example, see §3.20.12.

### ~~3.14.11~~3.20.12 exitSignalNumber property

If the SARIF producer process exited due to a signal, an `invocation` object **MAY** contain a property named `exitSignalNumber` whose value is an integer specifying the numeric value of the signal that caused the process to exit.

If the SARIF producer process did not exit due to a signal, the `exitSignalNumber` property **SHALL** be absent.

EXAMPLE:

```
{                        # An invocation object
  "exitSignalNumber": 3,
  "exitSignalName": "SIGQUIT"
}
```

### ~~3.14.12~~3.20.13 processStartFailureMessage property

If the analysis tool process failed to start, an `invocation` object **MAY** contain a property named `processStartFailureMessage` whose value is a string containing the operating system's message describing the failure.

NOTE: In this case, the SARIF file would not be produced by the analysis tool (since it failed to start), but rather by some other component of the user's engineering system which is responsible for monitoring the operation of the analysis tool.

If the analysis tool process started successfully (regardless of whether or how it subsequently failed), the `processStartFailureMessage` property **SHALL** be absent.

EXAMPLE:

```
{                        # An invocation object
  "processStartFailureMessage": "WebScan.exe is not recognized as a command."
}
```

### ~~3.14.13~~3.20.14 ~~toolExecutionSuccessful~~executionSuccessful property

An `invocation` object ~~**SHOULD**~~**SHALL** contain a property named ~~`toolExecutionSuccessful`~~`executionSuccessful` whose value is a Boolean that is `true` if the engineering system that started the process knows that the analysis tool succeeded, and `false` if the engineering system knows that the tool failed.

  

NOTE: This property is needed because not all programs exit with an exit code of 0 on success and non-0 on failure.

If this property is absent, it **SHALL** default to `false` if the `exitCode` property (§) is present and has a non-zero value; otherwise it **SHALL** default to `true`.

EXAMPLE:

```
{
  "exitCode": 1,
  "exitCodeDescription": "Scan successful; warnings detected.",
  "toolExecutionSuccessfulexecutionSuccessful": true
}
```

## 3.14.143.20.15 machine property

An `invocation` object **MAY** contain a property named `machine` whose value is a redactable (§3.5.2) string containing the name of the machine on which the tool was run.

## 3.14.153.20.16 account property

An `invocation` object **MAY** contain a property named `account` whose value is a redactable (§3.5.2) string containing the name of the account under which the tool was run.

## 3.14.163.20.17 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.

## 3.14.173.20.18 executableLocation property

An `invocation` object **MAY** contain a property named `executableLocation` whose value is a `fileLocation`an `artifactLocation` object (§3.4) specifying the absolute URIlocation of the tool's primary executable file.

Although in general a `fileLocation` object can specify either a relative reference or an absolute URI, the `fileLocation` object that is the value of the `executableLocation` property **SHALL** specify an absolute URI and **SHOULD** follow the guidance in § for non-deterministic absolute URIs.

NOTE 1: This property is defined in the `invocation` object rather than in the `tool``toolComponent` object (§3.19) because the identical tool might be invoked from different paths on different machines.

NOTE 2: This property might duplicate information in the `commandLine` property (§3.20.2). 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.

## 3.14.183.20.19 workingDirectory property

An `invocation` object **MAY** contain a property named `workingDirectory` whose value is an `artifactLocation` object (§3.4a string containing) specifying the fully qualified path name of the process's working directory in(a directory that the operating system associates with the process, with respect to which the analysis tool was invoked.operating system interprets relative file paths).

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

  

## 3.14.193.20.20 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 corresponding property value **SHALL** be an empty string.

NOTE 1: Environment variables might be useful to include in a log file because they might affect the tool's analysis output, for example, by specifying the location of a directory containing plugins (see §3.18.1). However, environment variable names and values are likely to reveal highly sensitive information. For example, on a machine running Microsoft 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 Microsoft Windows,®, it removes the variable from the environment. In Unix,UNIX®, an environment variable can have an empty value.

## 3.14.20 Both the property names and their values are redactable (§3.5.2). A distinct redaction token (§3.14.28toolNotifications property

A configuration) **SHALL** be used for each redacted property name.

NOTE 3: This is necessary to prevent the creation of an object with identical property names, which is invalid in the JSON serialization.

## 3.20.21 toolExecutionNotifications property

An invocation object **MAY** contain a property named toolNotificationstoolExecutionNotifications whose value is an array of zero or more notification objects (§3.58). Each element of the array represents a runtime condition detected by the invoked process, either by the tool's driver or by one of its extensions. The presence within this array of any notification object whose level property (§3.58.6) is "error" **SHALL** mean that the run failed. A SARIF consumer **SHALL NOT** assume that a failed run contains a complete set of analysis results.

NOTE: This is important in compliance scenarios, where, for example, a corporate policy might require that a project's entire code base be analyzed with a specified set of rules.

The information in toolNotificationstoolExecutionNotifications is primarily intended for the developers of the analysis tool, to aid them in diagnosing bugs in the tool. This contrasts with 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: 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 toolNotificationstoolExecutionNotifications, 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 toolNotificationstoolExecutionNotifications and then halt. If the tool exits abnormally, it might not have the opportunity to report the error. But if the tool is running under the control of an orchestration process that can detect the error, that process might add a notification for the error to the log file, or even synthesize a log file to hold the error, if the tool did not have the opportunity to create one.

  

### 3.14.213.20.22 configurationNotificationstoolConfigurationNotifications property

A configurationAn invocation object **MAY** contain a property named configurationNotificationstoolConfigurationNotifications whose value is an array of zero or more `notification` objects (§3.58). Each element of the array represents a condition relevant to the tool's configuration of the tool's driver or one of its extensions. The presence within this array of any `notification` object whose `level` property (§3.58.6) is `"error"` **SHALL** mean that the run failed.

The information in configurationNotificationstoolConfigurationNotifications is primarily intended for the engineers who configure the analysis tool, to aid them in diagnosing errors in the configuration. This contrasts with 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: 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 reportreports the problem in configurationNotificationstoolConfigurationNotifications. The tool might continue to run, reporting results for the rules that are correctly configured.

```
"toolConfigurationNotifications": [
  {                                  # A notification object
(§3.58"configurationNotifications": [
  {
).
    "descriptor": {
      "id": "UnknownRule",","
    },
    "associatedRule": {
      "ruleId": "ABC0001",","
    },
    "level": "warning",
    "message": {
      "text": "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 reportreports the problem in configurationNotificationstoolConfigurationNotifications. The tool might report the problem as a warning and continue to run, or it might report the problem as an error and terminate.

```
"toolConfigurationNotifications": [
  {                                  # A notification object
(§3.58"configurationNotifications": [
  {
).
    "descriptor": {
      "id": "UnknownCommandLineArgument",","
    },
    "level": "error",
    "message": {
      "text": "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~~reports the problem as an error in ~~`configurationNotifications`~~`toolConfigurationNotifications` and then ~~terminate~~terminates.

```
"toolConfigurationNotifications": [
  {                                   # A notification object
(§3.58"configurationNotifications": [
  {
).
    "descriptor": {
      "id": "CannotFindRulePlugin",AccessDenied"
    },
    "level": "error",
    "message": {
      "text": "Cannot find rule pluginread from directory
\"C:\\AnalysisTool\\CustomChecks.dll."code\"."
    }
  }
]
```

## ~~3.14.22~~3.20.23 stdin, stdout, stderr, and stdoutStderr properties

An `invocation` object **MAY** contain any or all of the properties `stdin`, `stdout`, `stderr`, and `stdoutStderr`, whose values are ~~`fileLocation`~~`artifactLocation` objects (§3.4) referring to files that contain the input to and output from the SARIF producer process. `stdin`, `stdout`, and `stderr` refer, respectively, to files containing the contents of the standard input, standard output, and standard error streams. `stdoutStderr` refers to a file containing the interleaved contents of the standard output and standard error streams. This is useful when the output of those two streams was written to the same file by means of command shell redirection syntax such as "`> output.txt 2>&1`".

A SARIF producer **MAY** embed the stream contents in the log file by mentioning the corresponding file in ~~`run.files`~~`theRun.artifacts` (§3.14.15) and providing a value for ~~`file`~~`artifact`.`contents` (§3.24.8).

## ~~3.14.23 properties property~~

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

# ~~3.15~~3.21 attachment object

## ~~3.15.1~~3.21.1 General

An `attachment` object describes ~~a file~~an artifact relevant to the ~~invocation of a tool (see §) or to the~~ detection of a result (see §3.27.26).

A SARIF producer **MAY** embed the contents of an attachment in the log file by mentioning the attachment ~~file~~ in ~~`run.files`~~`theRun.artifacts` (§3.14.15) and providing a value for ~~`file`~~`artifact`.`contents` (§3.24.8).

~~EXAMPLE 1: In this example, `.scanrc` is the configuration file for the tool being run:~~

```
{                                                   # A run object (§).
```

```
  "invocations": [                                        # See §.
    {                                                     # An invocation object (§).
      ...
      "attachments": [                                    # See §.
        {                                                 # An attachment object.
    EXAMPLE      "description": {                                # See §.
            "text": "Configuration file"
          },
          "fileLocation": {                               # See §.
            "uri": "file:///C:/Users/Mary/.scanrc"
          }
        }
      ]
    }
  ]
}
```

EXAMPLE 2: In this example, `image001.png` is a screen shot of the program being analyzed at the point where the result was detected. Note that this example is more appropriate to a dynamic analysis tool than to a static analysis tool.

```
{                                                         # A result object (§3.27).
  ...
  "attachments": [                                        # See §3.27.26.
    {                                                     # An attachment object.
      "description": {                                    # See §3.21.2.
        "text": "Screen shot"
      },
      "fileLocationlocation": {                                       # See §3.21.3.
        "uri": "file:///C:/ScanOutput/image001.png"
      }
    }
  ]
}
```

## 3.15.23.21.2 description property

An `attachment` object **SHOULD** contain a property named `description` whose value is a `message` object (§3.11) describing the role played by the attachment.

## 3.15.33.21.3 fileLocationlocation property

An `attachment` object **SHALL** contain a property named `fileLocationlocation` whose value is a `fileLocation`an `artifactLocation` object (§3.4) that specifies the location of the attachment file.

## 3.15.43.21.4 regions property

An `attachment` object **MAY** contain a property named `regions` whose value is an array of onezero or more unique (§3.7.3) `region` objects (§3.30),) each of which specifies**SHALL** specify a region of interest within the attachment. These `region` objects, and **SHOULD** contain a `message` property (§3.30.14) so a user can understand theirits relevance.

## 3.15.53.21.5 rectangles property

If the attachment is an image file (for example `.png` or `.svg`), anAn `attachment` object **MAY** contain a property named `rectangles` whose value is an array of onezero or more unique (§3.7.3) `rectangle` objects (§3.31),). If the attachment is an image (for example `.png` or `.svg`), each of which specifies`rectangle` object **SHALL** specify an area of interest within the image. These `rectangle` objects, and **SHOULD** contain a `message` property (§3.31.3) so a user can understand theirits relevance.

If the attachment is not an image file,, and `rectangles` is present, its value **SHALL** be absentan empty array.

# ~~3.16~~3.22 conversion object

## ~~3.16.1~~3.22.1 General

A `conversion` object describes how a converter transformed the output of an analysis tool from the analysis tool's native output format into the SARIF format.

> EXAMPLE: In this example, a converter has converted an AndroidStudio output file into a SARIF log file:

```
{
  ...
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "AndroidStudio"
        }
      },
      "conversion": {
        "tool": {                                   # see §3.22.2
          "driver": {
            "name": "SARIF SDK Multitool"
          }
        },
                                                    # see §3.22.3
        "invocation":
          "Sarif.Multitool.exe convert -t AndroidStudio northwind.log"

        "analysisToolLogFileLocation": {            # see §3.22.4
          "uri": "northwind.log",
          "uriBaseId": "$LOG_DIR$"
        }
      },
      "results": [
        ...
      ]
    }
  ]
}
```

## ~~3.16.2~~3.22.2 tool property

A `conversion` object **SHALL** contain a property named `tool` whose value is a `tool` object (§3.18) that describes the converter.

## ~~3.16.3~~3.22.3 invocation property

A `conversion` object **MAY** contain a property named `invocation` whose value is an `invocation` object (§3.20) that describes the invocation of the converter.

## ~~3.16.4~~3.22.4 analysisToolLogFiles property

Some analysis tools produce one or more output files that describe the analysis run as a whole; we refer to these as "per-run" files. ~~Other~~Some tools produce one or more output files for each result; we refer to these as "per-result" files. Some tools produce both per-run and per-result files.

~~If the analysis tool whose output was converted to SARIF produced any per-run files, the~~A `conversion` object **MAY** contain a property named `analysisToolLogFiles` whose value is an array of ~~one~~zero or more unique (§3.7.3~~) `fileLocation`~~) `artifactLocation` objects (§3.4) that specify the locations of ~~those~~the per-run files.

  

If the analysis tool did not produce any per-run files, and `analysisToolLogFiles` is present, its value **SHALL** be ~~absent~~an empty array.

Per-result files are handled by the ~~`result.conversionProvenance`~~`resultProvenance.conversionSources` property (§3.48.7).

# ~~3.17~~3.23 versionControlDetails object

## ~~3.17.1~~3.23.1 General

A `versionControlDetails` object specifies the information necessary to retrieve from a version control system (VCS) the correct revision of the files that were scanned during the ~~containing~~ run ~~(§)~~.

For an example, see §3.14.13.

## ~~3.17.2~~3.23.2 Constraints

A `versionControlDetails` object ~~**SHALL**~~**SHOULD** contain sufficient information to uniquely and permanently identify the revision of the files that were scanned.

> NOTE: The required set of properties depends on the VCS and on the engineering system within which it is used. Consider Git as an example. The `revisionId` property (containing a commit id) would suffice. The `branch` property (§3.23.5) might not suffice because a Git branch is a pointer to the latest commit along a line of development; however, `branch` together with `asOfTimeUtc` (§3.23.7~~`timestamp`~~) might suffice (although that is not an idiomatic use of Git). Similarly, `revisionTag` (§3.23.6~~`tag`~~) might not suffice because a Git tag can be removed, but if the engineering system guaranteed that certain tags (such as those specifying public releases) were stable, then ~~`tag`~~`revisionTag` might suffice.

## ~~3.17.3~~3.23.3 ~~uri~~repositoryUri property

A `versionControlDetails` object **SHALL** contain a property named ~~`uri`~~`repositoryUri` whose value is a string containing an absolute URI [RFC3986] that specifies the location of the repository containing the scanned files.

## ~~3.17.4~~3.23.4 revisionId property

A `versionControlDetails` object **SHOULD** contain a property named `revisionId` whose value is a redactable (§3.5.2) string that uniquely and permanently identifies the appropriate revision of the scanned files.

## ~~3.17.5~~3.23.5 branch property

A `versionControlDetails` object **MAY** contain a property named `branch` whose value is a redactable (§3.5.2) string containing the name of a branch containing the correct revision of the scanned files.

## ~~3.17.6~~3.23.6 ~~tag~~revisionTag property

A `versionControlDetails` object **MAY** contain a property named ~~`tag`~~`revisionTag` whose value is a redactable (§3.5.2) string containing a tag that has been applied to the revision in the VCS.

> NOTE 1: This specification refers to an identifier for a revision in a VCS as a "tag". Different VCSs use different terms; for example, Visual Studio Team Services Version Control calls it a "label".

> NOTE 2: Although VCSs generally allow a revision to have more than one tag, the ~~`tag`~~`revisionTag` property is not an array. The purpose of ~~`tag`~~`revisionTag` is to aid

in identifying a revision so that a scan can be reproduced, not to exhaustively describe the revision.

### ~~3.17.7~~3.23.7 ~~timestamp~~asOfTimeUtc property

A `versionControlDetails` object **MAY** contain a property named ~~`timestamp`~~`asOfTimeUtc` whose value is a string ~~specifying the date and time at which the revision was created. The string **SHALL** be~~ in the format specified in §3.9~~.~~, specifying a UTC date and time that can be used to synchronize an enlistment to the state of the repository as of that time.

> ~~properties~~NOTE: In some VCSs, the "synchronize by date" feature requires the time to be expressed in the server's time zone. In such a case, the SARIF producer would need to know the server's time zone to correctly populate `asOfTimeUtc`.

### ~~3.17.8~~3.23.8 mappedTo property

A `versionControlDetails` object **MAY** contain a property named ~~`properties`~~`mappedTo` whose value is an `artifactLocation` object (§3.4~~a~~) that specifies the location in the local file system to which the root of the repository was mapped at the time of the analysis.

This property ~~bag (§). This allows tools to include information about the VCS revision that is not explicitly~~makes it possible to map any `artifactLocation` to the repository, if any, to which the file belongs. The mapping algorithm **SHALL** be as follows, or any algorithm with the same result (a clarifying example follows):

1. Resolve the `artifactLocation` as far as possible using the procedure specified in §3.14.14~~the SARIF format~~. Denote the resolved `artifactLocation` by `a`.
2. For every `versionControlDetails` object `vcd` in `theRun.versionControlProvenance` (§3.14.13), resolve the `artifactLocation` object specified by `vcd.mappedTo`, again using the procedure specified in §3.14.14. Denote each such resolved `artifactLocation` object by `v`.
3. Let S be the set of all `versionControlDetails` objects `vcd` for which `v.uriBaseId` equals `a.uriBaseId` and `v.uri` is a prefix of `a.uri`.
4. If S is the empty set, then the file specified by `artifactLocation` does not belong to any repository.
5. Otherwise, the file specified by `artifactLocation` belongs to the repository specified by the member of S with the longest `v.uri`.

EXAMPLE: This example illustrates the mapping algorithm. Consider this SARIF file:

```
{
  "originalUriBaseIds": {
    "HOME": {
      "uri": "file:///home/user"
    }
    "PACKAGE_ROOT": {
      "uri": "package",
      "uriBaseId": "HOME"
    },
  },

  "versionControlProvenance": [
    {
      "repositoryUri": "https://github.com/example-corp/package",
      "revisionId": "b87c4e9"
      "mappedTo": {
        "uriBaseId": "PACKAGE_ROOT"
      },
    },
    {
      "repositoryUri": "https://github.com/example-corp/plugin1",
```

  

```
      "revisionId": "cafdac7"
      "mappedTo": {
        "uriBaseId": "PACKAGE_ROOT"
        "uri": "plugin1",
    },
    {
      "repositoryUri": "https://github.com/example-corp/plugin2",
      "revisionId": "d0dc2c0"
      "mappedTo": {
        "uriBaseId": "PACKAGE_ROOT"
        "uri": "plugin2",
    }
  ],

  "results": [
    {
      "ruleId": "CA1000",
      "locations": [
        {
          "physicalLocation": {
            "artifactLocation": {
              "uri": "plugin1/x.c",
              "uriBaseId": "PACKAGE_ROOT"
            }
          }
        }
      ]
    }
  ]
}
```

The object is to determine to which repository, if any, the file `plugin1/x.c` specified by the result location belongs. The algorithm proceeds as follows, using a simplified notation (*uriBaseId*, *uri*) to denote an `artifactLocation`:

## 3.18 Use the information in `originalUriBaseIds` and the procedure specified in §3.14.14file object

1. to calculate the "resolved artifact location" `a`:

   `(PACKAGE_ROOT, plugin1/x.c) → (HOME, package/plugin1/x.c) → (null, file:///home/user/package/plugin1/x.c).`

2. In the same way, calculate the resolved artifact location `v` from the `mappedTo` property of each element `vcd` of the `versionControlProvenance` array:

   - `(PACKAGE_ROOT, null) → (HOME, package) → (null, file:///home/user/package)`
   - `(PACKAGE_ROOT, plugin1) → (HOME, package/plugin1) → (null, file:///home/user/package/plugin1)`
   - `(PACKAGE_ROOT, plugin2) → (HOME, package/plugin2) → (null, file:///home/user/package/plugin2)`

3. The set of `vcd` for which `v.uriBaseId` equals `a.uriBaseId` (which is `null`) and for which `v.uri` is a *prefix* of `a.uri` (which is

  

file:///home/user/package/plugin1/x.c) contains the objects at indices 0 and 1. It does not contain the object at index 2 because file:///home/user/package/plugin2 is not a prefix of file:///home/user/package/plugin1/x.c.

4. The set is not empty (it contains indices 0 and 1).

5. The member of the set for with the longest v.uri is the object at index 1, because file:///home/user/package/plugin1 is longer than file:///home/user/package.

Therefore, the specified file belongs to the repository specified by the versionControlDetails object at index 1, namely https://github.com/example-corp/plugin1.

## 3.24 artifact object

### 3.18.13.24.1 General

A fileAn artifact object represents a single fileartifact.

### 3.18.23.24.2 fileLocationlocation property

Depending on the circumstances, a filean artifact object either **SHALL**, **MAY**, or **SHALL NOT** contain a property named fileLocationlocation whose value is a fileLocationan artifactLocation object (§3.4).

If the fileartifact object represents a top-level fileartifact, then fileLocation **MAY** location **SHALL** be present.

If it is present, the value of its uri property (§) **SHALL** equal the name of the property within run.files (§) whose value is this file object. If it is absent, it **SHALL** be taken to be present and to have a uri property with that same value.

If the fileartifact object represents a nested fileartifact whose location relative to the root of its parent can be expressed only by means of a path, then the fileLocation propertylocation **SHALL** be present, and the value of its uri property **SHALL** be a relative reference [RFC3986] beginning with "/" expressing that path.

If the fileartifact object represents a nested fileartifact 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 fileLocation propertylocation **SHALL** **NOT** be absentpresent.

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 fileLocation property or the offset property (§) or both **SHALL** be present; they **SHALL NOT** both be absent. If the fileLocation property is present, the value of its uri property **SHALL** be a relative reference expressing the path of the nested file within the parent.

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

```
"files": {
  "http://www.example.com/a.zip": {
    "fileLocation": {
      "uri": "http://www.example.com/a.zip"
    },
    "mimeType": "application/zip"
  },
```

  

```
  "http://www.example.com/a.zip#/src/file.c": {
    "fileLocation": {
      "uri": "/src/file.c"
    },
    "mimeType": "x-c",
    "parentKey": "http://www.example.com/a.zip" # See §
  }
}
```

EXAMPLE 2: The `fileLocation` property of the top-level file is omitted. It is interpreted as being present and having a `uri` property with the value `"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": {
    "fileLocation": {
      "uri": "/src/file.c"
    },
    "mimeType": "x-c",
    "parentKey": "http://www.example.com/a.zip"
  }
}
```

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

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

```
"files": {
  "http://www.example.com/a.zip": {
    "mimeType": "application/zip"
  },
  "http://www.example.com/a.zip#/media/b.zip": {
    "fileLocation": {
      "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": {
    "fileLocation": {
      "uri": "/images/c.png"
    },
    "mimeType": "image/png",
    "parentKey": "http://www.example.com/a.zip#/media/b.zip"
  }
}
```

### 3.18.3 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 reference that matches the property name of the parent file's `file` object within `run.files` (§).

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 reference specified by a `file` object's property name within `run.files` does not necessarily contain enough information to do so.~~

### ~~3.18.4~~1.1.1 If the `artifact` object represents a nested artifact~~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 `location` **MAY be** present; if it is absent, then `offset` (§3.24.4~~either the `fileLocation` property (§) or the `offset` property or both~~) **SHALL** be present~~; they **SHALL NOT** both be absent.~~. If ~~the `offset` property~~`location` is present, ~~its~~the value of its `uri` property **SHALL** be ~~that byte offset~~a relative reference expressing the path of the nested artifact within the parent.

For an example, see §3.24.3.

## 3.24.3 parentIndex property

If this `artifact` object represents a nested artifact, then it **SHALL** contain a property named `parentIndex` whose value is the array index (§3.7.4) of the parent artifact's `artifact` object within `theRun.artifacts` (§3.14.15).

If this `artifact` object represents a top-level artifact, then `parentIndex` **SHALL** be absent.

> NOTE: `parentIndex` makes it possible to navigate from the `artifact` object representing a nested artifact to the `artifact` objects representing each of its parent artifacts in turn, up to the top-level artifact.

> EXAMPLE: This example demonstrates two levels of artifact nesting. The top-level artifact is a ZIP archive represented by the `artifact` object at index 0 in the `artifacts` array. The archive contains a word processing document at the specified absolute path from its root; the document is represented by the `artifact` object at index 1. Finally, the document contains an embedded media object of the specified length at the specified offset from its beginning; the media object is represented by the `artifact` object at index 2. The media object's `parentIndex` property refers to its parent document; the document's `parentIndex` property refers to its parent ZIP archive, and the ZIP archive does not have a `parentIndex` property.

```
"artifacts": [
  {
    "location": {
      "uri": "file:///C:/Code/app.zip"
    },
    "mimeType": "application/zip",
  },
  {
    "location": {
```

  

```
      "uri": "/docs/intro.docx",
    },
    "mimeType":
      "application/vnd.openxmlformats-
officedocument.wordprocessingml.document",
    "parentIndex": 0
  },
  {
    "offset": 17522,
    "length": 4050,
    "mimeType": "application/x-contoso-animation",
    "parentIndex": 1
  }
}
```

### 3.24.4 offset property

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

If the `artifact` object represents a top-level artifact, then `offset` **SHALL NOT** be present.

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

If the `artifact` object represents a nested artifact 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 `offset` **SHALL NOT** be present.

If the `artifact` object represents a nested artifact 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 `offset` **MAY** be present; if it is absent, then `location` (§3.24.2) **SHALL** be present. If `offset` is present, its value **SHALL** be that byte offset.

### ~~3.18.5~~3.24.5 length property

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

If `length` is absent, it **SHALL** default to -1, which indicates that the value is unknown (not set).

### ~~3.18.6~~3.24.6 roles property

~~A `file`~~An `artifact` object **MAY** ~~have~~contain a property named `roles` whose value is an array of ~~one~~zero or more unique (§3.7.3~~distinct~~) strings~~,~~ each of which specifies a role that this ~~file~~artifact played in the analysis.

Each array element **SHALL** have one of the following values, with the specified meanings:

- `"analysisTarget"`: The analysis tool was instructed to scan this ~~file~~artifact.
- `"attachment"`: The ~~file~~artifact is an attachment mentioned in ~~`invocation.attachments` (§) or~~ `result.attachments` (§3.27.26).
- `"conversionSource"`: The artifact is an output from an analysis tool in a non-SARIF format that was converted to SARIF.
- `"directory"`: The artifact is a directory (a container for other files and directories) rather than a file.

  NOTE 1: URIs do not represent "directories" in the file system sense. Even if the URI `https://www.example.com/dir/file` addresses a resource, the URI `https://www.example.com/dir` might also address a resource. Nonetheless, if the analysis tool knows that `https://www.example.com/dir` is not itself a resource, but only a prefix for other URIs that *are* resources, it is appropriate for the tool to mark `https://www.example.com/dir` with the `"directory"` role.

- `"externalPropertyFile"`: The artifact is an external property file (§4).
- `"responseFile"`: The fileartifact contains command line arguments to a program, as specified in `invocation.responseFiles` (§3.20.4).
- `"referencedOnCommandLine"`: The artifact was referenced on the command line.
- `"resultFile"`: A result was detected in this fileartifact.
- `"standardStream"`: The fileartifact contains the contents of one of the standard input or output streams, as specified in `invocation.stdin`, `invocation.stdout`, `invocation.stderr`, or `invocation.stdoutStderr` (§3.20.23).
- `"debugOutputFile"`: The artifact contains debug output from the tool.
- `"driver"`: The file belongs to the analysis tool's driver (§3.18.2).
- `"extension"`: The file belongs to one of the analysis tool's extensions (§3.18.3).
- `"taxonomy"`: The file belongs to a taxonomy (§3.19.3).
- `"translation"`: The file belongs to a translation (§3.19.4).
- `"policy"`: The file belongs to a policy (§3.19.5`traceFile`).
- `"repositoryRoot"`: The artifact is the root directory of a source control repository containing files that were analyzed

  NOTE 2: A single run might analyze files from multiple repositories.

- `"tracedFile"`: The analysis tool traced through this fileartifact while executing or simulating the execution of the code under test.
- `"memoryContents"`: The artifact contains the contents of a portion of memory.
- `"userSpecifiedConfiguration"`: The artifact is a configuration file provided by the user.
- `"toolSpecifiedConfiguration"`: The artifact is a configuration file provided by the tool.

The following role values denote filesartifacts that have changed since the some previous time which we refer to as the "baseline run. If `baselineInstanceGuid`time."

A SARIF producer **MAY** determine the baseline time in any way. (For example, if `theRun.baselineGuid` (§3.14.5) is present on, the containing `run` object (§),tool might use its value **SHALL** specify the baseline run. If any of these role values are present but `baselineInstanceGuid` is absent, the engineering system **SHALL** provide out of band information that determinesstart time as the baseline run.time. Alternatively, the tool might use version control information, such as the time of some commit before the one being analyzed.)

- `"unmodifiedFileunmodified"`: The `file`artifact has not been modified since the baseline `run`time.
- `"modifiedFilemodified"`: The fileartifact was modified after the baseline runtime.
- `"addedFileadded"`: The fileartifact was added after the baseline runtime.
- `"deletedFiledeleted"`: The fileartifact was deleted after the baseline runtime.
- `"renamedFilerenamed"`: The fileartifact was renamed after the baseline runtime. In this case, the `fileartifact` object specifies the new name.
- `"uncontrolledFileuncontrolled"`: The fileartifact is not under version control.

  NOTE 3: The information conveyed by these values could be extracted from a VCS. These properties exist so SARIF consumers can have this information without needing access to the VCS.

### 3.18.73.24.7 mimeType property

A `file`An `artifact` object **SHOULDMAY** contain a property named `mimeType` whose value is a string that specifies the artifact's MIME type [RFC2045]]. For information about the use of mimeType by SARIF viewers, see Appendix Cthe file..

  

## ~~3.18.8~~3.24.8 contents property

~~A `file`~~An `artifact` object **MAY** contain a property named contents whose value is ~~a `fileContent`~~an `artifactContent` object (§3.3) representing the entire contents of the ~~file~~artifact.

## ~~3.18.9~~3.24.9 encoding property

If ~~a `file`~~an `artifact` object represents a text ~~file~~artifact, it **MAY** contain a property named `encoding` whose value is a case-sensitive string that specifies the ~~file's~~artifact's text encoding. The string **SHALL** be one of the character set names ~~specified in~~defined by IANA [IANA-ENC].~~ The property value **SHALL** be case-insensitive.~~

If the ~~`file`~~`artifact` object represents a text ~~file~~artifact and this property is absent, it **SHALL** default to the value of ~~the `defaultFileEncoding` property~~`theRun.defaultEncoding` (§3.14.24~~) of the containing `run` object (§~~), if that property is present; otherwise, the ~~file's~~artifact's encoding **SHALL** be taken to be unknown.

If the ~~`file`~~`artifact` object represents a binary ~~file, the~~artifact, `encoding` ~~property~~ **SHALL** be absent.

> EXAMPLE: In this example, the encoding of output.txt is UTF-16BE (obtained from the default), but the encoding of data.txt is UTF-16LE:

```
{                                        # A run object (§3.14)
  "defaultFileEncodingdefaultEncoding": "UTF-16BE",       # See §3.14.24.

  "files": {    artifacts": [                          # See §3.14.15.
    {
      "location": {
        "uri": "output.txt": {"
      }
      # encoding property omitted
    },

    {
      "location": {
        "uri": "data.txt": {"
      },
      "encoding": "UTF-16LE"
    }
  }]
}
```

## 3.24.10 sourceLanguage property

### 3.24.10.1 General

If an `artifact` object represents a text artifact that contains source code, it **MAY** contain a property named `sourceLanguage` whose value is a hierarchical string (§3.5.4) that specifies the programming language in which the source code is written. If the `artifact` object does not represent a text artifact containing source code, `sourceLanguage` **SHALL** be absent.

For the remainder of this section, we assume that the `artifact` object represents a text artifact that contains source code.

> NOTE 1: This property is intended to help SARIF viewers to render code snippets (§3.30.13) with appropriate syntax coloring.

If the artifact contains source code in a mix of languages, and if it is possible to identify one of those languages as the "primary" language of the artifact, then `sourceLanguage` **SHALL** specify that language.

> NOTE 2: Typically, this is the language implied by the file name extension.

  

EXAMPLE: In an HTML file that contains embedded JavaScript™, `sourceLanguage` would be `"html"`.

If it is not possible to identify a primary language, `sourceLanguage` **MAY** specify any language used in the artifact, or it **MAY** be absent.

NOTE 3: In either case, it is possible to specify a source language for any region by using `region.sourceLanguage` (see §3.30.15).

If `sourceLanguage` is absent, it **SHALL** default to the value of `theRun.defaultSourceLanguage` (§3.14.25). If both `artifact.sourceLanguage` and `theRun.defaultSourceLanguage` are absent, the artifact's source language **SHALL** be taken to be unknown. In that case, a SARIF viewer **MAY** use any method or heuristic to determine the artifact's source language, for example, by examining its file name extension or MIME type, or by prompting the user.

#### 3.24.10.2 Source language identifier conventions and practices

To maximize interoperability, SARIF producers and consumers **SHOULD** conform to the following conventions and practices with respect to the value of this property:

- Producers:
  - Use only lower-case letters, and numbers (for example, `"c"` rather than `"C"`).
  - Spell out symbols (for example, `"csharp"` rather than `"c#"`).
  - To denote a language variant, use the hierarchical string mechanism (for example, `"csharp/7"`).
  - Do not abbreviate (for example, `"visualbasic"`™ rather than `"vb"`).
- Consumers
  - Accept source language identifiers that conform to the above producer conventions.
  - In addition, accept a variety of common industry forms, for example, {`"cplusplus"`, `"c++"`, `"cpp"`}, or {`"javascript"`, `"js"`}.
  - Compare source language identifiers case-insensitively.

Appendix I, "Sample sourceLanguage values," provides sample values for common programming languages.

### ~~3.18.10~~3.24.11 hashes property

~~A `file`~~An `artifact` object **MAY** contain a property named `hashes` whose value is a non-empty object (§3.6~~an array of unique (§) hash objects (§),~~) each of ~~which~~whose property names specifies ~~a hashed value for the file specified by the `file` object, along with~~ the name of ~~the~~a hash function, and each of whose property values represents the value produced by that hash function.

EXAMPLE: In this example, each of the hash functions SHA-256 and SHA-512 were used to compute hash values for the ~~hash~~file.

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

```
The array SHOULD contain an entry whose algorithm property is "sha-256".{
# A file object.
  "hashes": {
    "sha-256": "...",
    "sha-512": "..."
  }
}
```

~~SARIF consumers that need to verify hash values **SHALL** be able to compute a SHA-256 hash.~~

To maximize interoperability, the ~~array **MAY** contain entries whose `algorithm` property is any name that appears~~property names **SHOULD** appear in the IANA registry of hash function textual names [IANA-HASH]. SARIF consumers that need to verify hash values **SHOULD** be able to compute any hash function whose name appears in ~~[].~~the IANA registry.

  

The object **SHOULD** contain a property named "sha-256". SARIF consumers that need to verify hash values **SHALL** be able to compute a SHA-256 hash.

arrayThe object **MAY** contain entriesproperties whose algorithm property doesnames do not appear in [],the IANA registry, but at the expense of interoperability. A SARIF consumer **MAY** implement any hash function, but it does not have to implement any hash function that does not appear in [].the IANA registry.

NOTEIf the hash function is one whose name appears in the IANA registry, the property name **SHALL** equal the name as it appears in the registry (for example, "sha-256" rather than "sha256"); otherwise the property name **MAY** be any suitable name, but it **SHALL NOT** equal any name defined in the IANA registry.

SARIF consumers **SHALL** treat the property name as case insensitive (even when comparing to hash function names in the IANA registry).

Each property value **SHALL** be a string representation of the hash digest of the artifact, computed by the hash function specified by the property name. The string **SHALL** conform to the format produced by the hash algorithm (for example, if the hash algorithm produces a string of hexadecimal digits, the producer would not prepend "0x" to it).

> NOTE 1: The value is represented as a string because hash values are typically represented in hexadecimal notation, and JSON integer values must be decimal.
>
> NOTE 2: 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 fileartifact object defines an arraya set 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 hash function. Compliance systems, for example, will favor the use of more secure hash functions (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 function (such as MD5 or SHA-1) even though they have known weaknesses that allow adversaries to more easily generate hash collisions.
>
> To populate the hashes property, an analysis tool needs 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 hash functions required by those systems. Analysis tools that are configurable to produce hashes with a variety of commonly used hash functions will interoperate most easily with such systems.

### 3.18.113.24.12 lastModifiedTimelastModifiedTimeUtc property

A fileAn artifact object **MAY** contain a property named lastModifiedTimelastModifiedTimeUtc whose value is a string in the format specified in §3.9, specifying the UTC date and time at which the fileartifact was most recently modified. The string **SHALL** be in the format specified in §.

> NOTE: In scenarios where a tool has analyzed files on a network file share or on a local disk, an engineering system might use this property, rather than hashes (§3.24.11), as the most lightweight mechanism to determine whether the analysis needs to be repeated.

  

### 3.18.123.24.13 propertiesdescription property

An `artifact` object **MAY** have a property named `description` whose value is a `message` object (§3.11) that describes the artifact.

## 3.25 specialLocations object

### 3.25.1 General

A `specialLocations` object defines locations of special significance to SARIF consumers.

> NOTE: This version of SARIF defines only one such location, `displayBase` (§3.25.2`file`). In the future, other specially treated locations might be defined.

### 3.25.2 displayBase property

A `specialLocations` object **MAY** contain a property named `displayBase` whose value is an `artifactLocation` object (§3.4) which provides a suggestion to consumers to display file paths relative to the specified location.

A consumer **MAY** act on this hint as follows:

1. Resolve `displayBase` to a URI (the "base URI") by the procedure defined in §3.14.14 or any procedure with the same result. If the result is not an absolute URI, the procedure fails.
2. Normalize the base URI and the displayed URI by the procedures defined in §3.10.1 and §3.10.2 or any procedures with the same result.
3. If the base URI and the displayed URI have the identical scheme, authority, and initial path segments, then display only the remaining path segments of the displayed URI, or "." if there are no remaining path segments.
4. Otherwise, render the displayed URI as an absolute URI (or in some other appropriate form, such as a (`uriBaseId`, `uri`) pair.

EXAMPLE: Given the following:

```
{                              # A run object (§3.14).
  "originalUriBaseIds": {     # See §3.14.14.
    "WEBHOST": {
      "uri": "http://www.example.com"
    },
    "ROOT": {
      "uri": "file:///"
    },
    "HOME": {
     "uri": "/home/user/",
     "uriBaseId": "ROOT"
    },
    "PACKAGE": {
      "uri": "mySoftware/",
      "uriBaseId": "HOME"
    },
    "SRC": {
      "uri": "src/",
      "uriBaseId": "PACKAGE"
    }
  },

  "specialLocations": {
    "displayBase": {          # An artifactLocation object (§3.4).
      "uri": "",              # Empty string is valid relative reference.
      "uriBaseId": "PACKAGE"
    }
  }
```

  

```
}
```

These equivalent locations would display as `src/f.c` because the scheme, authority, and initial path segments match:

```
{
  "uri": "f.c",
  "uriBaseId": "SRC"
}

{
  "uri": "src/f.c",
  "uriBaseId": "PACKAGE"
}

{
  "uri": "file:///home/user/mySoftware/src/f.c"
}
```

These equivalent locations would display as `/usr/include/stdio.h` because the scheme and authority match, but not the path:

```
{
  "uri": "/usr/include/stdio.h",
  "uriBaseId": "ROOT"
}

{
  "uri": "file:///usr/include/stdio.h"
}
```

These equivalent locations would display as `http://www.example.com/hello` because the scheme and authority do not match:

```
{
  "uri": "hello",
  "uriBaseId": "WEBHOST"
}

{
  "uri": "http://www.example.com/hello"
}
```

If `displayBase` were changed to

```
"displayBase": {
  "uri": "",
  "uriBaseId": "HOME"
}
```

the URIs displayed as `src/f.c` would instead be displayed as `mySoftware/src/f.c`. All other display values would be unchanged.

## 3.26 translationMetadata object

### 3.26.1 General

A `translationMetadata` object describes a translation. It is necessary because in a `toolComponent` object that represents a translation, the usual descriptive properties `name` (§3.19.8), `fullName` (§3.19.9), *etc.* contain the translations of the corresponding strings in the `toolComponent` being translated; therefore, they are not available to hold descriptive information for the translation itself.

  

Because they occur only in toolComponent objects that represent translations, the properties of a translationMetadata object are not themselves localized (§3.5.1whose value is a property bag (§). This allows tools to include information about the file that is not explicitly specified in the SARIF format.).

# 3.19 hash object

## 3.19.1 General

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

EXAMPLE:

```
{                                # A toolComponent object (§3.19).
  "language": "fr-FR",           # The language of the translation (see (§3.19.21{
    "value":"b13ce2678a8807ba0765ab94a0ecd394f869bc81",    # see §
    "algorithm":"sha-256"                                  # see §
}
value).

  "translationMetadata": {   # A translation metadata object.
    "name": "CodeScanner translation for fr-FR ",
    "fullName": "CodeScanner translation for fr-FR by Example Corp.",
    "shortDescription": {
      "text": "A good translation"
    },
    "fullDescription": {
      "text": "A good translation performed by native en-US speakers."
    }
  },

  "name": "(fr-FR translation of translated component's name)",
  "fullName": "(fr-FR translation of translated component's full name)",
  ...
}
```

## 3.19.23.26.2 name property

A hashA translationMetadata object **SHALL** contain a property named valuename whose value is a string representation of the hash digest of some file or collection of files, computed by containing a name for the hash function named in the algorithmtranslation.

## 3.26.3 fullName property (§). For hash functions that compute a numeric value, value SHALL

A translationMetadata object **MAY** contain a hexadecimal representation of the numeric value of the hash digest. A hexadecimal string value **SHALL NOT** include a hexadecimal prefix such as "0x" or a suffix such as "h". A SARIF consumer **SHALL** treat a hexadecimal string value as case-insensitive.

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

## 3.19.3 algorithm property

A hash object **SHALL** contain a property named algorithm named fullName whose value is a string specifying the name of the hash function used to compute the hash value (§). If the hash function is one whose name appears in the IANA registry of hash function textual names [], algorithm **SHALL** contain the name specified in the registry (for example, "sha-256" rather than "sha256"); otherwise, algorithm **MAY** contain any suitable name, but it **SHALL NOT** contain any name defined in the IANA registrycontaining the name of the translation along with any other useful identifying information.

### 3.26.4 shortDescription property

A `translationMetadata` object **MAY** contain a property named `shortDescription` whose value is a `multiformatMessageString` object (§3.12) containing a brief description of the translation.

### 3.26.5 fullDescription property

A `translationMetadata` object **MAY** contain a property named `fullDescription` whose value is a `multiformatMessageString` object (§3.12) containing a comprehensive description of the translation.

### 3.26.6 downloadUri property

A `translationMetadata` object **MAY** contain a property named `downloadUri` whose value is a string containing the absolute URI [RFC3986] from which the translation can be downloaded.

### 3.26.7 informationUri property

A `translationMetadata` object **MAY** contain a property named `informationUri` whose value is a string containing the absolute URI [RFC3986SARIF consumers **SHALL** treat `algorithm` as case-insensitive (even when comparing to hash function names in the IANA registry).

] at which information about the translation can be found.

## 3.203.27 result object

### 3.20.13.27.1 General

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

### 3.20.2 Each result is produced by the evaluation of a rule. If `theTool` contains a `reportingDescriptor` object (§3.49) that describes that rule, we refer to that object as `theDescriptor`, and we refer to the `toolComponent` object (§3.19Constraints

At least one of the `message` property (§) and the `ruleMessageId` property (§) **SHALL** be present. If they are both present, and they both refer to message strings that are present in the log file, then those message strings **SHALL** be identical.

> EXAMPLE 1: In this example, `result.message.text` (§) directly contains the message string `"Variable 'pBuffer' is uninitialized."`. `result.ruleId` (§) and `result.ruleMessageId` (§) together designate the rule message string `"Variable '{0}' is uninitialized."` which, along with the contents of `result.message.arguments` (§), produces the identical string.

```
{                                                  # A run object (§).
  "results": [                                     # See §.
    {                                              # A result object (§).
```

) that defines `theDescriptor` as `theComponent`.

```
      "ruleId": "CA2101",                          # See §.
      "message": {                                 # See §.
        "text": "Variable 'pBuffer' is uninitialized.",  # See §.
        "arguments": [ "pBuffer" ]                 # See §.
      },
      "ruleMessageId": "default"                   # See §.
```

  

```
    }
  ],

  "resources": {                                  # See §.
    "rules": {                                    # See §
      "CA2010": {                                 # A rule object (§).
        "messageStrings": {                       # See §3.30.7.
          "default": "Variable '{0}' is uninitialized."
        }
      }
    }
  }
}
```

EXAMPLE 2: In this example, the SARIF log file does not include rule metadata. The SARIF log file is valid even though the external resource string (§) designated by `ruleId` and `ruleMessageId` might not produce the same string as `message.text`.

```
{                                                 # A run object (§).
  "results": [                                    # See §.
    {                                             # A result object (§).
      "ruleId": "CA2101",                         # See §.
      "message": {                                # See §.
        "text": "Variable 'pBuffer' is uninitialized.",  # See §.
        "arguments": [ "pBuffer" ]                # See §.
      },
      "ruleMessageId": "default"                  # See §.
    }
  ]
}
```

### 3.20.33.27.2 Distinguishing logically identical from logically distinct results

Successive runs of the same tool, or even runs of different tools, might detect the same condition in the code. When two `result` objects represent the same condition, we say that the results are "logically identical;" when they represent different conditions, we say that the results are "logically distinct." Two results can be logically identical even if the `result` objects are not identical. For example, if code is inserted into thea file between runs, the same condition might be reported on two different lines.

To avoid reporting the same condition repeatedly, result management systems typically group results into equivalence classes such that results in any one class are logically identical and results in different classes are logically distinct.

Some result management systems do this by calculating a "fingerprint" for each result and considering results with the same fingerprint to be logically identical. A fingerprint is calculated from information contained in the result and might contain readable information from the result.

Other result management systems group results into equivalence classes *without* associating a computed fingerprint with each result, and they denote each equivalence class with an arbitrary unique identifier. This identifier is opaque: it is *not* calculated from information stored in the result, and hence contains no readable information about the result.

Still other result management systems compute a fingerprint, associate an arbitrary unique identifier with the fingerprint, and use that identifier rather than the fingerprint to identify the equivalence class of results.

SARIF accommodates all these types of result management systems. Result management systems that compute fingerprints **SHOULD** populate the `fingerprints` property (§3.27.16). Result management systems that group results into equivalence classes based on an arbitrary unique identifier **SHOULD** populate the `correlationGuid` property (§3.27.4), regardless of whether they also compute a fingerprint.

### 3.20.43.27.3 instanceGuidguid property

A `result` object **MAY** contain a property named `instanceGuidguid` whose value is a GUID-valued string (§3.5.3) defining a unique, stable identifier for the result.

  

Direct SARIF producers and SARIF converters ~~**SHOULD NOT**~~**MAY** but do not need to set this property. A result management system **SHOULD** set this property when it ingests a SARIF log file. If it does so, then later, when a SARIF consumer retrieves results in SARIF format from the result management system, the result management system **SHALL** set this property to the value it assigned.

A result management system **MAY** store multiple results with identical fingerprints (see §3.27.16 and Appendix B), but the ~~`instanceGuid`~~`guid` properties for those results **SHALL** be distinct.

## ~~3.20.5~~3.27.4 correlationGuid property

A `result` object **MAY** contain a property named `correlationGuid` whose value is a GUID-valued string (§3.5.3) that is shared by all results that are considered logically identical, and that is different between any two results that are considered logically distinct.

Direct SARIF producers and SARIF converters **SHOULD NOT** set this property. A result management system **MAY** set this property when it ingests a SARIF log file. If it does so, then later, when a SARIF consumer retrieves results in SARIF format from the result management system, the result management system **MAY** set this property to the value it assigned.

> NOTE: `correlationGuid` and `fingerprints` (§3.27.16) provide two different ways for result management systems to associate results that are logically identical. See §3.27.2 for more information.

## ~~3.20.6~~3.27.5 ruleId property

Depending on the circumstances, a `result` object either **SHALL**, **MAY**, or **SHALL NOT** contain a property named `ruleId` whose value is a hierarchical string (§3.5.4~~containing~~) whose leading components specify the stable~~, opaque~~ identifier ~~for~~of the rule that was evaluated to produce the result. In addition to being stable, `ruleId` **SHOULD** be opaque.

> NOTE: `ruleId` will usually consist entirely of the rule's stable opaque identifier. In some cases, it might be helpful to specify additional hierarchical components to more precisely describe the rule violation.

A SARIF viewer or result management system **MAY** use the additional hierarchical components to allow a user to suppress a subset of the violations of a given rule. A result management system **MAY** also use the additional components to more precisely match results between runs.

> EXAMPLE: In this example, the first result describes a violation of rule `CA2101`. Its `ruleId` consists entirely of the rule's identifier. The second and third results both describe violations of rule `CA5350`. Each of their `ruleId`s specifies an additional hierarchical component that more precisely describes the rule violation. Note that `rule.index` (§3.27.7, §3.52.5~~EXAMPLE 1:~~

```
"results": [
  {
    "ruleId": "CA2101"
    ...
  }
```

> ) for both those results is `1`; despite the additional hierarchical components in `ruleId`, both results describe violations of the same rule.
>
> A SARIF viewer or result management system might allow a user to suppress, for example, only those violations of rule `CA5350` which specify `md5` as the second hierarchical component of `ruleId`; that is, to allow the use of MD5 but still warn about the uses of other weak cryptographic algorithms.

```
{
  "tool": {
    "driver": {
      "name": "CodeScanner",
```

  

```
        "rules": [
          {
            "id": "CA2101",
            "shortDescription": {
              "text": "Specify marshaling for P/Invoke string arguments."
            }
          },
          {
            "id": "CA5350",
            "shortDescription": {
              "text": "Do not use weak cryptographic algorithms."
            }
          }
        ]
      }
    },
    "results": [
      {
        "ruleId": "CA2101",
        "rule": {
          "index": 0
        }
      },
      {
        "ruleId": "CA5350/md5",
        "rule": {
          "index": 1
        }
      },
      {
        "ruleId": "CA5350/sha-1",
        "rule": {
          "index": 1
        }
      }
    ]
]
```

Direct producers **SHALL** emit either or both of `ruleId` and `rule.id` (§3.27.7, §3.52.4~~`ruleId`.~~). If `rule.id` is absent, `ruleId` **SHALL** be present. If `rule.id` is present, `ruleId` **MAY** be present.

For an example of the interaction between `ruleId` and `rule.id`, see §3.52.4.

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

~~Some tools define multiple rules with the same id. If there is more than one rule with the desired, and if the containing `run` object (§) contains a `resources.rules` property (§, §), then instead of containing the rule id, `ruleId` **SHALL** contain a string that equals one of the property names in `resources.rules`. To improve the readability of the log file, this property name **SHOULD** be formed by appending a suffix to the rule id. In this case, the `"id"` property (§) of the specified rule object (§) **SHALL** contains the actual rule id.~~

> ~~EXAMPLE 2: In this example, there is more than one rule with id `CA1711`.The SARIF producer sets `ruleId` to a value that specifies which of the rules with that id is meant. That value is formed by appending the suffix `"-1"` to the rule id. The rule id is specified by `resources.rules["CA1711-1"].id`, which evaluates to `"CA1711"`.~~

```
{                               # A run object (§).
  "results": [                  # See §.
```

  

```
    {                         # A result object (§).
      "ruleId": "CA1711-1",   # Specifies a property name within "rules".
      ...
    }
  ],
  "resources": {              # See §.
    "rules": {                # See §.
      "CA1711-1": {           # A rule object (§).
        "id": "CA1711",       # See §.
        ...
      },
      "CA1711-2": {           # Another rule object with the same rule id.
        "id": "CA1711",
        ...
      }
    }
  }
}
```

### 3.20.71.1.1 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:

Each SARIF converter might synthesize `ruleId` in a different way. Therefore, a SARIF consumer **SHOULD NOT** attempt to compare or combine the output from different converters for the same analysis tool. See Appendix D for more information about production of SARIF by converters.

### 3.27.6 ruleIndex property

If `theDescriptor` exists (that is, if `theTool` contains a `reportingDescriptor` object (§3.49) that describes the rule that was violated), a `result` object **MAY** contain a property named `ruleIndex` whose value is the array index (§3.7.4) of `theDescriptor` within `theComponent.ruleDescriptors` (§3.19.23). Otherwise, `ruleIndex` **SHALL** be absent.

The semantics of `ruleIndex` are identical to the semantics of `reportingDescriptorReference.index` (§3.52.5), and are described there.

If `ruleIndex` and `rule.index` (§3.27.7, §3.52.5) are both present, they **SHALL** be equal.

### 3.27.7 rule property

Depending on the circumstances, a `result` object either **SHALL NOT**, **SHOULD**, or **MAY** contain a property named `rule` whose value is a `reportingDescriptorReference` object (§3.52) that identifies `theDescriptor`. The procedure for looking up a `reportingDescriptor` from a `reportingDescriptorReference` is described in §3.52.3.

If `theDescriptor` does not exist (that is, if `theTool` does not contain a `reportingDescriptor` object (§3.49) that describes the rule that was violated), then `rule` **SHALL NOT** be present.

If `theDescriptor` occurs in `theTool.extensions` (§3.18.3), then `rule` **SHOULD** be present.

> NOTE 1: If `theDescriptor` occurs in `theTool.extensions` and `ruleDescriptorReference` is absent, the SARIF consumer will not be able to locate the rule metadata, even if `ruleIndex` (§3.27.6) is present, because `ruleIndex` alone does not specify which extension contains `theDescriptor`.

If `theDescriptor` occurs in `theTool.driver` (§3.18.2) and `ruleIndex` is absent, then again `ruleDescriptorReference` **SHOULD** be present.

  

NOTE 2: If theDescriptor occurs in theTool.driver and ruleIndex is absent, the SARIF consumer will not be able to locate the rule metadata within theTool.driver.ruleDescriptors.

If theDescriptor occurs in theTool.driver and ruleIndex is present, then ruleDescriptorReference **MAY** be present.

NOTE 3: If theDescriptor occurs in theTool.driver, then ruleIndex suffices to locate the rule metadata within theTool.driver.ruleDescriptors.

If rule.id (§3.52.4) is absent, it **SHALL** default to thisObject.ruleId. If rule.id and thisObject.ruleId are both present, they **SHALL** be equal.

If rule.index (§3.52.5) is absent, it **SHALL** default to thisObject.ruleIndex. If rule.index and thisObject.ruleIndex are both present, they **SHALL** be equal.

If rule is absent, it **SHALL** default to a reportingDescriptorReference object whose id property is set to thisObject.ruleId and whose index property is set to thisObject.ruleIndex.

NOTE: If the relevant rule is defined by the driver (see §3.18.1), which is likely to be the most common case, then ruleId and/or ruleIndex suffice to identify the rule, and take up less space in the log file than rule.

### 3.27.8 taxa property

A result object **MAY** contain a property named taxa whose value is an array of zero or more unique (§3.7.3) reportingDescriptorReference objects (§3.52) each of which refers to a taxon (see §3.19.3) into which this result falls.

If the toolComponent object (§3.19) theComponent that defines the rule that was violated contains a reportingDescriptor object (§3.49) theDescriptor (a member of toolComponent.rules (§3.19.23)) that describes that rule, then thisObject.taxa **SHALL** contain elements corresponding to those elements of theDescriptor.relationships (§3.49.15) that describe taxa into which this result falls. thisObject.taxa does not need to contain elements which correspond to superset or equals relationships; rather, the result **SHALL** implicitly be taken to fall into all the taxa described by those relationships.

NOTE 1: See the example below for an illustration of this point. See §3.53.3 for descriptions of the various types of relationships.

Otherwise (that is, if theDescriptor does not exist), thisObject.taxa **SHALL** contain elements that describe all taxa into which the result falls.

In either case, if there is no toolComponent that defines the taxonomy to which an element of thisObject.taxa refers, then that element (a reportingDescriptorReference object) **SHALL NOT** contain index (§3.52.5) or toolComponent.index (§3.52.7, §3.54.4).

NOTE 2: The rationale for this restriction is that toolComponent.index serves to locate the toolComponent object defining the rule, and index serves to locate the rule within that toolComponent. If there is no relevant toolComponent object, neither of those properties is meaningful. On the other hand, properties such as id (§3.52.4), guid (§3.52.6), toolComponent.name (§3.54.3), and toolComponent.guid (§3.54.5) are useful for readability and for identification, even if the toolComponent itself is absent, so they are permitted.

EXAMPLE: In this example, a tool defines a custom taxonomy (see §3.19.3) consisting of three taxa with ids "SUP", "INC1", and "INC2". The tool emits a result that falls into the taxa "SUP" and "INC2", but not into "INC1". According to relationships[0], "SUP" is a superset of "CA2101"; that is, every result that violates "CA2101" falls into the taxon "SUP". Therefore, it is not necessary to mention "SUP" in theResult.taxa. On the other hand, according to relationships[2], "INC2" is incomparable to

  

"CA2101"; that is, the set of results that violate "CA2101" intersects with but is neither a superset nor a subset of the set of results that fall into the taxon "INC2". Therefore, it is necessary to mention "INC2" in theResult.taxa.

```
{                                     # A run object (§3.14).
  "tool": {
    "driver": {
      "name": "CodeScanner",
      ...
      "rules": [
        {
          "id": "CA2101",
          ...
          "relationships": [
            {
              "target": {
                "id": "SUP",
                "guid": "11111111-1111-1111-1111-111111111111"
              },
              "kinds": [
                "superset"
              ]
            },
            {
              "target": {
                "id": "INC1",
                "guid": "22222222-2222-2222-2222-222222222222"
              },
              "kinds": [
                "incomparable"
              ]
            },
            {
              "target": {
                "id": "INC2"
                "guid": "33333333-3333-3333-3333-333333333333"
              },
              "kinds": [
                "incomparable"
              ]
            }
          ]
        }
      ],
      "taxa": [
        {
          "id": "SUP",
          "guid": "11111111-1111-1111-1111-111111111111",

          ...
        },
        {
          "id": "INC1",
          "guid": "22222222-2222-2222-2222-222222222222",
          ...
        },
        {
          "id": "INC2",
          "guid": "33333333-3333-3333-3333-333333333333",
          ...
        }
      ]
    }
  },
```

  

```
  "results": [
    {
      "ruleId": "CA2101",
      "rule": {
        "index": 0
      },
      "taxa": [
        {
          "id": "INC2",
          "guid": "33333333-3333-3333-3333-333333333333"
        }
      ]
    }
  ]
}
```

### 3.27.9 kind property

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

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

- `"pass"`: The rule specified by the `ruleId` property (§3.27.5), `ruleIndex` (§3.27.6), and/or `rule` (§3.27.7) 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.
- `"open"`: The rule specified by the `ruleId` propertyrule was evaluated, and the tool concluded that there was insufficient information to decide whether a problem exists.

  NOTE 1: This value is used by proof-based tools. Sometimes such a tool can prove that there is no violation (`kind` = `"pass"`), sometimes it can prove that there is a violation (`kind` = `"fail"`), and sometimes it does not detect a violation but is unable to prove that there is none (`kind` = `"open"`). In such a tool, a `kind` value of `"open"` might be an indication that the user should add additional assertions to enabe the tool to determine if there is a violation.

- `"informational"`: The specified rule was evaluated and produced a purely informational result that does not indicate the presence of a problem. (See the example below.)
- `"notApplicable"`: The rule specified by the `ruleId` property was not evaluated, because it does not apply to the analysis target.

  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. It also has a rule that checks the compiler version and produces an informational result:

```
"results": [
  {
    "ruleId": "ABC0001",
    "levelkind": "notApplicable",
    "message": {
      "text": "\"MyTool64.exe\" was not evaluated for rule ABC0001
               because it is not a 32-bit binary."
    },
    "locations": [
      {
        "physicalLocation": {
          "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",
    "kind": "informational",
    "message": {
      "text": "\"MyTool64.exe\" was compiled with Example Corporation
                 Compiler version 10.2.2."
    },
    "locations": [
      {
        "physicalLocation": {
          "uri": "file://C:/bin/MyTool64.exe"
        }
      }
    ]
  }
]
```

- "review": The result requires review by a human user to decide if it represents a problem.

  NOTE 2: This value is used by tools that are unable to check for certain conditions, but that wish to bring to the user's attention the possibility that there might be a problem. For example, an accessibility checker might produce a result with the message "Do not use color alone to highlight important information," with kind = "review". A user might address this issue by visually inspecting the UI.

- "fail": The result represents a problem whose severity is specified by the level property (§3.27.10).

If kind is absent, it **SHALL** default to "fail".

If level has any value other than "none" and kind is present, then kind **SHALL** have the value "fail".

### 3.27.10 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:

- "warning": The rule specified by ruleId was evaluated and a problem was found.
- "error": The rule specified by ruleId was evaluated and a serious problem was found.
- "note": The rule specified by ruleId was evaluated and a minor problem or an opportunity to improve the code was found.
- "none": The concept of "severity" does not apply to this result because the kind property (§3.27.9) has a value other than "fail".

  EXAMPLE: In this example, the tool reports an opportunity to improve the code.

```
"results": [
  {
```

  

```
    "ruleId": "ABC0003",
    "kind": "fail",
    "level": "note",
    "message": {
      "text": "Consider using 'nameof(start)' instead of hard-coding
               the parameter name 'start'."
    },
    "locations": [
      {
        "physicalLocation": {
          "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",
If kind (§3.27.9") has any value other than "fail", then if level": "note",
    "message": {
      "text": "A new version of rule ABC0003 is available."
    }
  }
]
```

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

```
"results": [
  {
    "level": "note",
    "message": {
      "text": "Version 11.0 of SuperLint is now available."
    }
  }
]
```

If the `level` property is absent, it **SHALL** default to the `defaultLevel` property (§) of the `ruleConfiguration` object (§) contained in the `configuration` property (§) of the `rule` object (§) specified by this `result` object's `ruleId` property (§)."`none`", and if it is present, it **SHALL** have the value "`none`".

In that case, ifIf `kind` has the value "`fail`" and `level` is absent, then `level` **SHALL** be determined by the following procedure:

IF `rule` (§3.27.7`run`) is present THEN

  LET `theDescriptor` be the `reportingDescriptor` object (§3.49) that it specifies.

  # Is there a configuration override for the `level` property?

  IF `result.provenance.invocationIndex` (§3.27.29, §3.48.6) is >= 0 THEN

    LET `theInvocation` be the `invocation` object (§3.20) that it specifies.

    IF `theInvocation.ruleConfigurationOverrides` (§3.20.5) is present

        AND it contains a `configurationOverride` object (§3.51) whose descriptor property (§3.51.2) specifies theDescriptor THEN

LET theOverride be that configurationOverride object.

IF theOverride.configuration.level (§3.51.3, §3.50.3) is present THEN

Set level to theConfiguration.level.

ELSE

# There is no configuration override for level. Is there a default configuration for it?

IF theDescriptor.defaultConfiguration.level (§3.49.14, §, §3.50.3) containing this) is present THEN

SET level to theDescriptor.defaultConfiguration.level.

IF level has not yet been set THEN

SET level to "warning".

### 3.27.11 message property

A result does not include a resources.rules property (§, §) (and no external resource file is available), or if the resources.rules property does not specify information for the rule object associated with this result, or if the rule object associated with this result does not specify a configuration.defaultLevel property, then the level property **SHALL** default to "warning".

### 3.20.81.1.1 SHALLmessage property

A result object **MAY** contain a property named message whose value is a message object (§3.11) that describes the result. If the message property is absent, the ruleMessageId property (§) **SHALL** be present. Both message and ruleMessageId **MAY** be present. See § for more information.

The message property **SHOULD** provide sufficient details to allow an end user to resolve any problem that the result might indicate. In particular, it **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 1: This is an example of a message:

```
"results": [
  {
    "message": {
      "text": "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."
    }
  }
]
```

  

## 3.20.9 See §3.11.7ruleMessageId property

A result object MAY contain a property named ruleMessageId whose value is a string that identifies the message within the rule metadata for the rule used in this result. If ruleMessageId is absent, message (§) SHALL be present. Both message and ruleMessageId MAY be present. See § for more information.

If resources.rules (§, §) is present on the containing run object (§), then ruleMessageId SHALL equal one of the property names for the procedure for looking up a message string from a message object, in particular, for the case where the message object occurs as the value of result.message.

EXAMPLE 2: In this example, message.id refers to the property named default defined in the messageStrings property (§) of the rule object (§) whose property name within resources.rules equals the ruleId property (§) of this result object. ruleMessageId MAY also equal one of the property names in the richMessageStrings property (§) of that rule object.

> EXAMPLE 1: In this example, the result object's ruleId and ruleMessageId properties together specify the string reportingDescriptor object identified by "default" within the rule metadata for the rule whose property name within resources.rules is "CA2101".

```
{                                          # A run object (§3.14).
  "tool": {                                # See §3.14.6.
    "driver": {                            # See §3.18.2.
      "name": "CodeScanner",
      "rules": [                           # See §3.19.23.
        {                                  # A runreportingDescriptor object (§3.49).
          "id "results": [
    {                                # A result object (§).
      "ruleId": "CA2101",
      "ruleMessageId": "default",
      ...
    }
  ],

  "resources": {                     # A resources object (§).
    "rules": {
      "CA2101": {
        "id": "CA2101",
          "messageStrings": {
              "default": {           # A multiformatMessageString object
(§3.12"This is the).
                "text": "The default message for this rule.",
                "markdown": "The default message for *this* rule."
              },
              "special": "This is another{
                "text": "Another message, for this rule, used in special
cases.".",
        },
        "richMessageStrings": {
          "default": "This is _the_ default     "markdown": "Another message, for
**this** rule.",
            "*special": "This is another message for this rule, used in
_special_*    cases."
              }
            }
          }
    }    ]
    }
  },
  "results": [
    {                                # A result object (§}
```

~~If the message string identified by `ruleId` and `ruleMessageId` includes placeholders (§), then `result.message.arguments` (§, §) **SHALL** contain the replacement values for the placeholders. In this situation, `result.message` will contain *only* the `arguments` property.~~

> ~~EXAMPLE 2: In this example, the message string identified by `ruleId` and `ruleMessageId` has a single placeholder `"{0}"`. `message.arguments` holds the replacement value `"counter"`.~~

```
{                                 # A run object (§).
  "results": [
3.27    {                              # A result object (§).
).
       "ruleId": "CA2101",
       "ruleMessageIdrule": {
         "index": 0
       },
       "message": {
         "id": "default",","
       "message": {},
         "arguments": [...
           "counter"}
   ]
      }
    }
  ],

  "resources": {
    "rules": {
      "CA2101": {                   # A rule object
        "messageStrings": {
          "default": "Variable \"{0}\" is uninitialized."
        }
      }
    }
  }
}
```

### ~~3.20.10~~3.27.12 locations property

A `result` object **SHOULD** contain a property named `locations` whose value is an array of ~~one~~zero or more ~~unique (§)~~ `location` objects (§3.28~~),~~) each of which specifies a location where the result occurred.

> NOTE 1: In rare circumstances, it might not be possible to specify a location for a result. However, the `locations` property contains 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.

> NOTE 2: The `locations` array is not defined to contain unique (§3.7.3) elements because some tools report a line number but not a column number for a result's location. Such a tool might report the same result twice on the same line, in some cases producing multiple identical `location` objects.

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~~supports "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 might contain a single result object, which would 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.

EXAMPLE 4: In contrast, consider a tool which locates misspelled words in variable names. If the tool detects a misspelled variable name, it ~~must~~might 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.

### ~~3.20.11~~3.27.13 analysisTarget property

If the analysis target differs from the result file, a `result` object **SHOULD** contain a property named `analysisTarget` whose value is ~~a `fileLocation`~~an `artifactLocation` object (§3.4) that specifies the analysis target.

If the analysis target and the result file are the same, the `analysisTarget` property **SHOULD** be absent.

EXAMPLE: In this example, the tool's analysis target was the file mouse.c. ~~In the course of~~During the scan, the tool detected a result in the included file mouse.h.

```
{                                  # A result object (§3.27).
  "analysisTarget": {              # A fileLocationAn artifactLocation object
(§3.4).
    "uri": "input/mouse.c",
    "uriBaseId": "SRCROOT"
  },

  "locations": [                   # See §3.27.12.
    {                              # A location object (§3.28).
      "physicalLocation": {        # See §3.28.3.
        "fileLocation": {          # A fileLocationartifactLocation": {      #
An artifactLocation object.
          "uri": "input/mouse.h",
          "uriBaseId": "SRCROOT"
        },

        "region": {
          "startLine": 42
        }
      }
    }
  ]
}
```

### 3.27.14 webRequest property

A `result` object **MAY** contain a property named `webRequest` whose value is a `webRequest` object (§3.46) that describes the HTTP request which led to this result.

NOTE: This property is primarily useful to web analysis tools.

## 3.27.15 webResponse property

A `result` object **MAY** contain a property named `webResponse` whose value is a `webResponse` object (§3.47) that describes the response to the HTTP request which led to this result.

> NOTE: This property is primarily useful to web analysis tools.

## ~~3.20.12~~3.27.16 fingerprints property

A `result` object **MAY** contain a property named `fingerprints` whose value is ~~a JSON~~an object (§3.6).

Each property value in this object **SHALL** be a string that provides a stable identifier for the result. This identifier **SHALL**, to the extent that it is feasible, be the same for all results that are logically identical, and different for any two results that are logically distinct. This requirement is intended to ensure that a fingerprint is resistant to changes that do not affect the logical identity of the result, such as the location of the root of a source code enlistment, or the line number where a result appears in a source file.

Each property name in this object **SHALL** be a versioned hierarchical string (§3.5.4.2). A result management system **MAY** use the property names to identify the method used to calculate the fingerprint.

> EXAMPLE 1: In this example, the producer has calculated a fingerprint using version 2 of a fingerprinting method it refers to as "~~contextRegionHash~~stableResultHash":

```
{
    "fingerprints": {
      "contextRegionHashstableResultHash/v2": "097886bc876fe"
    }
}
```

When a result management system uses fingerprint information to determine whether two results are logically identical, it **SHOULD** use the latest version of the fingerprint available in both results.

> EXAMPLE 2: In this example, one result has values for versions 1 and 2 of the “context region hash” fingerprint. Another result has values for versions 2 and 3. A result management system would use version 2 (the greatest common version) to compare the two results.

```
{                                   # A run object (§3.14).
  "results": [                      # See §3.14.23.
    {                               # A result object (§).
      "fingerprints": {
        "contextRegionHashstableResultHash/v1": "1234567900abc"
        "contextRegionHashstableResultHash/v2": "234567900abcd"
    },
    {
      "fingerprints": {
        "contextRegionHashstableResultHash/v2": "234567900abcd"
        "contextRegionHashstableResultHash/v3": "34567900abcde"
    }
  ]
}
```

> NOTE: This property is an array, rather than a single string, ~~to~~for two reasons:
>
> - To allow a result management system to ~~select among~~continue to support outdated fingerprinting algorithms while upgrading to a ~~variety of methods for deciding whether two results are logically identical or logically distinct~~newer, more reliable algorithm.

- Less likely but possible, to allow multiple result management systems to record their final fingerprints.

A direct SARIF producer **SHOULD NOT** populate this property. A SARIF converter **MAY** populate this property if the analysis tool's native output format provides a value that qualifies as a fingerprint (a stable identifier for the result). A result management system **MAY** populate this property when it ingests a SARIF file. If it does so, then later, when a SARIF consumer retrieves results in SARIF format from the result management system, the result management system **MAY** set this property to the value it assigned.

Appendix B provides requirements for how a result management system computes fingerprints.

NOTE: `fingerprints` and `correlationGuid` (§3.27.4) provide two different ways for result management systems to associate results that are logically identical. See §3.27.2 for more information.

## ~~3.20.13~~3.27.17 partialFingerprints property

A `result` object **MAY** contain a property named `partialFingerprints` whose value is ~~a JSON~~an object (§3.6).

Each property value in this object **SHALL** be a string that contributes to the stable, unique identity, or "fingerprint," of the result (see §3.27.16~~).~~). Appendix B explains how a result management system can compute these fingerprints.

Each property name in this object **SHALL** be a versioned hierarchical string (§3.5.4.2). A SARIF producer **MAY** use the property name to identify the nature of the information used to compute the partial fingerprint.

EXAMPLE 1: In this example, the producer has calculated a partial fingerprint using version 3 of a partial fingerprint value it refers to as "`prohibitedWordHash`":

```
{                                  # A result object (§3.27).
  "partialFingerprints": {
    "prohibitedWordHash/v3": "097886bc876fe"
  }
}
```

When a result management system uses partial fingerprint information to determine whether two results are logically identical, it **SHOULD** use the latest version of the partial fingerprint available in both results.

EXAMPLE 2: In this example, one result has values for versions 1 and 2 of the "prohibited word hash" partial fingerprint. Another result has values for versions 2 and 3. A result management system would use version 2 (the greatest common version) to compare the two results.

```
{                                  # A run object (§3.14).
  "results": [                     # See §3.14.23.
    {                              # A result object (§).
      "partialFingerprints": {
        "prohibitedWordHash/v1": "1234567900abc"
        "prohibitedWordHash/v2": "234567900abcd"
    },
    {
      "partialFingerprints": {
        "prohibitedWordHash/v2": "234567900abcd"
```

  

```
        "prohibitedWordHash/v3": "34567900abcde"
    }
  ]
}
```

A result management system **MAY** use any algorithm to combine the information contained in the various partial fingerprints. (For example, it might decide that two results are logically identically if any one of their partial fingerprints match, or only if a majority of them match, or only if all of them match.)

To make use of the information, if any, embodied in the property names, a result management system requires knowledge of the naming convention used by the SARIF producer. A result management system with that knowledge **MAY** use the property names to decide which partial fingerprints to include in its fingerprint computation. A result management system lacking that knowledge ~~**SHALL** include all~~**SHOULD NOT** attempt to interpret the information embodied in the partial ~~fingerprints in its~~ fingerprint ~~computation~~names.

Because result management systems might come to depend on the choice of property names, SARIF producers that use property names to identify the nature of the information used to compute the partial fingerprint **SHOULD** adhere to the following guidelines:

- Choose meaningful property names that describe the information used to compute the partial fingerprint.
- Document the property names.
- When introducing a partial fingerprint computed with a different approach, associate it with a new property name.
- Avoid removing existing property names and partial fingerprints, since existing result management systems might rely on them.

EXAMPLE ~~1~~3: In this example, a SARIF-producing document checker has computed ~~two~~a partial ~~fingerprints, one being a hash of~~fingerprint that hashes a word that should not appear in a document~~, and the other being a hash of~~ together with the document's language.

```
{                              # A result object.
  ...
  "partialFingerprints": {
    "wordHash": "wordPlusLangHash":
      "2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae","
    "langHash": "5c49f88dafe66e0ecdca8f682ae0b38c38ccd3ad464e3358e899beca88c18560"
  }
}
```

EXAMPLE ~~2~~4. In this example, the SARIF producer has ~~computed a single partial fingerprint. It has~~ chosen an arbitrary value for the ~~corresponding~~ property name.

```
{                              # A result object
  ...
  "partialFingerprints": {
    "1": "56eaf900cc8f6"
  }
}
```

## ~~3.20.14~~3.27.18 codeFlows property

A `result` object **MAY** contain a property named `codeFlows` whose value is an array of ~~one~~zero or more ~~unique (§)~~ `codeFlow` objects (§3.36). The `codeFlows` property is intended for use by analysis tools that provide execution path details that illustrate a possible problem in the code.

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

  

### ~~3.20.15~~3.27.19 graphs property

A `result` object **MAY** contain a property named `graphs` whose value is an array of ~~one~~zero or more unique (§3.7.3) `graph` objects (§3.39~~) each of which~~). A `graph` object represents a directed graph~~. A directed graph is~~: a network of nodes and directed edges that describes some aspect of the structure of the code (for example, a call graph).

A `graph` object defined at the `result` level **SHALL** be referenced only by `graphTraversal` objects (§3.42) defined in the `graphTraversals` property (§3.27.20) of the `result` object in which it is defined. This contrasts with `graph` objects defined at the `run` level (§3.14.20), which **MAY** be referenced by `graphTraversal` objects defined in the `graphTraversals` property of any `result` object in ~~the containing `run`~~`theRun`.

### ~~3.20.16~~3.27.20 graphTraversals property

If a `result` object contains a `graphs` property (§3.27.19), or if ~~its containing `run` object (§)~~`theRun` contains a `graphs` property (§3.14.20), then the `result` object **MAY** contain a property named `graphTraversals` whose value is an array of ~~one~~zero or more unique (§3.7.3) `graphTraversal` objects (§3.42). If neither the `result` object nor ~~its containing `run` object~~`theRun` contains a `graphs` property, the `graphTraversals` property **SHALL** be absent. A graph traversal is a path through the code that visits one or more nodes in a specified graph.

### ~~3.20.17~~3.27.21 stacks property

A `result` object **MAY** contain a property named `stacks` whose value is an array of ~~one~~zero or more unique (§3.7.3) `stack` objects (§3.44). The `stacks` property is intended for use by analysis tools that ~~collects~~compute or collect 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.

### ~~3.20.18~~3.27.22 relatedLocations property

A `result` object **MAY** contain a property named `relatedLocations` whose value is an array of ~~one~~zero or more unique (§3.7.3) `location` objects (§3.28~~),~~) 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": {
      "text": "Name 'index' cannot be used in this scope because
               it would give a different meaning to 'index'~~."~~
               ([declared here](0))."
    },
    "locations": [
      {
        "physicalLocation": {
```

  

```
          "uri": "file:///C:/Code/a.js",
          "region": {
            "startLine": "6",
            "startColumn": "10"
          }
        }
      }
    ],
    "relatedLocations": [    # An array of location objects
                             # (§3.28)
      {                      # A location object.
        "id": 0,
        "message": {
          "text": "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.
```

## 3.20.193.27.23 suppressionStatessuppressions property

### 3.20.201.1.1 General

A `result` object **MAY** contain a property named `suppressionStates``suppressions` whose value is an array of zero or more unique (§3.7.3) `suppression` objects (§3.35) strings. This property **SHALL** be present if and only if the analysis tool ) each of which describes a request to "suppress" a result (that produced the log file wishes to convey the is, to exclude it from result lists, bug counts, *etc.*).

If `suppressions` is absent, it **SHALL** default to `null`.

The presence of an array value, whether or not the array is empty, **SHALL** mean that suppression information that is available for the result. In this case, if the array is empty, a consumer **SHALL** treat the condition described by the result object should be "as not suppressed. If the array is non-empty, a consumer that needs to determine the result's suppression state **SHALL** examine the `status` properties (§3.35.3".) of the `suppression` objects in the array.

> 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 absence of an array value, or the reason or reasonspresence of a `null` value, **SHALL** mean that suppression information is not available for the result has been. A SARIF consumer **SHALL** treat such a result as not suppressed. The supported reasons for suppressing a result are:

  

The ~~developer has suppressed the~~suppressions values for all result objects in ~~the source code (see §).~~theRun **SHALL** be either all null or all non-null.

- ~~•~~ NOTE: The ~~result~~rationale is ~~marked as suppressed in an external store such as a database (see §).~~

#### ~~3.20.20.1 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~~an engineering system will generally evaluate all results for suppression, or none of them. Requiring 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,~~suppressions values be either all null or ~~if the tool cannot or chooses not~~all non-null enables a consumer 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"`.~~

#### ~~3.20.20.2 suppressedExternally value~~

~~Some development environments provide a persistent store, for example a database, containing historical information about the results from 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"`.~~ is available for the run by examining a single result object.

### ~~3.20.21~~3.27.24 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, which we refer to as the "baseline run."

If ~~`baselineInstanceGuid`~~`theRun.baselineGuid` (§3.14.5) is present~~ on the containing `run` object (§),~~, its value **SHALL** specify the baseline run.

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 run.
- `"~~existing~~unchanged"`: This result was detected both in the current run and in the baseline run, and it did not change between those two runs in any way that the tool considers significant.
- `"updated"`: This result was detected both in the current run and in the baseline run, but it changed between those two runs in a way that the tool considers significant.
- `"absent"`: This result was detected in the baseline run but was not detected in the current run.

~~If `baselineInstanceGuid` is present but `baselineState` is absent, `baselineState` **SHALL** be considered to have the value `"new"`.~~

> NOTE 1: The purpose of `baselineState` 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 needs a way to determine whether a result is logically "the same", in some sense, as a result that appeared in the baseline. Appendix B discusses how a result management system can assign a "fingerprint" to each result. See also the description of the fingerprints (§3.27.16) and partialFingerprints (§3.27.17) properties.

An analysis tool that works together with such a result management system can use the fingerprint to determine whether two results are logically the same; two results with the same fingerprint are considered logically the same.

NOTE 2: A result management system might respond to a "new" result by filing an issue in a bug tracking system. It might respond to an "updated" result by editing the details of an existing issue in the bug tracking system, or by attaching an updated SARIF log to the issue. It might respond to an "absent" result by resolving the issue. It might take no action at all for an "unchanged" issue, or it might simply update its internal information about the range of runs that contained the result.

If baselineState is present on any result object in theRun, it **SHALL** be present on every such result object.

NOTE 3: The presence of baselineState on any result implies that the SARIF producer performed a comprehensive comparison between the results in the current run and those in some previous run. A SARIF consumer is entitled to expect that the differencing operation produced a baselineState value for every result.

This is conceptually similar to a tool that compares two text files, and for every line, concludes that it exists in the left-hand file, the right-hand file, or both. The tool must provide this information for every line in both files; it cannot leave some lines "undetermined."

## 3.27.25 rank property

A result object **MAY** contain a property named rank whose value is a number between 0.0 and 100.0 inclusive, representing the priority or importance of the result. 0.0 is the lowest priority and 100.0 is the highest.

rank is only meaningful if kind (§3.27.9) has the value "fail".

If kind has the value "fail", then if rank is absent, it **SHALL** default to the value determined by the procedure defined for level (§3.27.10), except throughout the procedure, replace "level" with "rank" and replace "warning" with -1.0.

If kind has any other value, then rank **SHALL** be absent.

If rank is absent, it **SHALL** default to -1.0, which indicates that the value is unknown (not set).

NOTE: rank values produced by different tools are in general not commensurable. If Tool A produces one result with rank 0.65 and a second result with rank 0.70, the consumer is entitled to assume that the second result is of higher priority than the first. But if Tool A produces a result with rank 0.65 and Tool B produces a result with rank 0.70, the result produced by Tool B might or might not be of higher priority than the result produced by Tool A. In an engineering system that aggregates results from multiple tools, rank values might need to be adjusted, either automatically or by end users, so that rank values from different tools can be interleaved in a meaningful way.

## ~~3.20.22~~3.27.26 attachments property

A result object **MAY** contain a property named attachments whose value is an array of ~~one~~zero or more unique (§3.7.3) attachment objects (§3.21~~). Each attachment object **SHALL** describe a file~~) each of which describes an artifact relevant to the detection of the result.

~~For an example, see EXAMPLE 2 in §.~~

  

## ~~3.20.23~~3.27.27 workItemUris property

A `result` object **MAY** contain a property named `workItemUris` whose value is either `null` or an array of ~~one~~zero or more unique (§3.7.3) strings~~,~~ each ~~containing~~of which contains the absolute URI [RFC3986] of a work item associated with this result.

~~NOTE~~If `workItemUris` is absent, it **SHALL** default to `null`.

An empty array **SHALL** mean that there are no work items associated with this result. `null` **SHALL** mean that the set of work items associated with this result, if any, is not known.

The `workItemUris` values for all `result` objects in `theRun` **SHALL** be either all `null` or all non-`null`.

> NOTE 1: The rationale is that an engineering system will generally track work item status for all results or for none of them. Requiring that the `workItemUris` values be either all `null` or all non-`null` enables a consumer to determine whether work item information is available for the run by examining a single `result` object.

> NOTE 2: Result management systems are likely to generate work items from at least some of the results in a SARIF log file. Depending on the engineering system, these work items might take the form of Git issues, Jira tickets, TFS work items, or the equivalent in other work item tracking systems.

## ~~3.20.24~~3.27.28 ~~conversionProvenance~~hostedViewerUri property

~~Some analysis tools produce output files that describe the analysis run as a whole; we refer to these as "per-run" files. Other tools produce one or more output files for each result; we refer to these as "per-result" files. Some tools produce both per-run and per-result files.~~

~~If the `run` object (§) containing this `result` object was produced by a converter, and if the analysis tool whose output was converted to SARIF produced any per-result files for this result, then the~~ A `result` object **MAY** contain a property named ~~`conversionProvenance`~~`hostedViewerUri` whose value is a string containing an absolute URI [RFC3986~~array of one or more unique (§) `physicalLocation` objects (§)~~] at which ~~specify the relevant portions of those files.~~

~~Direct producers~~the result can be viewed. The URI **SHALL** ~~**NOT** emit the `conversionProvenance` property.~~be valid as of the time the tool generated this result. It is not guaranteed to be valid at later times (for example, the hosting environment might not keep results older than a specified age).

~~Per-run files are handled by the `conversion.analysisToolLogFiles` property (§).~~

> ~~NOTE: This property is intended to be useful to developers of converters, to help them debug the conversion from the analysis tool's native output format to the SARIF format.~~

> ~~EXAMPLE: Given this Android Studio output file:~~

```
<?xml version="1.0" encoding="UTF-8"?>
<problems>
  <problem>
    <file></file>
    <line>242</line>
    ...
    <problem_class ...>Assertions</problem_class>
    ...
    <description>Assertions are unreliable. ...</description>
  </problem>
</problems>
```

> ~~a SARIF converter might transform it into the following SARIF log file:~~

```
{
  ...
  "runs": [
    {
```

```
        "tool": {
          "name": "AndroidStudio",
          ...
        },
        "conversion": {  # A conversion object (see §)
          ...
        },
        "results": [
          {
            "ruleId": "Assertions",
            "message": {
              "text": "Assertions are unreliable. ..."
            },
            ...
            "conversionProvenance": [      # An array of physicalLocation objects (§).
              {
                "fileLocation": {                     # See §.
                  "uri": "AndroidStudio.log",
                  "uriBaseId": "$LOGSROOT"
                },
                "region": {                           # See §.
                  "startLine": 3,
                  "startColumn": 3,
                  "endLine": 12,
                  "endColumn": 13
                  "snippet": {
                    "text": "<problem>\n ... \n  </problem>",
                  }
                }
              }
            ],
            ...
          }
        ]
      }
    ]
}
```

## 3.20.251.1.1 fixes property

NOTE: This property can be used by tools that provide an online viewing experience for the results they generate. This experience might be specifically designed to display the results from that tool, as opposed to a generic SARIF viewer that displays results from any tool that produces SARIF.

## 3.27.29 provenance property

A `result` object **MAY** contain a property named `provenance` whose value is a `resultProvenance` object (§3.48) that contains information about how and when the result was detected.

## 3.27.30 fixes property

namesA `result` object **MAY** contain a property named `fixes` whose value is an array of onezero or more unique (§3.7.3) `fix` objects (§3.55).

## 3.20.263.27.31 propertiesoccurrenceCount property

A `result` object **MAY** contain a property named `propertiesoccurrenceCount` whose value is a positive integer specifying the number of times a result with `theResult.correlationGuid` (§3.27.4) has been observed.

NOTE: This property is intended for the scenario where multiple SARIF files are being merged into a single SARIF file, with the intent that each logically distinct result (see

  

§3.27.2bag (§). This allows tools to include information about the ) occurs only once in the merged file. In that case, the system performing the merge would select one occurrence of each logically distinct result that is not explicitly specified in the SARIF formatto serve as the exemplar for that class of results, and it would set `occurrenceCount` on that instance to the number of times a result with that `correlationGuid` occurred in the input files.

This property can also be useful even in the context of a single log file. Consider an accessibility checker that detects an accessibility problem at a particular location. Suppose the checker has access to activity logs that trace user paths through the application. The checker could use those logs to determine how many times users encountered the location with the accessibility problem, and store that information in `occurrenceCount`.

## 3.213.28 location object

### 3.28.1 General

### 3.21.1 General

A `location` object describes a location. Depending on the circumstances, a `location` object is described by physical location (§3.29), a logical location (§3.33), both, or in rare circumstances, neither (see below).

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 Among the reasons to includefor including logical locations in the SARIF format in addition to physical locations: are the following:

- 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 for more information about fingerprinting. The `fullyQualifiedLogicalName` property (§) is particularly convenient for fingerprinting.

- Logical location information is an important contributor to fingerprinting scenarios because it is typically more resilient to changes in source code than are the line numbers included in physical locations. See Appendix B for more information about fingerprinting. The `logicalLocation.fullyQualifiedName` property (§3.33.5) is particularly convenient for fingerprinting.

- In the analysis of structured data files such as XML or JSON, internal structural information (such as an XML path like `"/orders[2]/customers/lastName"`) might be helpful.

In rare circumstances, theirthere might be neither physical nor logical location information available for a `location` object. See §0 for an example. In that case, the location object **SHOULD** contain a message property (§3.28.5) explaining the significance of this “location.”

  

## 3.28.2 id property

A `location` object **MAY** contain a property named `id` whose value is a non-negative integer that is unique among all `location` objects belonging to `theResult`. The value does not need to be unique across all `result` objects (§3.27) in `theRun`.

If `id` is absent, it **SHALL** default to -1, which indicates that the value is unknown (not set).

> NOTE: Negative values are forbidden because their use would suggest some non-obvious semantic difference between positive and negative values.

> EXAMPLE: Within a `result` object, the following property values (among others) are `location` objects, and no two of them can have the same value for `id`:

```
result.relatedLocations[0]
result.codeFlows[0].threadFlows[0].locations[0].location
result.stacks[0].frames[0].location
```

The `id` property has two purposes: to enable an embedded link (§3.11.6) within a `message` object (§3.11) to refer to `thisObject`, and to identify `thisObject` as the target of a `locationRelationship` (§3.34). If no `message` object within `theResult` refers to `thisObject` *via* an embedded link and no `locationRelationship` object within `theResult` specifies `thisObject` as its target, the `id` property does not need to appear.

## 3.21.23.28.3 physicalLocation property

If physical location information is availableDepending on the circumstances, a `location` object either **SHALL**, **MAY**, or **SHALL NOT** contain a property named `physicalLocation` whose value is a `physicalLocation` object (§3.29) that identifies the file within which the location lies. If physical location information is available and the `logicalLocations` property (§3.28.4) is absent or empty, `physicalLocation` **SHALL** be present. If physical location is available and `logicalLocations` is present and non-empty, `physicalLocation` **MAY** be present. If physical location information is not available, `physicalLocation` **SHALL** **NOT** be absentpresent.

## 3.21.33.28.4 fullyQualifiedLogicalName propertylogicalLocations property

Depending on the circumstances, a `location` object either **SHOULDSHALL**, **MAY**, or **MAYSHALL NOT** contain a property named `fullyQualifiedLogicalName` `logicalLocations` whose value is an array of zero or more unique (§3.7.3) `logicalLocation` objects (§3.33a string) that identify the programmatic construct within which specifies the fully qualified name of the location lies. If logical location information is available and the `physicalLocation` property (§3.28.3. If physical ) is absent, `logicalLocations` **SHALL** be present and non-empty. If logical location information is available and `physicalLocation` is present, `logicalLocations` **MAY** be present. If logical location information is not available, `fullyQualifiedLogicalName` **SHOULD** be present. Otherwise, it **MAY**`logicalLocations` **SHALL NOT** be present.

The format of `fullyQualifiedLogicalName` **SHALL** follow the naming rules for fully qualified logical locations described in §.

> EXAMPLE 1: C: `create_process`

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

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

> If the NOTE: `logicalLocations` is an array because some logical locations can be expressed in more than one way. For example, the logical location of an element in an HTML document might be expressed by an XML Path expression such as `/html/body/img[1]` or by a CSS selector such as `#logo`.

  

### 3.28.5 message property

A locationlogicalLocations property (§) of the containing run object (§) is present, fullyQualifiedLogicalName **SHALL** equal the name of one of the properties on that logicalLocations object.

If during a run a tool produces results in two or more distinct logical locations with the same fully qualified logical name, and if the containing run object contain a logicalLocations property (§), then instead of containing the fully qualified logical name, fullyQualifiedLogicalName **SHALL** contain a string that equals one of the property names in run.logicalLocations. To improve the readability of the log file, this property name **SHOULD** be formed by appending a suffix to the fully qualified logical names. In this case, the fullyQualifiedName property (§) of the logicalLocation object (§) **SHALL** contain the actual fully qualified logical name.

NOTE: This is an extremely rare corner case.

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 fullyQualifiedLogicalName property in one of the results to A::B-1, and it might populate run.logicalLocations as follows:

```
"logicalLocations": {
  "A::B": {
    "name": "B",                        # Must specify because it differs from property name.
    "kind": "namespace",                # But fullyQualifiedName matches, so can be omitted.
    "parentKey": "A"
  },
  "A": {                                # Both name and fullyQualifiedName match property
    "kind": "namespace"                 # name, so can be omitted.
  },
  "A::B-1": {
    "name": "B",                        # Must specify because it differs from  object MAY
contain a property name.
    "fullyQualifiedName": "A::B",       # Must specify because it differs from property name.
    "kind": "type",
    "parentKey": "A-1"
  },
  "A-1": {
    "name": "A",                        # Must specify because it differs from property name.
    "fullyQualifiedName": "A"           # Must specify because it differs from property name.
     "kind": "namespace"
  }
}
```

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

- run.logicalLocations might not be present.
- It allows a SARIF viewer to display the logical location in a way that is easily understood by users.
- As mentioned in §, 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").

### 3.21.4 message property

A location object MAY contain a property named message whose value is a message object (§3.11) relevant to the location.

### 3.21.53.28.6 annotations property

A location object **MAY** contain a property named annotations whose value is an array of onezero or more unique (§3.7.3) region objects (§3.30),) each of which describes a region within the fileartifact specified by the location object that is relevant to the location. . Each of these region objects **SHOULD** contain a message property (§3.30.14) that explains the relevance of the region to the location.

EXAMPLE: Consider a location object which describes the declaration statement

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

If the analysis tool wanted to emphasize the expression (y + z), it might set the annotations property to:

```
"annotations": [                  # An array of region objects.
  {                               # A region object (§3.30).
    "startLine": 12,
    "startColumn": 139,
    "endColumn": 1916,
    "message": {
      "text": "(y + z) = 42"
    }
  }
]
```

### 3.21.63.28.7 propertiesrelationships property

A location object **MAY** contain a property named propertiesrelationships whose value is a property bag an array of zero or more unique (§3.7.3) locationRelationship objects (§3.34). This allows tools) each of which declares one or more directed relationship from thisObject to include information about theanother location object, which we refer to as theTarget, specified by locationRelationship.target (§3.34.2). The natures of the relationships between thisObject and theTarget are specified by locationRelationship.kinds (§3.34.3that is not explicitly specified in the SARIF format.).

## 3.223.29 physicalLocation object

### 3.22.13.29.1 General

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

### 3.29.2 Constraints

Either the artifactLocation property (§3.29.3), the address property (§3.29.6), or both **SHALL** be present.

If region.byteLength (§3.29.4, §3.30.12) and address.length (§3.29.6, §3.32.9id) are both present, then region.byteLength **SHALL** equal the absolute value of address.length.

### 3.22.23.29.3 artifactLocation property

A physicalLocation object **MAY** contain a property named idartifactLocation whose value is a non-negative integer that **SHALL** be unique among all physicalLocation objects belonging to the containing result object (§). The value does not need to be unique across all result objects in the run.

> EXAMPLE: Within a resultan artifactLocation object, the following property values (among others) are physicalLocation objects, and no two of them can have the same values for their id properties:

```
result.relatedLocations[0].physicalLocation
result.codeFlows[0].threadFlows[0].locations[0].physicalLocation
result.stacks[0].frames[0].physicalLocation
```

The purpose of the id property is to enable an embedded link (§) within a message object (§) to refer to the location. If no message object within the containing result object refers to this location *via* an embedded link, the id property does not need to appear.

### 3.22.3 fileLocation property

A physicalLocation object **SHALL** contain a property named fileLocation whose value is a fileLocation object (§ (§3.4) that represents the location of the artifact. If artifactLocation is absent, then address (§3.29.6file) **SHALL** be present.

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

> EXAMPLE: In this example, results[0].locations[0].physicalLocation.fileLocation.uri equals the name of the property files[0][file:///C:/Code/main.c].

```
{                                    # A run object (§).
  "files": {
    "file:///C:/Code/main.c": [
      {
        "mimeType": "text/x-c",
      }
    ]
  },
  "results": [
    {
      "ruleId": "CA2101",
      "level": "error",
      "locations": [
        {
          "physicalLocation": {
            "fileLocation": {
```

  

```
                "uri": "file:///C:/Code/main.c"
              },
              "region: {
                "startLine": 24,
                "startColumn": 9
              }
            }
          }
        }
      }
    }
  }
```

### 3.22.43.29.4 region property

A `physicalLocation` object **MAY** contain a property named `region` whose value is a `region` object (§3.30) that represents a relevant portion of the fileartifact. In particular, if the `physicalLocation` object occurs within the `locations` property (§3.27.12) of a `result` object (§3.27), the region property **SHALL** specify the region within the fileartifact where the result was detected.

> EXAMPLE 1: In this example, a `physicalLocation` object specifies the location where a result was detected. Its `region` property specifies the portion of the file where the result was detected.

```
{                               # A result object (§3.27).
  "locations": [                # See §3.27.12.
    {                           # A location object (§3.28).
      "physicalLocation": {     # See §3.28.3.
        "fileLocation": {               artifactLocation": {  # A
physicalLocationartifactLocation object.
          "uri": "ui/window.c",
          "uriBaseId": "SRCROOT"
        },

        "region": {             # The region specifies the portion of the file
          "startLine": 42       # where the result was detected.
        }
      }
    }
  ]
}
```

If the `physicalLocation` object specifies a location in a nested fileartifact, then the `region` property **SHALL** specify the location with respect to the innermost nested fileartifact.

> EXAMPLE 2: 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.

If the `region` property is absent, the `physicalLocation` object refers to the entire fileartifact.

### 3.22.53.29.5 contextRegion property

If a `physicalLocation` object contains a `region` property (§3.29.4), it **MAY** also contain a property named `contextRegion` whose value is a `region` object (§3.30) which specifies a region that is a proper superset of the region specified by the `region` property. If the `region` property is absent, the `contextRegion` **SHALL** be absent.

> The purpose ofNOTE: `contextRegion` is to enableenables a viewer to provide visual context when displaying a portion of a filean artifact. It can also be used to improve result matching.

  

EXAMPLE In this example, an analysis tool detected a result on line 42. The tool provides additional context for SARIF viewers by specifying a range of content surrounding the result line.

```
{                                               # A result object (§3.27).
  "locations": [                                # See §3.27.12.
    {                                           # A location object (§3.28).
      "physicalLocation": {                # See §.
        "fileLocation": {                       # A physicalLocation object (§3.29.).
        "artifactLocation": {                   # An artifactLocation object (§3.4).
          "uri": "ui/window.c",
          "uriBaseId": "SRCROOT"
        },

        "region": {                             # See §3.29.4.
          "startLine": 42,
          "snippet": {
            "text": "int n = m + 1;"
          }
        },

        "contextRegion": {
          "startLine": 41,
          "endLine": 43,
          "snippet": {
            "text": "int m;\nint n = m + 1\n\n"
          }
        }
      }
    }
  ]
}
```

## 3.29.6 address property

A `physicalLocation` object **MAY** contain a property named address whose value is an `address` object (§3.32) that represents the physical or virtual address of this location. If `address` is absent, then `artifactLocation` (§3.29.3) **SHALL** be present.

## 3.23 3.30 region object

### 3.23.1 3.30.1 General

A `region` object represents a region, that is, a contiguous portion of a file an artifact.

The `region` object defines both "text properties" and "binary properties." The text properties represent a region as a contiguous range of zero or more characters (a "text region"). The binary properties represent a region as a contiguous range of zero or more bytes (a "binary region").

For regions in text files, a `region` object **SHOULD** contain text properties and **MAY** also contain binary properties. If `startLine` (§3.30.5) > 0 or `charOffset` (§3.30.10) >= 0, this `region` object **SHALL** define a text region. If `byteOffset` (§3.30.11both text properties and binary properties are present, they) >= 0, this `region` object **SHALL** define a binary region. If a `region` object defines both a text region and a binary region, the text region and the binary region **SHALL** specify the identical range of bytes in the fileartifact, as determined by the file'sartifact's character encoding.

For regions in text artifacts, a `region` object **SHOULD** define a text region and **MAY** also define a binary region; it **SHALL** define either a text region or a binary region or both.

For regions in binary filesartifacts, a region object **SHALL** containdefine a binary propertiesregion and **SHALL NOT** containdefine a text propertiesregion.

  

If any text properties are present, enough text properties **SHALL** be present to fully specify a text region (see §3.30.2). If any binary properties are present, then enough binary properties **SHALL** be present to fully specify a binary region (see §3.30.3).

### ~~3.23.2~~3.30.2 Text regions

NOTE 1: The examples in this section assume a text file with the following contents:

```
abcd\r\nefg\r\nhijk\r\nlmn\r\n
```

Breaking the lines for the sake of readability, the contents are:

```
abcd\r\n
efg\r\n
hijk\r\n
lmn\r\n
```

The file contains four lines, each of which ends with the two-character newline sequence `"\r\n"`, which is explicitly displayed for clarity.

The line number of the first line in a text ~~file~~artifact **SHALL** be 1. The column number of the first character in each line **SHALL** be 1. The character offset of the first character in the ~~file~~artifact **SHALL** be 0.

The values of text properties **SHALL NOT** depend on the presence or absence of a byte order mark (BOM) at the start of the ~~file~~artifact.

Column numbers are expressed in the measurement unit specified by ~~the~~ `theRun.columnKind` ~~property~~ (§3.14.27~~) of the containing `run` object (§~~).

A SARIF viewer **MAY** choose to present column numbers that match the visual offset of each character from the beginning of the line. These "visual" column numbers might not match the column numbers contained in the SARIF file.

NOTE 2: Such a mismatch might occur if, for example, the line contains a tab character, or an accented character represented by a base character plus a combining character.

A text ~~file's~~artifact's character encoding determines the number of bytes that represent each character, and therefore determines the range of bytes represented by a text region. A SARIF consumer **SHALL** consider ~~a file~~an artifact to have the encoding specified by ~~`file`~~`artifact.encoding` (§3.24.9), if present, or else by ~~`run.defaultFileEncoding`~~`theRun.defaultEncoding` (§3.14.24), if present. If neither is present, the consumer **MAY** use any heuristic or procedure to determine the encoding, including (for example) prompting the user.

NOTE ~~2~~3: If a consumer incorrectly determines ~~a file's~~an artifact's encoding, it might not display the ~~file~~artifact correctly. For example, when it attempts to highlight a region, it might highlight an incorrect range of characters.

A text region **MAY** be specified in ~~three~~two ways:

- By means of the "line/column" properties `startLine` (§3.30.5), `startColumn` (§3.30.6), `endLine` (§3.30.7), and `endColumn` (§3.30.8).
- By means of the "offset/length" properties `charOffset` (§3.30.9) and `charLength` (§3.30.10).
- ~~By a combination of line/column and offset/length properties. If properties from both sets are present, they **SHALL** be consistent, as described below.~~

A text region **SHALL** specify both its start (the location of its first character) and its end (the location of its last character).

~~The start of a text region **MAY** be specified by a combination of `startLine` and `startColumn`, or by `charOffset`, or both.~~

NOTE 4: The end of a text region ~~**MAY**~~does not have to be specified ~~by a combination of~~explicitly if the default values for `endLine` ~~and~~, `endColumn`, and/or ~~by~~ `charLength`~~, or both~~ correctly describe the region.

  

A text region does not include the character specified by `endColumn` (see §3.30.8.).

If `charOffset` is present, then either or both of `startLine` and `startColumn` **MAY** be absent. If either is absent, it **SHALL** be taken to have the value implied by `charOffset`. If either is present, it **SHALL** equal the value implied by `charOffset`.

EXAMPLE 1: The region

```
{ "charOffset": 8 }
```

is identical to thesefollowing regions (among others):

```
{ "charOffset": 8, "startLine": 2, "startColumn": 3 }
{ "charOffset": 8, "startLine": 2 }
{ "charOffset": 8, "startColumn": 3 }
```

The first character in each of those regions is ) all specify the "g" on line 2.range of characters "bc".

If `charOffset` is absent, then `startLine` **SHALL** be present. In that case, if `startColumn` is absent, it **SHALL** be taken to have the value 1. `charOffset` **SHALL** be taken to have the value implied by `startLine` and `startColumn`.

EXAMPLE 2: The region

```
{ "startLine": 2, "startColumn": 3 }
```

is identical to the region

```
{ "charOffset": 8, "startLine": 2, "startColumn": 3 }
```

and to all the other regions in EXAMPLE 1, among others.

```
{
  "startLine": 1,
  "startColumn": 2,
```

"EXAMPLE 3: The region

```
{ "startLine": 2 }
```

Is identical to these regions (among others):

```
{ "startLine": 2, "startColumn": 1 }
{ "startLine": 2, "startColumn": 1, "charOffset": 6 }
```

The first character in each of those regions is the "e" at the start of line 2.

If `charLength` is present, then either or both of `endLine` and `endColumn` **MAY** be absent. If either is absent, it **SHALL** be taken to have the value implied by `charLength`. It either is present, it **SHALL** have the value implied by `charLength`.

```
endLine": 1,
 EXAMPLE 4: The region{ "startLine": 1, "charLength": 14 }
```

includes the characters from the "a" on line 1 through the "j" on line 3. It is identical to these regions (among others):

```
{ "startLine": 1, "charLength": 14, "endLine": 3, "endColumn": 4 }     # The
region excludes the character at endColumn.
{}

{
  "charOffset": 1,
```

  

```
  "charLength": 2
}

{
  "startLine": 1, "charLength": 14,
  "startColumn": 2,
  "endLine": 3 }1,
 { "startLine": 1, "charLength": 14, "endColumn": 4 },
```

Note that the region does *not* include the character in column 4 or line 3 (the "k").

If charLength is absent then if endLine is absent, endLine **SHALL** be taken to have the same value as startLine (whose value might, in turn, have been implied by charOffset). If endColumn is absent, it **SHALL** default to one greater than the number of characters on the last line of the region, excluding the newline sequence. charLength **SHALL** default to the value implied by endLine and endColumn.

EXAMPLE 5: The region

```
{ "startLine": 1, "startColumn": 2 }
includes the characters from   "charOffset": 1,
  "charLength": 2
}
```

EXAMPLE 2: The following region is invalid, even though it might appear to specify the same range of characters "bc" as in EXAMPLE 1:

```
{
  "startLine": 1,
  "charOffset": 1,    # Specifies the "b" on line 1 through
  "endColumn": 4      # Specifies the "d" atcolumn one past the end of line 1.
endColumn"c"
}
```

This is because the line/column properties and the offset/length properties, taken independently, specify different regions:

- "startColumn" is absent, and so defaults to 1 (see §3.30.65 (because there are 4 characters on the line, excluding the newline sequence). charLength).
- "endLine" is absent, and so defaults to "startLine", which in this example is 1 (see §3.30.73.).
- It"charLength" is identical,absent, and so defaults to 0 (see §3.30.10these regions (among others):).

{In summary, the above region is equivalent to the region

```
{
  "startLine": 1,
  "startColumn": 1,
  "startColumn": 2, "endLine": 1 },
 { "startLine": 1, "startColumn": 2, "endLine": 1, "endColumn": 5 }4,
{ "startLine": 1, "startColumn": 2, "endLine": 1,
  "charOffset": 1,
  "charLength": 3 }0
}
Now we can see that the line/column properties represent the range of
characters "abc", while the offset/length properties represent an insertion
point before the character "b" (see §3.30.10{ "startLine": 1, "startColumn":
2, "endColumn": 5, "charLength": 3 }
```

EXAMPLE 6: The region

  

```
{ "startLine": 2 }
```

includes the entire contents of line 2, excluding the newline sequence, namely "efg".

It is identical to these regions (among others):

```
{ "startLine": 2 }
{ "startLine": 2, "startColumn": 1 }
{ "startLine": 2 , "charLength": 3 }
{ "startLine": 2, "endColumn": 4 }
```

). Those two regions are not the same, and so the region is invalid.

If a region spans one or more than one linelines, it **SHALL** include the newline sequences of all but the last line in the region.

EXAMPLE 7: The region

NOTE 5: This is not an independent requirement; it is a consequence of the specification for the default value of `endColumn`.

EXAMPLE 3: The region

```
{ "startLine": 2 }
```

includes the characters "efg".

EXAMPLE 4: The region

```
{ "startLine": 2, "endLine": 3 }
```

includes the characters "efg\r\nhijk".

To specify an entire line together with its trailing newline sequence, specify the region's end point to be column 1 on the next line.

NOTE 6: This is again a consequence of the specification of `endColumn`, which states that it specifies the character one past the end of the region.

EXAMPLE 5: The region

```
{ "startLine": 2, "endLine": 3, "endColumn": 1 }
```

includes the characters "efg\r\n".

A region of length 0 is referred to as an "insertion point." An insertion point **MAY** be specified either by specifying `charLength` as 0, or by specifying the same values for `startColumn` and `endColumn`.

NOTE 3: This7: Once more, this is consistent withagain a consequence of the rule that a region does not include the character in columnspecification of `endColumn`.

EXAMPLE 86: These regions (among others) specify an insertion point before the "b" on line 1.

```
{ "startLine": 1, "startColumn": 2, "endColumn": 2 }
{ "startLinecharOffset": 1, "startColumn": 2, "charLength": 0 }
```

EXAMPLE 97: These regions (among others) specify an insertion point at the beginning of the file:

```
{ "startLine": 1, "startColumn": 1, "endColumn": 1 }
{ "startLine": 1, "startColumn": 1, "charLength{ "charOffset": 0 }
{ "startLine": 1, "charLength": 0 }
```

  

To specify an insertion point after the last character in ~~a file~~an artifact, set `endLine` to the number of the last line in the ~~file~~artifact, and set `endColumn` to a value one greater than the number of characters on the line, *including* any trailing newline sequence~~.~~.

> EXAMPLE ~~10~~8: These regions (among others) specify an insertion point at the very end of the file. Note that the last line contains the five characters (including the newline sequence) `"lmn\r\n"`.

```
{ "startLine": 4, "startColumn": 6, "endColumn": 6 }
{ "startLine": 4, "startColumn": 6charOffset": 22, "charLength": 0 }
```

## ~~3.23.3~~3.30.3 Binary regions

The byte offset of the first byte in ~~a file~~an artifact **SHALL** be 0.

To specify a byte region, at least `byteOffset` (§3.30.11) **SHALL** be present. `byteLength` (§3.30.12) **MAY** also be present. `byteOffset` specifies the start of the region. `byteLength` specifies the region's length and thereby, indirectly, its end~~ of the region~~. A `byteLength` value of 0 represents an insertion point before the byte specified by `byteOffset`.

## ~~3.23.4~~3.30.4 Independence of text and binary regions

The text-related and binary-related properties in a `region` object **SHALL** be treated independently. That is, the value of a text-related property **SHALL NOT** be inferred from the value of any set of binary-related properties, and *vice versa*.

> EXAMPLE: This example is based on the sample text file ~~show~~shown in NOTE 1 of §3.30.2. It represents invalid SARIF because the text-related and binary-related properties are inconsistent. At first glance they appear to be consistent because the byte at offset 2 is indeed on line 1:

```
{ "startLine": 1, "byteOffset": 2, "byteLength": 6 }
```

> However, because the default values for the missing text-related properties are determined entirely from the existing text-related properties, and independently of any binary-related properties, this region is in fact equivalent to this one:

```
{
  "startLine": 1,
  "startColumn": 1,  // Missing startColumn defaults to 1.
  "endLine": 1,      // Missing endLine defaults to startLine.
  "endColumn": 65,    // Missing endColumn defaults to (length of endLine +
1),
                     // exclusive of newline sequence.
  "byteOffset": 2
  "byteLength": 6
}
```

This makes it clear that the text-related and binary-related properties represent different ranges of bytes, and therefore the region is invalid.

## ~~3.23.5~~3.30.5 startLine property

When a `region` object represents a text region specified by line/column properties, it ~~**MAY**~~**SHALL** contain a property named `startLine` whose value is a positive integer equal to the line number of the line containing the first character in the region.

~~If `startLine` is absent, its value **SHALL** be inferred as specified in §.~~

  

### ~~3.23.6~~3.30.6 startColumn property

When a `region` object represents a text region specified by line/column properties, it **MAY** contain a property named `startColumn` whose value is a positive integer equal to the column number of the first character in the region.

If `startColumn` is absent, ~~its value~~it **SHALL** ~~be inferred as specified in §.~~default to 1.

### ~~3.23.7~~3.30.7 endLine property

When a `region` object represents a text region specified by line/column properties, it **MAY** contain a property named `endLine` whose value is a positive integer equal to the line number of the line containing the last character in the region.

If `endLine` is absent, its value **SHALL** ~~be inferred as specified in §.~~default to `startLine`.

### ~~3.23.8~~3.30.8 endColumn property

When a `region` object represents a text region specified by line/column properties, it **MAY** contain a property named `endColumn` whose value is an integer whose value is one greater than the column number of the last character in the region.

If `endColumn` is absent, ~~its~~ it **SHALL** default to a value ~~**SHALL** be inferred as specified in §.~~one greater than the column number of the last character on the line, excluding any newline sequence.

### ~~3.23.9~~3.30.9 charOffset property

When a `region` object represents a text region~~, it **MAY**~~ specified by offset/length properties, it **SHALL** contain a property named `charOffset` whose value is an integer equal to the zero-based character offset of the first character in the region from the beginning of the ~~file.~~

artifact. If `charOffset` is absent, it **SHALL** ~~be inferred as specified in §.~~default to -1, which indicates that the value is unknown (not set).

### ~~3.23.10~~3.30.10 charLength property

When a `region` object represents a text region specified by offset/length properties, it **MAY** contain a property named `charLength` whose value is a non-negative integer equal to the number of characters in the region.

If ~~the region consists of 0 characters (an insertion point), then either `charLength` **SHALL** be~~ `charLength` is absent, ~~or~~ it **SHALL** default to 0, which **SHALL** be interpreted as an insertion point at the position specified by `charOffset` (§3.30.9~~have the value 0.~~)

The sum of `charOffset` ~~(§)~~ and `charLength` **SHALL** be greater than or equal to 0 and less than or equal to the number of characters in the ~~file~~artifact.

A region whose `charOffset` is equal to the number of characters in the ~~file~~artifact and whose `charLength` is 0 is permitted and **SHALL** represent an insertion point at the end of the ~~file~~artifact.

### ~~3.23.11~~3.30.11 byteOffset property

When a `region` object represents a binary region, it **SHALL** contain a property named `byteOffset` whose value is an integer equal to the zero-based byte offset of the first byte in the region from the beginning of the ~~file.~~artifact. If `byteOffset` is absent, it **SHALL** default to -1, which indicates that the value is unknown (not set).

### ~~3.23.12~~3.30.12 byteLength property

When a `region` object represents a binary region, it **MAY** contain a property named `byteLength` whose value is an integer equal to the number of bytes in the region. If `byteLength` is absent, it **SHALL** default to 0, which **SHALL** be interpreted as an insertion point at the position specified by `byteOffset` (§3.30.11~~defaults to 0.~~).

The sum of `byteOffset` ~~(§)~~ and `byteLength` **SHALL** be greater than or equal to 0 and less than or equal to the number of bytes in the ~~file~~artifact.

A `region` object whose `byteOffset` equals the number of bytes in the ~~file~~artifact and whose `byteLength` is 0 is permitted, and **SHALL** represent an insertion point at the end of the ~~file~~artifact.

### ~~3.23.13~~3.30.13 snippet property

A `region` object **MAY** contain a property named `snippet` whose value is ~~a `fileContent`~~an `artifactContent` object (§3.3) representing the portion of the ~~file~~artifact specified by the `region` object.

> NOTE: The ~~purpose of the~~ `snippet` property ~~is to allow~~has various uses:
>
> - It allows a SARIF viewer to present the contents of the region even if the ~~file~~artifact from which it was taken is not available.
> - It also allows an end user examining a SARIF log file to see the relevant ~~file~~ content without opening another file.
> - It can be used to improve result matching.

### ~~3.23.14~~3.30.14 message property

A `region` object **MAY** contain a property named `message` whose value is a `message` object (§3.11) containing a message relevant to the region.

A SARIF viewer ~~**SHOULD**~~**MAY** display this message when the user interacts with the region. (For example, if the user hovers over the region with the mouse, the viewer might present the message as hover text~~.~~.)

### 3.30.15 sourceLanguage property

If the `region` object represents a portion of a text artifact that contains source code, it **MAY** contain a property named `sourceLanguage` whose value is a hierarchical string (§3.5.4) that specifies the programming language in which this portion of the source code is written. If the `region` object does not represent a portion of a text artifact containing source code, then `sourceLanguage` **SHALL** be absent.

For the remainder of this section, we assume that the `region` object represents a portion of a text artifact that contains source code.

> NOTE: This property is intended to help SARIF viewers to render code snippets (§3.30.13) with appropriate syntax coloring. It is intended for use in mixed-language files, such as HTML files that contain JavaScript™. For more information about this usage, see §3.24.10.

if `sourceLanguage` is absent, it **SHALL** default to the value of the `sourceLanguage` property (§3.24.10) of the `artifact` object (§3.24) which describes the artifact that contains the region. `artifact.sourceLanguage` in turn defaults to `theRun.defaultSourceLanguage` (§3.14.25). If all three of `region.sourceLanguage`, `artifact.sourceLanguage`, and `theRun.defaultSourceLanguage` are absent, the source language of the region object **SHALL** be taken to be unknown. In that case, a SARIF viewer **MAY** use any method or heuristic to determine the region's source language, for example, by examining the file's file name extension or MIME type, or by prompting the user.

For conventions and practices regarding the value of this property, see §3.24.10.2.

  

## ~~3.24~~3.31 rectangle object

### ~~3.24.1~~3.31.1 General

A `rectangle` object specifies a rectangular area within an image. When a SARIF viewer displays an image, it ~~**SHOULD**~~**MAY** indicate the presence of these areas, for example, by highlighting them or surrounding them with a border.

### ~~3.24.2~~3.31.2 top, left, bottom, and right properties

A `rectangle` object **SHALL** contain properties named `top`, `left`, `bottom`, and `right`, each of which contains a number (as defined by the JSON Schema standard [JSCHEMA01]) specifying one of the coordinates of the rectangle within the image. These properties **SHALL** be measured in the image format's natural units (for example, pixels for raster-based image formats). These values **MAY** be positive or negative, depending on the natural coordinate system of the image format. They **MAY** increase either from left to right or from right to left, and either from top to bottom or from bottom to top, again depending on the natural coordinate system of the image format.

> NOTE: A number in JSON schema can take a variety of forms, including simple integers (`42`) and floating-point numbers (`3.14`).

### ~~3.24.3~~3.31.3 message property

A `rectangle` object **SHOULD** contain a property named `message` whose value is a `message` object (§3.11) containing a message relevant to this area of the image.

A SARIF viewer ~~**SHOULD**~~**MAY** display this message when the user interacts with the area. For example, if the user hovers over the area with the mouse, the viewer might present the message as hover text.

## 3.32 address object

### 3.32.1 General

An `address` object describes a physical or virtual address, or a range of addresses, in an "addressable region" (memory or a binary file).

### 3.32.2 Parent-child relationships

`address` objects can be linked by their `parentIndex` properties (§3.32.13) to form a chain in which each address is specified as an offset from a "parent" object which we refer to as `theParent`.

> EXAMPLE: In this example, the location of the Sections region of a Windows ® Portable Executable file [PE] is expressed as an offset from the start of the module. The location of the .text section is in turn expressed as an offset from Sections.

```
{                                    # A run object (§3.14).
  "addresses": [                     # See §3.14.18.
    {
      "name": "Multitool.exe",       # See §3.32.10.
      "kind": "module",              # See §3.32.12.
      "absoluteAddress": 1024        # See §3.32.6.
    },
    {
      "name": "Sections",
      "kind": "header",
      "parentIndex": 0,              # See §3.32.13.
      "offsetFromParent": 376,       # See §3.32.8.
      "absoluteAddress": 1400,
      "relativeAddress": 376         # See §3.32.7.
    },
```

  

```
    {
      "name": ".text",
      "kind": "section",
      "parentIndex": 1,
      "offsetFromParent": 136,
      "absoluteAddress": 1536,
      "relativeAddress": 512
    }
  ],
  ...
}
```

### 3.32.3 Absolute address calculation

Each `address` object has an associated value called its "absolute address" which is the offset of the address from the start of the addressable region. The absolute address is calculated by executing the function `CalculateAbsoluteAddress` defined below on `thisObject` or by any procedure with the same result.

This procedure assumes that the `offsetFromParent` (§3.32.8) and `parentIndex` (§3.32.13) properties are either both present or both absent; if this is not the case, the SARIF file is invalid.

```
FUNCTION CalculateAbsoluteAddress(addr)
  IF addr.absoluteAddress exists THEN
    RETURN addr.absoluteAddress
  ELSE IF addr.parentIndex exists THEN
    LET theParent = the parent object (see §3.32.2) of addr
    RETURN addr.offsetFromParent + CalculateAbsoluteAddress(theParent)
  ELSE
    ERROR "Absolute address cannot be determined".
```

If `CalculateAbsoluteAddress(thisObject)` or any of its recursive invocations encounters an ERROR, the absolute address cannot be determined.

If both `absoluteAddress` and `offsetFromParent` exist, then `absoluteAddress` **SHALL** equal the value that `CalculateAbsoluteAddress` would have returned if `absoluteAddress` were absent, if `CalculateAbsoluteAddress` would have returned successfully in that circumstance.

### 3.32.4 Relative address calculation

Each `address` object has an associated value called its "relative address" which is the offset of the address from the address of the top-most object in its parent chain. The relative address is calculated by executing the function `CalculateRelativeAddress` defined below on `thisObject` or by any procedure with the same result.

This procedure assumes that the `offsetFromParent` (§3.32.8) and `parentIndex` (§3.32.13) properties are either both present or both absent; if this is not the case, the SARIF file is invalid.

```
FUNCTION CalculateRelativeAddress(addr)
  IF addr.relativeAddress exists THEN
    RETURN addr.relativeAddress
  ELSE IF addr.parentIndex exists THEN
    LET theParent = the parent object (see §3.32.2) of addr
```

  

    RETURN `addr.offsetFromParent` + `CalculateRelativeAddress(theParent)`
  ELSE
    RETURN 0

If `CalculateRelativeAddress(thisObject)` or any of its recursive invocations encounters an ERROR, the relative address cannot be determined.

If both `relativeAddress` and `offsetFromParent` exist, then `relativeAddress` **SHALL** equal the value that `CalculateRelativeAddress` would have returned if `relativeAddress` were absent, if `CalculateRelativeAddress` would have returned successfully in that circumstance.

### 3.32.5 index property

Depending on the circumstances, an `address` object either **MAY, SHALL NOT**, or **SHALL** contain a property named `index` whose value is the array index (§3.7.4) within `theRun.addresses` (§3.14.18) of an `address` object that provides the properties for `thisObject`. We refer to the object in `theRun.addresses` as the “cached object.”

If `thisObject` is an element of `theRun.addresses`, then `index` **MAY** be present. If present, its value **SHALL** be the index of `thisObject` within `theRun.addresses`.

Otherwise, if `theRun.addresses` is absent, or if it does not contain a cached object for `thisObject`, then `index` **SHALL NOT** be present.

Otherwise (that is, if `thisObject` belongs to a result, and `theRun.addresses` contains a cached object for `thisObject`), then `index` **SHALL** be present, and its value **SHALL** be the array index within `theRun.addresses` of the cached object.

If `index` is present, `thisObject` **SHALL** take all properties present on the cached object. If `thisObject` contains any properties other than `index`, they **SHALL** equal the corresponding properties of the cached object.

> NOTE 1: This allows a SARIF producer to reduce the size of the log file by reusing the same `address` object in multiple results.

> NOTE 2: For examples of the use of an `index` property to locate a cached object, see §3.38.2.

### 3.32.6 absoluteAddress property

An `address` object **MAY** contain a property named `absoluteAddress` whose value is a non-negative integer containing the absolute address (see §3.32.3) of `thisObject`.

If `absoluteAddress` is absent, it **SHALL** default to -1, which indicates that the value is unknown (not set).

### 3.32.7 relativeAddress property

If `parentIndex` (§3.32.13) is present, an `address` object **MAY** contain a property named `relativeAddress` whose value, if present, is an integer containing the relative address (see §3.32.4) of `thisObject`.

If `parentIndex` is absent, `relativeAddress` **SHALL** be absent.

If `relativeAddress` is absent, it **SHALL** default to `null`, which indicates that the value is unknown (not set).

  

### 3.32.8 offsetFromParent property

If `parentIndex` (§3.32.13) is present, an `address` object **MAY** contain a property named `offsetFromParent` whose value, if present, is an integer containing the offset of this address from the absolute address of `theParent` (see §3.32.2). This is the case even if the absolute address of the parent cannot be determined by the procedure in §3.32.3.

> NOTE 1: The rationale is that the absolute address always exists, even if the log file does not contain enough information to determine it, so it is always sensible to talk about an offset from that address.

If `parentIndex` is absent, `offsetFromParent` **SHALL** be absent.

If `offsetFromParent` is absent, it **SHALL** default to `null`, which indicates that the value is unknown (not set).

### 3.32.9 length property

An `address` object **MAY** contain a property named `length` whose value, if present, is an integer whose absolute value specifies the number of bytes in the range of addresses specified by this object.

A negative value for `length` **SHALL** mean that the data structure being described grows from higher addresses towards lower addresses (as, for example, is often the case for a stack).

If `length` is absent, it **SHALL** default to `null`, which indicates that the value is unknown (not set).

### 3.32.10 name property

An `address` object **MAY** contain a property named `name` whose value is a string containing the name of this address.

### 3.32.11 fullyQualifiedName property

An `address` object **MAY** contain a property named `fullyQualifiedName` whose value is a string containing the fully qualified name of this address.

> EXAMPLE: `"fullyQualifiedName": "MyDll.dll+0x47"`
>
> This name consists of two components. The first component is the name of the address at which the module was loaded into memory. The second component represents an offset from that address.

### 3.32.12 kind property

An `address` object **MAY** contain a property named `kind` whose value is a string that specifies the kind of addressable region in which this address is located.

When possible, SARIF producers **SHOULD** use the following values, with the specified meanings.

- `"data"`: An addressable location containing non-executable data.
- `"header"`: A data structure that precedes one or more addressable regions and specifies the layout and location of objects within the address space.
- `"function"`: An addressable region, possibly named, containing a sequence of instructions that perform a specified task.
- `"instruction"`: An addressable location containing executable code.
- `"page"`: An addressable region whose contents can be moved between primary and secondary storage.
- `"section"`: A named region of a file containing executable code or data, which in some circumstances is loaded into memory.
- `"segment"`: 1. A data structure in a binary that describes a region of memory, specifying its addressing and permissions information, as well as information about which sections are to be

  

loaded into the segment. 2. A region of memory whose contents are specified by the information in a segment defined in a binary, or by the operating system.

- "stack": An addressable region containing a call stack.
- "stackFrame": An addressable region containing a single frame from within a call stack.
- "module": The location at which a module was loaded.
- "table": An addressable region with a distinct purpose and a specified internal organization

The definitions of some of these "kind" values vary across operating systems. A SARIF producer **SHOULD** use the term most appropriate for the target operating system.

Although a function does contain executable code, the value "function" **SHOULD** be used for the address of the start of a function, because it is more specific. The value "instruction" **SHOULD** be used for an address within the body of a function.

If none of these values are appropriate, a SARIF producer **MAY** use any value.

### 3.32.13 parentIndex property

If theParent exists (that is, if thisObject is expressed as an offset from some other address), then an address object **SHALL** contain a property named parentIndex whose value is the array index (§3.7.4) of theParent within theRun.addresses (§3.14.18).

If theParent does not exist, then parentIndex **SHALL** be absent.

## ~~3.25~~3.33 logicalLocation object

### ~~3.25.1~~3.33.1 General

A logicalLocation object describes a logical location. A logical location is a location specified by a programmatic construct such as a namespace, a type, or a method, without regard to the physical location where the construct occurs.

logicalLocation objects occur in two places: as ~~property values within the~~array elements of run.logicalLocations ~~object~~ (§3.14.17) and as array elements of location.logicalLocations (§3.28.4).

### ~~3.25.2~~3.33.2 Logical location naming rules

Every logical location has a “fully qualified logical name” (more briefly, a “fully qualified name”) that fully specifies the programmatic construct to which it refers. When programmatic constructs are nested (such as a method within a class within a namespace), the fully qualified name is typically a hierarchical identifier such as "N.C.F(void)" or "N::C::F(void)". We refer to the rightmost component of this hierarchical identifier as the “logical name” (more briefly, the “name”) of the logical location.

~~Logical location names and fully qualified names appear in various properties in the SARIF format:~~

- ~~logicalLocation.name (§): a logical name.~~
- ~~logicalLocation.fullyQualifiedName (§): a fully qualified logical name.~~
- ~~location.fullyQualifiedLogicalName (§): a fully qualified logical name, with one rare exception (see §).~~
- ~~The property names in the object specified by run.logicalLocations (§): fully qualified logical names, with one rare exception (see §).~~

Whenever possible, logical names and fully qualified logical names ~~**SHALL**~~**SHOULD** conform to the syntax of the programming language in which the programmatic construct specified by the logical location was expressed.

EXAMPLE 1: The fully qualified logical name of the C++ method f(void) in class C in namespace N is "N::C::f(void)". Its logical name is "f(void)".

This is not always possible, for two reasons:

  

- For certain values of `logicalLocation.kind` (§3.33.7), there is no language syntax to specify the fully qualified name.

  EXAMPLE 2: Suppose the logical location is the local variable `pBuffer` in the C++ method `"N::C::f(void)"`. `logicalLocation.kind` is `"variable"`. There is no way to express the fully qualified name in C++. The SARIF producer might choose a fully qualified name such as `"N::C::f(void)?pBuffer"`.

- For other values of `logicalLocation.kind`, it is sometimes but not always possible to express the logical location in language syntax.

  EXAMPLE 3: Suppose the logical location is the anonymous callback function in this JavaScript™ function:

```
function click_it() {
  $("button").click(function(){
      alert("Clicked!");");
  });
}
```

`logicalLocation.kind` is `"function"`, for which it is sometimes possible to specify a fully qualified name. But there is no language syntax to express the name of an anonymous callback. The SARIF producer might choose a fully qualified name such as `"click_it?anon-1"`.

### 3.25.33.33.3 nameindex property

With one exception described belowDepending on the circumstances, a `logicalLocation` object either **MAY, SHALL NOT**, or **SHALL** contain a property named `index` whose value is the array index (§3.7.4) within `theRun.logicalLocations` (§3.14.17) of a `logicalLocation` object that provides the properties for `thisObject`. We refer to the object in `theRun.logicalLocations` as the "cached object."

If `thisObject` is an element of `theRun.logicalLocations`, then `index` **MAY** be present. If present, its value **SHALL** be the index of `thisObject` within `theRun.logicalLocations`.

Otherwise, if `theRun.logicalLocations` is absent, or if it does not contain a cached object for `thisObject`, then `index` **SHALL NOT** be present.

Otherwise (that is, if `thisObject` belongs to a result, and `theRun.logicalLocations` contains a cached object for `thisObject`), then `index` **SHALL** be present, and its value **SHALL** be the array index within `theRun.logicalLocations` of the cached object.

If `index` is present, `thisObject` **SHALL** take all properties present on the cached object. If `thisObject` contains any properties other than `index`, they **SHALL** equal the corresponding properties of the cached object.

> NOTE 1: This allows a SARIF producer to reduce the size of the log file by reusing the same `logicalLocation` object in multiple results.

> NOTE 2: For examples of the use of an `index` property to locate a cached object, see §3.38.2.

### 3.33.4 name property

A `logicalLocation` object **SHOULD** contain a property named `name` whose value is the logical name of the programmatic construct specified by this object. For example, this property might contain the name of a class or a method.

The `name` property **SHALL** be suitable for display and **SHALL** follow the naming rules for logical names described in §3.33.2.

  

EXAMPLE 1NOTE: A C++ analysis tool might have available both the source code form of a function name and the compiler's "decorated" function name (which encodes the function signature in a manner that is compiler-dependent and not easily readable). The tool would place the source code form of the function name in the `name` property, and the decorated name in the `decoratedName` property (§3.33.6).

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

EXAMPLE 2: In this C++ example, the fully qualified name is `"b::c(float)"`, so `"name"` is the rightmost component, `"c(float)"`.

```
"logicalLocations":{                    # See §.
  "b::c(float)": {
  {                                         #   "name": "c(float)",
      ...
  }
}
```

EXAMPLE 3: In this example, the logical location is a top-level C++ function named `functionF`, and `name` matches the property name, so it can be omitted.

```
"logicalLocations": {
    "functionF": {
        "kind": "function"
    }
}
```

EXAMPLE 4: In this example, the logical location is a top-level C++ function, and `name` equals the property name, but the log file creator has chosen to include it anyway.

```
"logicalLocations": {
    "functionF": {
        "name": "functionF",
        "kind": "function"
    }
}
```

EXAMPLE 5: 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. `fullyQualifiedName` also does not equal the property name, so it cannot be omitted either.

```
"logicalLocations": {
    "functionF-0": {
        "name": "functionF",
        "fullyQualifiedName": "functionF",
        "kind": "function"
    }
}
```

## 3.25.41.1.1 fullyQualifiedName property

```
A logicalLocation object.
  "name": "c(float)",
  "fullyQualifiedName": "b::c(float)", # See §3.33.5.
  "kind": "function"                   # See §3.33.7 either SHALL
}
```

## 3.33.5 fullyQualifiedName property

Depending on the circumstances, a `logicalLocation` object either **SHOULD** or **MAY** contain a property named `fullyQualifiedName` whose value is the fully qualified name of the logical location. This name **SHALL** follow the naming rules for fully qualified names described in §3.33.2.

If ~~the fully qualified name does not equal the property name for~~ this `logicalLocation` ~~object in the `run.logicalLocations`~~ object ~~(§),~~represents a top-level logical location, then `fullyQualifiedName` **MAY** be present. If present, it **SHALL** equal `name`; if absent, it **SHALL** default to `name`. If this object does not represent a top-level logical location, `fullyQualifiedName` **SHOULD** be present.

It is possible for two or more distinct logical locations to have the same fully qualified name.

NOTE: This is an extremely rare corner case.

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 ~~**SHALL** be present. This is an extremely rare corner case. See § for an explanation of the corner case and for an example. Otherwise,~~ `fullyQualifiedName` ~~**MAY** be present.~~for both would be `A::B`. However, they would be distinguished by their `parentIndex` properties:

```
"logicalLocations": [
  {
    "name": "B",
    "fullyQualifiedName": "A::B",
    "kind": "namespace",
    "parentIndex": 1
  },
  {
    "name": "A",
    "kind": "namespace"
  },
  {
    "name": "B",
    "fullyQualifiedName": "A::B",
    "kind": "type",
    "parentIndex": 3
  },
  {
    "name": "A",
    "kind": "namespace"
  }
]
```

  

NOTE: There are a few reasons the fullyQualifiedName property exists, even though the information it contains can be reconstructed from the name properties of this object and its parent objects in run.logicalLocations:

- run.logicalLocations might not be present.
- It allows a SARIF viewer to display the logical location in a way that is easily understood by users.
- As mentioned in §3.28.1, fullyQualifiedName 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").

### 3.25.53.33.6 decoratedName property

A logicalLocation 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 (§) also be present.~~

NOTE: Some compilers refer to this representation as a "mangled name." It typically encodes the function's name, signature, return type, and the class and namespace (if any) to which it belongs.

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

```
{                                              # A "logicalLocation" object
  "name": "c(float)",
  "fullyQualifiedName": "b::c(float)",
  "decoratedName": "?c@b@@AAGXM@Z"
}
```

### 3.25.63.33.7 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:.

Although the values suggested here are useful in the specified categories (for example, "member" is useful in describing executable code), they **MAY** be used in other contexts as appropriate.

- Values for locations within executable code:
  - "function"
  - "member"
  - "module"
  - "namespace"
- ~~"package"~~
  - "resource"
  - "type"
  - "returnType"
  - "parameter"
  - "variable"

  

- Values for locations within XML or HTML documents:
    - `"element"`
    - `"attribute"`
    - `"text"`
    - `"comment"`
    - `"processingInstruction"`
    - `"dtd"`
    - `"declaration"`

EXAMPLE 1: Consider the following XML document:

```
1.  <?xml version="1.0"?>
2.  <orders>
3.    <order number="">
4.      <total>-$3.25</total>
5.    </order>
6.  </order>
```

Suppose that an analysis tool detects errors on line 3 (the order number is blank) and line 4 (the total is negative). It might represent the logical locations of these errors as XML Paths (although this is not required), as follows:

```
{                                  # A run object (§3.14)
  "results": [                     # See §3.14.23.
    {                              # A result object (§3.27).
      "locations": [               # See §3.27.12.
        {                          # A location object (§3.28).
          "logicalLocations": [    # See §3.28.4.
            {                      # A logicalLocation object.
              "fullyQualifiedName": "/orders/order[1]/@number",
              "index": 2
            }
          ]
        }
      ],
      ...
    },
    {
      "locations": [
        {
          "logicalLocations": [
            {
              "fullyQualifiedName": "/orders/order[1]/total/text()",
              "index": 3
            }
          ]
        }
      ],
      ...
    }
  ],

  "logicalLocations": [            # See §3.14.17.
    {                              # A logicalLocation object.
      "name": "orders",
      "fullyQualifiedName": "/orders",
      "kind": "element"
    },
    {
      "name": "order[1]",
      "fullyQualifiedName": "/orders/order[1]",
      "kind": "element",
      "parentIndex": 0
```

  

```
    },
    {
      "name": "number",
      "fullyQualifiedName": "/orders/order[1]/@number",
      "kind": "attribute",
      "parentIndex": 1
    },
    {
      "name": "text",
      "fullyQualifiedName": "/orders/order[1]/text()",
      "kind": "text",
      "parentIndex": 1
    }
  ]
}
```

- Values for locations within JSON documents:
  - `"object"`
  - `"array"`
  - `"property"`
  - `"value"`

EXAMPLE 2: Consider the following JSON document:

```
1.  {
2.    "orders": [
3.      {
4.        "productIds": [ "A-101", "", "A-223" ],
5.        "total": "-$3.25"
6.      }
7.    ]
8.  }
```

Suppose that an analysis tool detects errors on line 4 (one of the product ids blank) and line 5 (the total is negative). It might represent the logical locations of these errors as JSON Pointers (although this is not required), as follows:

```
{                                 # A run object (§3.14)
  "results": [                    # See §3.14.23.
    {                             # A result object (§3.27).
      "locations": [              # See §3.27.12.
        {                         # A location object (§3.28).
          "logicalLocation": {    # See §3.28.4.
            "fullyQualifiedName": "/orders/0/productIds/1",
            "index": 3
          }
        }
      ]
    },
    {
      "locations": [
        {
          "logicalLocation": {
            "fullyQualifiedName": "/orders/0/total",
            "index": 4
          }
        }
      ]
    }
  ],

  "logicalLocations": [           # See §3.14.17.
    {                             # A logicalLocation object (§3.33).
      "name": "orders",
```

  

```
        "fullyQualifiedName": "/orders",
        "kind": "array"
      },
      {
        "name": "0",
        "fullyQualifiedName": "/orders/0",
        "kind": "object",
        "parentIndex": 0
      },
      {
        "name": "productIds",
        "fullyQualifiedName": "/orders/0/productIds",
        "kind": "array",
        "parentIndex": 1
      },
      {
        "name": "1",
        "fullyQualifiedName": "/orders/0/productIds/1",
        "kind": "value",
        "parentIndex": 2
      },
      {
        "name": "total",
        "fullyQualifiedName": "/orders/0/total",
        "kind": "property",
        "parentIndex": 1
      }
    ]
}
```

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

### 3.25.7 parentKey property

If thea logical location represented by theis both a member and a type (for example, a nested class in C++ or C#), the value of `kind`, if present, **SHALL** be `"type"`.

> NOTE: The purpose of this property is to help result management systems group results that occur in the same logical location. If one result specifies the logical location “namespace A”, and another result specifies the logical location “class A”, the difference in the `kind` property between the two results tells the result management system to sort them into different groups.

### 3.33.8 parentIndex property

If this `logicalLocation` object isrepresents a nested logical location, then the `logicalLocation` objectit **SHALL** contain a property named `parentKeyparentIndex` whose value is a string that matches the array index (§3.7.4property name) of the parent `logicalLocation` object within `runtheRun.logicalLocations` (§3.14.17).

If the logical location represented by the `logicalLocation` object isIf `thisObject` represents a top-level logical location, then the `parentKey` property`parentIndex` **SHALL** be absent.

> NOTE: `parentIndex` makes it possible to navigate from the `logicalLocation` object representing a nested logical location to the `logicalLocation` objects representing each of its parent logical locations in turn, up to the top-level logical location.

> EXAMPLE: In this example, the logical location `n::f(void)` is nested within the top-level logical location `n`. The `logicalLocation` object representing `n::f(void)` contains a `parentIndex` property that points to the object representing `n`; the object representing `n` does not contain a `parentIndex` property.

  

```
{                                                # A run object (§3.14).
  "logicalLocations": [                          # See §3.14.17.
    {
      "name": "f(void)",                         # See §3.33.4.
      "fullyQualifiedName": "n::f(void)",        # See §3.33.5.
      "kind": "function",                        # See §3.33.7.
      "parentIndex": 1
    },
    {
      "name": "n",
      "kind": "namespace"
    }
  ]
}
```

## 3.34 locationRelationship object

### 3.34.1 General

A `locationRelationship` object specifies one or more directed relationships from one `location` object (§3.28), which we refer to as `theSource`, to another one, which we refer to as `theTarget`.

`locationRelationship` objects appear as elements of the `location.relationships` array (§3.28.7). The `location` object containing this property is `theSource`.

> EXAMPLE: In this example, the location relationships specify that the file f.h in which the result was found is included by g.h, which is in turn included by g.c. Depending on the circumstances, it might or might not be useful to include both the `"includes"` and `"isIncludedBy"` relationships, as this example does for g.h.

```
{                                            # A result object (§3.27).
  "locations": [                             # See §3.27.12.
    {                                        # A location object (§3.28).
      "id": 0,                               # See §3.28.2.
      "physicalLocation": {
        "artifactLocation": {
          "uri": "f.h"
        },
        "region": {
          "startLine": 42
        }
      },
      "relationships": [                     # See §3.28.7
        {                                    # A locationRelationship object.
          "target": 1,                       # See §3.34.2.
          "kinds": [ "isIncludedBy" ]        # See §3.34.3.
        }
      ]
    }
  ],

  "relatedLocations": [                      # See §3.27.22.
    {
      "id": 1,
      "physicalLocation": {
        "artifactLocation": {
          "uri": "g.h"
        },
        "region": {
          "startLine": 17                    # The line that includes f.h.
        }
      },
```

  

```
      "relationships": [
        {
          "target": 0,
          "kinds": [ "includes" ]
        },
        {
          "target": 2,
          "kinds": [ "isIncludedBy" ]
        }
      ]
    },
    {
      "id": 2
      "physicalLocation": {
        "artifactLocation": {
          "uri": "g.c"
        },
        "region": {
          "startLine": 8                  # The line that includes g.h.
        }
      },
      "relationships": [
        {
          "target": 1,
          "kinds": [ "includes" ]
        }
      ]
    }
  ]
}
```

## 3.34.2 target property

A `locationRelationship` object **SHALL** contain a property named `target` whose value is a non-negative integer which identifies `theTarget` (see §3.34.1) among all `location` objects (§3.28) in `theResult` by virtue of being equal to `theTarget.id` (§3.28.2).

> NOTE: Negative values are forbidden because their use might suggest some non-obvious semantic difference between positive and negative values.

## 3.34.3 kinds property

A `locationRelationship` object **MAY** contain a property named `kinds` whose value is an array of one or more unique (§3.7.3) strings each of which specifies a relationship between `theSource` and `theTarget` (see §3.34.1). If `kinds` is absent, it **SHALL** default to `[ "relevant" ]` (see below for the meaning of `"relevant"`).

When possible, SARIF producers **SHOULD** use the following values, with the specified meanings.

- `"includes"`: The artifact identified by `theSource` includes the artifact identified by `theTarget`.
- `"isIncludedBy"`: The artifact identified by `theSource` is included by the artifact identified by `theTarget`.
- `"relevant"`: `theTarget` is relevant to `theSource` in a way not covered by other relationship kinds.

If none of these values are appropriate, a SARIF producer **MAY** use any value.

> NOTE: Although `"relevant"` is a catch-all for any relationship not described by the other values, a producer might still wish to define its own more specific values.

  

In particular, the values defined for `logicalLocation.kind` (§3.33.7) and `threadFlowLocation.kinds` (§3.38.8) might prove useful.

### 3.34.4 description property

A `locationRelationship` object **MAY** contain a property named `description` whose value is a `message` object (§3.11) that describes the relationship.

## 3.35 suppression object

### 3.35.1 General

A `suppression` object describes a request to suppress a result.

> NOTE 1: The `suppression` object is valuable in compliance scenarios, where teams must show an auditor that they have looked at all results that corporate policy requires, and either fixed them or explicitly decided not to fix them. The `kind` property (§3.35.2) enables a review process that ensures that the engineering team agrees with the suppression, and makes the agreement explicit in the log file.

> NOTE 2: 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.

### 3.35.2 kind property

A `suppression` object **SHALL** contain a property named `kind` whose value is a string with one of the following values, with the specified meanings:

- `"inSource"`: The result is suppressed by a syntactic construct offered by the programming language.

  EXAMPLE: The `SuppressMessage` attribute in the .NET Framework.

- `"external"`: The result is suppressed in an external, persistent store.

  EXAMPLE: A database containing historical information about the results from analysis tools. Such a store might offer the ability to mark a result as "suppressed," meaning that if the result is encountered again, it is to be ignored.

### 3.35.3 status property

A `suppression` object **MAY** contain a property named `status` whose value is a string with one of the following values, with the specified meanings:

- `"accepted"`: The suppression is accepted.
- `"underReview"`: The engineering team is discussing the result to decide if they will suppress it.
- `"rejected"`: The engineering team decided not to suppress the result.

### 3.35.4 location property

A `suppression` object **MAY** contain a property named `location` whose value is a `location` object (§3.28) that specifies the location where the suppression is persisted.

> NOTE: In the common scenario, a suppression is represented by a source code construct (which we will refer to as a "suppression construct") such as an attribute or a specially formatted comment at the location where the result was detected. In this scenario, `location` is unnecessary, although it is permitted, because an end user who

  

navigates from the result to the source code location will see the suppression attribute or comment near the relevant code.

Nevertheless, there are several scenarios where `location` is useful. Here are some examples:

When the suppression construct is placed in a separate compiled source file, `kind` (§3.35.2) is `"inSource"`, and `location.physicalLocation` (§3.28.3) specifies the location of the suppression attribute in that separate file.

Even when the suppression construct is adjacent to the result line, `location.physicalLocation` can be useful because it allows you to include in the log file a source code snippet containing the suppression construct, using `location.physicalLocation.region.snippet` (§3.29.4, §3.30.13).

When a tool detects a result within a method, but the suppression construct is applied to some higher-level construct such as the enclosing class, then `kind` is again `"inSource"`, `location.logicalLocation` (§3.28.4) can specify the construct to which the suppression was applied, and `location.physicalLocation` can still usefully specify the location of the suppression construct in the source file, since it is distant from the result.

In a similar case, a binary analysis tool that detected the suppression within an executable file's metadata could provide `location.logicalLocation` even if it could not provide `location.physicalLocation`.

If a suppression is stored in a separate, non-compiled file, sometimes called a "sidecar file," `kind` is `"external"`, and `location.physicalLocation` specifies the location of the suppression within the sidecar file. The sidecar file might even be another SARIF file.

If a suppression is stored in a database, `kind` is again `"external"`, and `location.physicalLocation` might specify the URI of a query that returns the database information that describes the suppression.

### 3.35.5 guid property

A `suppression` object **MAY** contain a property named `guid` whose value is a GUID-valued string (§3.5.3).

NOTE: This can be used, for example, to link a `suppression` object in a SARIF file to suppression information in a result management system's database.

### 3.35.6 justification property

A `suppression` object **MAY** contain a property named `justification` whose value is a user-supplied string that explains why the result was suppressed.

This is one of the few properties that contain textual content supplied by a user rather than by a tool or taxonomy (see §3.19.3) vendor. As such, it might contain undesirable content. Therefore, SARIF consumers **SHOULD** exercise appropriate caution when displaying, sharing, or publishing this information.

NOTE: This property exists because the information it contains is commonly made available by existing suppression mechanisms such as the `SuppressMessage` attribute in the .NET Framework.

  

# 3.36 codeFlow object

## 3.36.1 General

# ~~3.26~~1.1 ~~codeFlow object~~

## ~~3.26.1~~1.1.1 ~~General~~

A `codeFlow` object describes the progress of one or more programs through one or more thread flows, which together ~~result in~~lead to the detection of a ~~result~~problem in the system being analyzed. We define a thread flow as a temporally ordered sequence of code locations occurring within a single thread of execution, typically an operating system thread or a fiber. The thread flows in a code flow **MAY** lie within a single process, within multiple processes on the same machine, or within multiple processes on multiple machines.

EXAMPLE

```
{                                       # A result object (§3.27).
  "codeFlows": [                        # See §3.27.18.
    {                                   # A codeFlow object (§)..
      "message": {                      # See §3.36.2.
        "text": "..."
      },

      "threadFlows": [                  # See §3.36.3.
        {                               # A threadFlow object (§3.37).
          "id": "thread-123",           # See §3.37.2.
          "message": {                  # See §3.37.3.
            "text": "..."
          },

          "locations": [                # See §3.37.6.
            {                           # A threadFlowLocation object (§0).
              "location": {             # See §3.38.3.
                "physicalLocation": {   # See §3.28.3.
                  "fileLocationartifactLocation": {
                    "uri": "ui/window.c",
                    "uriBaseId": "SRCROOT"
                  },

                  "region": {
                    "startLine": 42
                  }
                }
              },

              "state": {                # See §3.38.9.
                "x": {
                  "text": "42","
                },
                "y": {
                  "text": "54","
                },
                "x + y": {
                  "text": "96"
                }
              },

              "nestingLevel": 0,        # See §3.38.10.
              "executionOrder": 2       # See §3.38.11.
            }
```

  

```
            ]
          }
        ]
      }
    ]
}
```

## ~~3.26.2~~3.36.2 message property

A `codeFlow` object **MAY** contain a property named `message` whose value is a `message` object (§3.11) relevant to the code flow.

## ~~3.26.3~~3.36.3 threadFlows property

A `codeFlow` object **SHALL** contain a property named `threadFlows` whose value is an array of one or more ~~unique (§)~~ `threadFlow` objects (§3.37~~),~~) each of which describes the progress of a program through a single thread of execution such as an operating system thread or a fiber.

## ~~3.26.4 properties property~~

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

# ~~3.27~~3.37 threadFlow object

## ~~3.27.1~~3.37.1 General

A thread flow is a sequence of code locations that specify a possible path through a single thread of execution such as an operating system thread or a fiber.

For an example, see §3.36.1.

## ~~3.27.2~~3.37.2 id property

A `threadFlow` object **MAY** contain a property named `id` whose value is a string that uniquely identifies this `threadFlow` within its containing `codeFlow` object (§3.36).

> NOTE: A tool might choose to use an operating system thread id for this purpose. However, if thread ids are reused on a single machine, or if the code flow includes thread flows from more than one machine, the thread id might not be unique.

## ~~3.27.3~~3.37.3 message property

A `threadFlow` object **MAY** contain a property named `message` whose value is a `message` object (§3.11) relevant to the thread flow.

## 3.37.4 initialState property

A `threadFlow` object **MAY** contain a property named `initialState` whose value is an object (§3.6) each of whose property values is a `multiformatMessageString` object (§3.12) that represents the initial value of a relevant item prior to the first location in the thread flow. This property, together with `threadFlowLocation.state` (§3.38.9), enables a SARIF viewer to present a debugger-like "watch window" experience as the user traverses a thread flow.

This property **SHOULD NOT** include items whose values remain constant throughout the thread flow. Such items **SHOULD** be stored in the `immutableState` property (§3.37.5).

For details of how properties within a "state" object are represented, see EXAMPLE 1 in §3.38.9.

#### 3.37.5 immutableState property

A `threadFlow` object **MAY** contain a property named `immutableState` whose value is an object (§3.6) each of whose property values is a `multiformatMessageString` object (§3.12) that represents the value of a relevant item that remains constant throughout the thread flow.

> EXAMPLE: In this example, `immutableState` holds the value of a global variable that remains constant throughout the thread flow.

```
{                                           # A threadFlow object.
  "immutableState": {
    "MaxFiles": {
      "text": "1000"
    }
  }
}
```

#### 3.27.43.37.6 locations property

A `threadFlow` object **SHALL** contain a property named `locations` whose value is an array of one or more `codeFlowCodeLocation``threadFlowLocation` objects (§0). 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 execution order that demonstrates the tool visited them.problem. The elements do not 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.

#### 3.27.5 properties property

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

## 3.281.1 graph object

### 3.28.11.1.1 General

A `graph` object represents a directed graph, a network of nodes and directed edges that describes some aspect of the structure of the code (for example, a call graph). `graph` objects **MAY** be defined both at the run level in `run.graphs` (§) and at the result level in `result.graphs` (§).

A path through a graph, called a "graph traversal," is represented by a `graphTraversal` object (§).

### 3.28.2 id property

A `graph` object **SHALL** contain a property named `id` whose value is a string that uniquely identifies the graph within its containing `run.graphs` property (§) or `result.graphs` property (§). The `id` property does not have to be unique across all `graph` objects in all `result.graphs` properties in the `run`.

### 3.28.3 description property

A `graph` object **MAY** contain a property named `description` whose value is a `message` object (§) that describes the graph.

  

### 3.28.41.1.1 nodes property

A graph object **SHALL** contain a property named nodes whose value is an array of unique (§) node objects (§) which represent the nodes of the graph.

### 3.28.51.1.1 edges property

A graph object **SHALL** contain a property named edges whose value is an array of unique (§) edge objects (§) which represent the edges of the graph.

### 3.28.6 properties property

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

## 3.29 node object

### 3.29.1 General

A node object represents a node in the graph represented by the containing graph object (§).

### 3.29.2 id property

A node object **SHALL** contain a property named id whose value is a string that uniquely identifies the node within the containing graph object (§). id **SHALL** be unique among all nodes in the graph, regardless of nesting (see §).

EXAMPLE: This graph is invalid because two nodes have the same id, even though the nodes are within unrelated nested graphs.

```
{                                  # A graph object (§).
  "nodes": [                       # See §.
    {                              # A node object.
      "id": "n1",
      "children": [                # See §.
        {
          "id": "n3"
        }
      ]
    },
    {
      "id": "n2",
      "children": [
        {
          "id": "n3"               # INVALID: duplicate id.
        }
      ]
    }
  ],
  ...
}
```

### 3.29.31.1.1 label property

A node object **MAY** contain a property named label whose value is a message object (§) that provides a short description of the node.

  

### 3.29.4 location property

A node object **SHOULD** have a property named location whose value is a location object (§) that specifies the location associated with the node.

### 3.29.51.1.1 children property

A node object **MAY** contain a property named children whose value is an array of unique (§) node objects, referred to as "child nodes."

Child nodes are considered to be logically subordinate to their containing node, and to form a "nested graph" within that node.

### 3.29.6 properties property

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

## 3.301.1 edge object

### 3.30.11.1.1 General

An edge object represents a directed edge in the graph represented by the containing graph object (§).

### 3.30.2 id property

An edge object **SHALL** contain a property named id whose value is a string that uniquely identifies the edge within the containing graph object (§).

### 3.30.3 label property

An edge object **MAY** contain a property named label whose value is a message object (§) that provides a short description of the edge.

### 3.30.41.1.1 sourceNodeId property

An edge object **SHALL** contain a property named sourceNodeId whose value is a string that identifies the source node (the node at which the edge starts). It **SHALL** equal the id property (§) of one of the node objects (§) in the containing graph object (§). It **MAY** equal the id of any node within the graph, regardless of nesting (see §).

EXAMPLE: In this example, an edge connects two nodes defined in unrelated nested graphs.

```
{                                  # A graph object (§).
  "nodes": [                       # See §.
    {                              # A node object.
      "id": "n1",
      "children": [                # See §.
        {
          "id": "n3"
        }
      ]
    },
    {
      "id": "n2",
      "children": [
        {
          "id": "n4"
        }
```

  

```
      }
    }
  ],
  "edges": [                        # See §.
    {
      "sourceNodeId": "n3",         # Source node and target node are in separate
      "targetNodeId": "n4"          # nested graphs: ok.
    }
  ],
  ...
}
```

#### 3.30.51.1.1 targetNodeId property

An `edge` object **SHALL** contain a property named `targetNodeId` whose value is a string that identifies the target node (the node at which the edge ends). It **SHALL** equal the `id` property (§) of one of the `node` objects (§) in the containing `graph` object (§). It **MAY** equal `sourceNodeId` (§).

### 3.30.6 properties property

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

## 3.311.1 graphTraversal object

### 3.31.11.1.1 General

A `graphTraversal` object represents a "graph traversal," that is, a path through a graph specified by a sequence of connected "edge traversals," each of which is represented by an `edgeTraversal` object (§). For an example, see §.

### 3.31.2 graphId property

A `graphTraversal` object **SHALL** contain a property named `graphId` whose value is a string that equals the `id` property (§) of the `graph` object (§) being traversed.

The value of `graphId` **SHALL** equal the `id` of a `graph` object that occurs in the `graphs` property (§) of the containing `result` object (§), or the `id` of a `graph` object that occurs in the `graphs` property (§) of the containing `run` object (§), or both (in which case the `graph` object in `result.graphs` takes precedence).

### 3.31.3 description property

A `graphTraversal` object **MAY** contain a property named `description` whose value is a `message` object (§) that describes the graph traversal.

### 3.31.4 initialState property

A `graphTraversal` object **MAY** contain a property named `initialState` whose value is a JSON object (§) each of whose properties represents the value of a relevant expression at the point of entry to the graph. This property, together with `edgeTraversal.finalState` (§), enables a SARIF viewer to present a debugger-like "watch window" experience as the user traverses a graph.

For details of how properties within a "state" object are represented, see §.

### 3.31.51.1.1 edgeTraversals property

A `graphTraversal` object **SHALL** contain a property named `edgeTraversals` whose value is an array of `edgeTraversal` objects (§) which together represent the sequence of edges traversed during this graph traversal.

The `edgeTraversal` objects **SHALL** be connected end to end; that is, the target node of every traversed edge **SHALL** equal the source node of the next edge.

> EXAMPLE: In this example, the `graphTraversal` contains two `edgeTraversal` objects. The id of the first traversed edge is `"e1"`, which connects node `"n1"` to node `"n2"`. The id of the second traversed edge is `"e3"`, which connects node `"n2"` to node `"n4"`. This is a valid graph traversal because the target node of each traversed edge is the source node of the next.
>
> This example also demonstrates the usage of `graphTraversal.initialState` (§) and `edgeTraversal.finalState` (§).

```
{                                                # A result object (§).
  "graphs": [                                    # See §.
    {                                            # A graph object (§).
      "id": "g1",                                # See §.

      "nodes": [                                 # See §.
        { "id": "n1" },                          # A node object (§).
        { "id": "n2" },
        { "id": "n3" },
        { "id": "n4" }
      ],

      "edges": [                                 # See §.
        {                                        # An edge object (§).
          "id": "e1",                            # See §.
          "sourceNodeId": "n1",                  # See §.
          "targetNodeId": "n2"                   # See §.
        },
        {
          "id": "e2",
          "sourceNodeId": "n2",
          "targetNodeId": "n3"
        },
        {
          "id": "e3",
          "sourceNodeId": "n2",
          "targetNodeId": "n4"
        }
      ]
    }
  ],
  "graphTraversals": [                           # See §.
    {                                            # A graphTraversal object (§).
      "graphId": "g1",                           # See §.

      "initialState": {                          # See §.
        "x": "1",
        "y": "2",
        "x + y": "3"
      },

      "edgeTraversals": [                        # See §.
        {                                        # An edgeTraversal object (§).
          "edgeId": "e1",                        # See §.

          "finalState": {                        # See §.
            "x": "4",
```

  

```
              "y": "2",
              "x + y": "6"
            }
          },
          {
            "edgeId": "e3",

            "finalState": {
              "x": "4",
              "y": "7",
              "x + y": "11"
            }
          }
        ]
      }
    ]
}
```

### 3.31.6 properties property

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

## 3.321.1 edgeTraversal object

### 3.32.11.1.1 General

An edgeTraversal object represents the traversal of a single edge during a graph traversal.

### 3.32.21.1.1 edgeId property

An edgeTraversal object **SHALL** contain a property named edgeId whose value is a string which equals the id property (§) of one of the edge objects (§) in the graph identified by the graphId property (§) of the containing graphTraversal object (§).

### 3.32.3 message property

An edgeTraversal object **MAY** contain a property named message whose value is a message object (§) that contains a message to display to the user as the edge is traversed.

### 3.32.41.1.1 finalState property

An edgeTraversal object **MAY** contain a property named finalState whose value is a JSON object (§) each of whose properties represents the value of a relevant expression after the edge has been traversed. This property, together with graphTraversal.initialState (§), enables a viewer to present a debugger-like "watch window" experience as the user traverses a graph.

For details of how properties within a "state" object are represented, see §.

### 3.32.51.1.1 stepOverEdgeCount property

An edgeTraversal object **MAY** contain a property named stepOverEdgeCount whose value is an integer specifying the number of edges a user can step over.

This property is intended to enable a viewing experience in which the user can either step over or step into the traversal of a nested graph (§). Therefore, this property **SHOULD** be specified only on an edge that leads from a node to one of its child nodes, and its value **SHOULD** be the number of edges the user would need to traverse to return to the current nesting level.

  

If this property is present, a SARIF viewer **SHOULD** provide a visual cue informing the user that they have the option of either stepping over the current edge and into the nested graph, or of stepping over the entire traversal of the nested graph.

EXAMPLE: This example defines a graph containing two nested graphs, the first representing code locations in function `A` and the second representing locations in function `B`. Node `na2` in function `A` represents a call to function `B`.

The example defines a graph traversal consisting of a set of edge traversals which start at node `"na1"` in function `A`, call into function `B`, and ultimately return to and continue execution in function `A`.

Suppose the user executes the first edge traversal, which traverses edge `ea1`.The next edge traversal has a `stepOverEdgeCount` property value of 4. Therefore, the SARIF viewer informs her that she can now choose to either step into function `B` by traversing edge `"eab"`, or step over the function call by traversing 4 edges, the last of which (edge `"eba"`) returns to function `A` at node `"na3"`.

If she chooses to enter the nested graph, she will visit the following nodes, in this order:

`[ na1, na2, nb1, nb2, nb3, na3, na4 ]`

If she chooses not to enter the nested graph, the traversal of the edges

`[ eab, eb1, eb2, eba ]`

will be collapsed into a single “step over.” As a result, she will visit the following nodes, in this order:

`[ na1, na2, na3, na4 ]`

```
{                                                   # A result object (§).
  "graphs": [                                       # See §.
    {                                               # A graph object (§).
      "id": "code"
      "nodes": [
        {
          "id": "functionA",
          "children": [
            { "id": "na1" },
            { "id": "na2", "label": "Call functionB" },
            { "id": "na3" },
            { "id": "na4" }
          ]
        },
        {
          "id": "functionB",
          "nodes": [
            { "id": "nb1" },
            { "id": "nb2" },
            { "id": "nb3" }
          ],
        }
      ]
      "edges": [
        { "id": "ea1", "sourceNodeId": "na1", "targetNodeId": "na2" },
        { "id": "ea2", "sourceNodeId": "na2", "targetNodeId": "na3" },
        { "id": "eab", "sourceNodeId": "na2", "targetNodeId": "nb1" },
        { "id": "ea3", "sourceNodeId": "na3", "targetNodeId": "na4" },
        { "id": "eb1", "sourceNodeId": "nb1", "targetNodeId": "nb2" },
        { "id": "eb2", "sourceNodeId": "nb2", "targetNodeId": "nb3" },
        { "id": "eba", "sourceNodeId": "nb3", "targetNodeId": "na3" }
      ]
    }
  ],
```

  

```
  "graphTraversals": [                           # See §.
    {                                            # A graphTraversal object (§).
      "graphId": "code",                         # The graph being traversed.
      "edgeTraversals": [
        { "edgeId": "ea1" },
        {
          "edgeId": "eab",
          "stepOverEdgeCount": 4
        },
        { "edgeId": "eb1" },
        { "edgeId": "eb2" },
        { "edgeId": "eba" },
        { "edgeId": "ea3" }
      ]
    }
  ]
}
```

### 3.32.6 properties property

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

## 3.33 stack object

### 3.33.11.1.1 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.

### 3.33.2 message property

A `stack` object **MAY** contain a property named `message` whose value is `message` object (§) relevant to this call stack.

### 3.33.31.1.1 frames property

A stack object **SHALL** contain a property named `frames` whose value is an array of one or more `stackFrame` objects (§). 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.

### 3.33.4 properties property

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

  

## 3.34 stackFrame object

### 3.34.11.1.1 General

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

### 3.34.2 location property

A `stackFrame` object **MAY** contain a property named `location` whose value is a `location` object (§) specifying the location to which this stack frame refers.

### 3.34.31.1.1 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.

### 3.34.41.1.1 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.

### 3.34.5 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.

### 3.34.6 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.

> NOTE: This is distinct from the `physicalLocation.region.byteOffset` property (§), if any, specified by the `physicalLocation` property (§). `physicalLocation.region.byteOffset` specifies an offset from the start of a file, not from the start of a method.

### 3.34.7 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.

### 3.34.8 properties property

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

## 3.353.38 threadFlowLocation object

### 3.35.13.38.1 General

A `threadFlowLocation` object represents a location visited by an analysis tool in the course of simulating or monitoring the execution of a program.

  

## ~~3.35.2~~3.38.2 ~~step~~index property

~~A~~ Depending on the circumstances, a `threadFlowLocation` object either **MAY**, **SHALL NOT**, or **SHALL** contain a property named ~~`step`~~`index` whose value is ~~an integer specifying the 1-based sequence number of~~ the array index (§3.7.4~~location~~) within `theRun.threadFlowLocations` (§3.14.19) of a `threadFlowLocation` object that provides the ~~thread flow: 1~~properties 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 thread flow.~~

~~A user reading the log file can easily~~`thisObject`. We refer to the ~~location~~object in ~~conversation, for example, "I think the problem occurs at step 6~~`theRun.threadFlowLocations` as the "cached object."

If `thisObject` is an element of `theRun.threadFlowLocations`, then `index` **MAY** be present. If present, its value **SHALL** be the index of `thisObject` within `theRun.threadFlowLocations`.

Otherwise, if `theRun.threadFlowLocations` is absent, or if it does not contain a cached object for `thisObject`, then `index` **SHALL NOT** be present.

Otherwise (that is, if `thisObject` belongs to a result, and `theRun.threadFlowLocations` contains a cached object for `thisObject`), then `index` **SHALL** be present, and its value **SHALL** be the index within `theRun.threadFlowLocations` of the cached object.

If `index` is present, `thisObject` **SHALL** take all properties present on the cached object. If `thisObject` contains any properties other than `index`, they **SHALL** equal the corresponding properties of the cached object.

> NOTE 1: This allows a SARIF producer to reduce the size of the log file by reusing the same `threadFlowLocation` object in multiple thread flows.

> EXAMPLE 1: In this example, `thisObject` is an element of `theRun.threadFlowLocations`. Its array index is known to be 1, so `thisObject.index` does not need to be present, but since it is present, it equals the array index, as required.

```
{                                  # A run object (§3.14).
  "threadFlowLocations": [         # See §3.14.19.
    ...
    {                              # A threadFlowLocation object: thisObject.
      "index": 1,                  # Optional.
      "location": {
        ...
      }
    },
    ...
  ],
  ...
}
```

> EXAMPLE 2: In this example, `thisObject` is not an element of `theRun.threadFlowLocations`; rather, it is an element of `theResult.codeFlows[0].threadFlows[0].locations`. There is no cached object; that is, there is no object in `theRun.threadFlowLocations` that provides the properties for `thisObject`. Therefore, `thisObject.index` is absent, as required.

```
{                                  # A run object (§3.14).
  "results": [                     # See §3.14.23.
    {                              # A result object (§3.27).
      "codeFlows": [               # See §3.27.18.
        {                          # A codeFlow object (§3.36).
```

  

```
          "threadFlows": [          # See §3.36.3.
            {                       # A threadFlow object (§3.37).
              "locations": [        # See §3.37.6.
                {                   # A threadFlowLocation object (thisObject).
                  "location": {     # See §3.38.3.
                    ...
                  }
                }
              ]
            }
          ]
        }
      ],
      ...
    }
  ],
  ...
  "threadFlowLocations": [          # See §3.14.19.
    ...
  ],
}
```

EXAMPLE 3: In this example, `thisObject` is again an element of `theResult.codeFlows[0].threadFlows[0].locations`, not an element of `theRun.threadFlowLocations`. But in this example, there is a cached object, an element of `theRun.threadFlowLocations` that provides the properties for `thisObject`. Therefore, `thisObject.index` is present, as required.

```
{                                   # A run object (§3.14).
  "results": [                      # See §3.14.23.
    {                               # A result object (§3.27).
      "codeFlows": [                # See §3.27.18.
        {                           # A codeFlow object (§3.36).
          "threadFlows": [          # See §3.36.3.
            {                       # A threadFlow object (§3.37).
              "locations": [        # See §3.37.6.
                {                   # An threadFlowLocation object: thisObject.
                  "index": 0        # index is present so no other properties.
                }
              ]
            }
          ]
        }
      ],
      ...
    }
  ],
  ...
  "threadFlowLocations": [          # See §3.14.19.
    {                               # The cached threadFlowLocation object.
      "location": {                 # See §3.38.3.
        ...
      }
    },
    ...
  ],
}
```

### ~~3.35.3~~3.38.3 location property

If location information is available, a `threadFlowLocation` object **SHALL** contain a property named `location` whose value is a `location` object (§3.28) that specifies the location to which the `threadFlowLocation` object refers. If location information is not available, `location` **SHALL** be absent.

There are analysis tools whose native output format includes the equivalent of a SARIF code flow, but which do not provide location information for every step in the code flow. A SARIF converter for such a format might not be able to populate `location`. However, if the native output format associates a human readable message with such a step, the SARIF converter **SHOULD** create a `location` object and populate only its `message` property (§3.28.5). A SARIF direct producer which creates such code flows **SHOULD** populate `location.message`, even if no actual location information is available.

> EXAMPLE: In this example, a file is locked by another program before a thread attempts to write to it. The analysis tool has no location information for the other program; in fact, the analysis tool might merely be simulating an execution sequence in which a *hypothetical* external program locks the file. Nevertheless, it provides a helpful message.
>
> Note the use of `executionOrder` (§3.38.11) to ensure that the location in the external program executes before the location in the program being analyzed.

```
{                                     # A codeFlow object (§3.36).
  "threadFlows": [                    # See §3.36.3.
    {                                 # A threadFlow object (§3.37).
      "message": {                    # See §3.37.3.
        "text": "An external program."
      },
      "locations": [                  # See §3.37.6.
        {                             # A threadFlowLocation object.
          "executionOrder": 1,
          "location": {               # A location object with only a message.
            "message": {
              "text": "File is now locked."
            }
          }
        }
      ]
    },
    {                                 # Another threadFlow object.
      "message": {
        "text": "The program being analyzed."
      },
      "locations": [
        ...
        {
          "executionOrder": 2,
          "location": {
            "message": {
              "text": "Attempt to write to the file."
            },
          "location": {
            "physicalLocation": {
              "fileLocationartifactLocation": {
                "uri": "io/logger.c",
                "uriBaseId": "SRCROOT"
              },
              "region": {
                "startLine": 42,
                "snippet": {
                  "text": "    fprintf(fd, "test\\n");\n"");"
                }
              }
            }
          }
        }
      ]
```

  

```
    }
  ]
}
```

### 3.35.43.38.4 module property

A `threadFlowLocation` 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.

### 3.35.53.38.5 stack property

A `threadFlowLocation` object **MAY** contain a property named `stack` whose value is a `stack` object (§3.44) that represents the call stack leading to this location.

### 3.35.63.38.6 kindwebRequest property

A `threadFlowLocation` object **MAY** contain a property named `kind``webRequest` whose value is a `webRequest` object (§3.46) that describes an HTTP request sent from this location.

NOTE: This property is primarily useful to web analysis tools.

### 3.38.7 webResponse property

A `threadFlowLocation` object **MAY** contain a property named `webResponse` whose value is a `webResponse` object (§3.47string) that describes the response to the HTTP request sent from this location.

NOTE: This property is primarily useful to web analysis tools.

### 3.38.8 kinds property

A `threadFlowLocation` object **MAY** contain a property named `kinds` whose value is an array of unique (§3.7.3) strings that describe the meaning of this location. The interpretation of `kind`strings **SHOULD** be human-readable (as opposed to, for example, GUIDs or hash values).

When possible, SARIF producers **SHOULD** use the following values, with the specified meanings.

Verbs:

- `"acquire"`: Gain ownership of something.
- `"release"`: Relinquish ownership of something.
- `"enter"`: Entry point to a section of the program such as a function.
- `"exit"`: Exit point from a section of the program such as a function.
- `"call"`: Point of call into a section of the program such as a function.
- `"return"`: Point of return from a section of the program such as a function.
- `"branch"`: Conditional transfer of control.

NOTE 1: These values are typically combined with nouns from the list below, as in the examples below.

Nouns:

- `"taint"`: Value obtained from user input.
- `"function"`: Section of a program that can be called into and returned from.
- `"handler"`: Code invoked in response to an exception, signal, or event.
- `"lock"`: Limits access to a resource.
- `"memory"`: Portion of computer's internal storage.

  

* "resource": Anything that can be acquired and released.
* "scope": Section of a program that limits the visibility of variables defined within it.
* "value": The value of a variable.

NOTE 2: "kinds": [ "acquire", "value" ] can be used to denote a variable assignment or initialization.

Miscellaneous:

* "implicit": Code was invoked implicitly, for example by a garbage collector.
* "false": A condition evaluated to false.
* "true": A condition evaluated to true.
* "caution": Execution of the code at this location in the current circumstance requires care.
* "danger": Execution of the code at this location in the current circumstance is dangerous.
* "unknown": The state of an item is not known.
* "unreachable": Code at this location is unreachable.

NOTE 3: Some analysis tools effectively "uncomment" unreachable code, allowing a simulated execution to flow through it. If such a tool detected a problem in the uncommented code, it could mark the threadFlowLocation as "unreachable". An engineering team might then decide to treat this problem with lower priority.

If none of these values are appropriate, a SARIF producer **MAY** use any value.

The interpretations of values other than those above depends on the tool that produced the log file.producer. A SARIF consumer that wishes to take actionact based on kind **SHALL**such values **SHOULD** examine run.tool (§, §)theTool to determine if it (the consumer) knows how to interpret the kind values produced by that toolthem.

kind **SHOULD** be a human-readable string (as opposed to, for example, a GUID or a hash value).

NOTE 4: This might not be necessary if, for example, the consumer has out of band information telling it how to interpret the values.

A SARIF producer **MAY** provide additional kind-dependent information by populating threadFlowLocation.properties with properties whose names and values depend on the kind. A SARIF consumer that knows how to interpret kindkinds for this tool **MAY** use this additional information.

EXAMPLE:

```
"kind": "taintedDataSource"
```

EXAMPLE 1: In this example, tainted data enters the system at this location.

```
"kinds": [
  "acquire",
  "taint"
]
```

EXAMPLE 2: In this example, the "taint" state of a data item at this location is unknown:

```
"kinds": [
  "taint",
  "unknown"
]
```

EXAMPLE 3: In this example, control leaves a function at this location.

```
"kinds": [
```

  

```
  "exit",
  "function"
]
```

### 3.35.7 3.38.9 state property

A `threadFlowLocation` object **MAY** contain a property named `state` whose value is a JSONan object (§3.6)) in which each of whose propertiesproperty name represents the value of an expressionitem relevant to the location in the context of the code flow, and the corresponding property value is a `multiformatMessageString` object (§3.12.) that specifies either the value of or a constraint on that item.

NOTE: This property enables a SARIF viewer to present a debugger-like "watch window" experience as the user navigates through a code flow.

A SARIF viewer **SHALL NOT** assume that expressions mentioned in previous steps but not mentioned in the current step are still present with unchanged values.

EXAMPLE 1: In this example, the `state` property captures the values of the expressions `"x"`, `"y"`, and `"x + y"`, and a constraint on the expression `"y - x"`.

```
{                               # AnA threadFlowLocation object.
  "state": {
    "x": {
      "text": "42","
    },
    "y": {
      "text": "54","
    },
    "x + y": {
      "text": "96"
  }  },
}
```

NOTE: A viewer might use these values to provide a "watch window" experience, showing the changing values of selected variables and expressions as the user steps through a code flow.

The format of each property name **SHALL** be consistent with the syntax of an expression in the programming language in which the code being analyzed was written. Each property value **SHALL** be a string whose format is consistent with the syntax of a value in the programming language in which the code being analyzed was written

```
    "y - x": {
      "text": "{expr} > 0"
    }
  }
}
```

EXAMPLE 2: In C++, a property name within the `state` object might be:

- A variable name such as `"index"`.
- An array element reference such as `"names[index]"`.
- An object property reference such as `"names[index]->first"`.
- Any other expression that produces a value.

EXAMPLE 3: In C++, a property value within the `state` object might be:

- An integer such as `"42"` (note that the property value is a string).

- A string such as `"\"John\""` (~~note the escaped double quotes~~the double quotes are escaped as they would be in a JSON serialization; other serializations might represent the double quotes differently).
- A Boolean such as `"true"`.

In a property value that represents a constraint, the item being constrained **SHALL BE** represented by the string `"{expr}"`. (See EXAMPLE 1 above, which shows a constraint on the expression `"y - x"`.)

A constraint which expresses the equality of `"{expr}"` with a literal value **SHALL** be considered equivalent to that literal value.

> EXAMPLE 4: In a language where `==` denotes value equality, the property value `"{expr} == 42"`, which represents a constraint, is identical in meaning to the property value `"42"`, which represents a value.

### ~~3.35.8~~3.38.10 nestingLevel property

A `threadFlowLocation` object **MAY** contain a property named `nestingLevel` whose value is ~~an~~a non-negative integer that represents any type of logical containment hierarchy among the `threadFlowLocation` objects in the `threadFlow`. Typically, it represents function call depth.

A viewer that renders a `threadFlow` **SHOULD** provide a visual representation of the value of `nestingLevel`. Typically, this would be an indentation indicating the depth of each location in the call tree.

### ~~3.35.9~~3.38.11 executionOrder property

A `threadFlowLocation` object **MAY** contain a property named `executionOrder` whose value is a ~~positive~~non-negative integer that represents the temporal order in which execution reached this location, across all `threadFlowLocation` objects within all `threadFlow` objects belonging to a single `codeFlow` (§3.36). `executionOrder` values are assigned in increasing order of time; for example, execution reaches a `threadFlowLocation` whose `executionOrder` is 2 occurs before it reaches a `threadFlowLocation` whose `executionOrder` is 3. If two `threadFlowLocation`s in different `threadFlow` objects within the same `codeFlow` have the same value for `executionOrder`, it means that execution reached both of those locations simultaneously. For that reason, values of `executionOrder` within a single `threadFlow` **SHALL** be unique.

It is only necessary to assign a value to `executionOrder` when the temporal ordering of a `threadFlowLocation` relative to a location in a different `threadFlow` is significant to the detection of a result.

If ~~this property~~`executionOrder` is absent, it **SHALL** default to ~~0~~-1, which ~~is not otherwise a valid~~indicates that the value ~~for `executionOrder`.~~is unknown (not set).

> ~~timestamp~~NOTE: Negative values are forbidden because their use would suggest some non-obvious semantic difference between positive and negative values.

### ~~3.35.10~~3.38.12 executionTimeUtc property

A `threadFlowLocation` object **MAY** contain a property named ~~`timestamp`~~`executionTimeUtc` whose value is a string in the format specified in §3.9, specifying the UTC date and time at which the thread of execution through the code ~~at~~reached this location ~~was executed. The string **SHALL** be in the format specified in §.~~.

### ~~3.35.11~~3.38.13 importance property

A `threadFlowLocation` **MAY** contain a property named `importance` whose value is a string that specifies the importance of this `threadFlowLocation` in understanding the code flow.

  

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.

### 3.35.123.38.14 propertiestaxa property

A `threadFlowLocation` object **MAY** contain a property named `propertiestaxa` whose value is a property bag (§). 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.

## 3.36 resources object

### 3.36.11.1.1 General

A `resources` object represents items that can be localized, such as message strings and rule metadata.

### 3.36.21.1.1 messageStrings property

an array of zero or more unique (§3.7.3A `resources` object **MAY** contain a property named `messageStrings` whose value is a JSON object (§) each of whose properties represents a single localized string. The property names correspond to resource identifiers (§) within `message`) `reportingDescriptorReference` objects (§). If the property name is used as the value of the `messageId` property (§) of any `message` object in the containing `run` object (§), the property value **SHALL** be a plain text string (§). If the property name is used as the value of the `richMessageId` property (§) of any `message` object in the containing `run` object, the property value **SHALL** be a rich text string (§). A given resource identifier **SHALL NOT** appear both as the value of a `messageId` property and the value of a `richMessageId` property in the same `run` objecteach of which specifies a category into which this `threadFlowLocation` falls.

NOTE: The motivation for this property is an analysis tool that uses a set of rules to guide its analysis as it traces tainted data from a source to a sink. For example, at one location, the tool might apply a rule that says: "If the input to `String.Substr` is tainted, then so is the return value." Such a tool can represent these "helper rules" as a custom taxonomy (§3.19.3), an array of `reportingDescriptor` objects (§3.49). Each member of `threadFlowLocation.taxa` can reference one of these helper rules.

EXAMPLE: This example illustrates the scenario in the above note.

```
{                                    # A run object (§3.14).
```

  "tool": {                # See §3.14.6EXAMPLE:

```
"resources": {
  "messageStrings": {
```

  

```
    "call": "Function call",
    "return": "Function return"
  }
}
```

### 3.36.3 rules property

A `resources` object **MAY** contain a property named `rules` whose value is a JSON object (§), each of whose properties represents a `rule` object (§).

If there is only one `rule` object with a particular `id` (§), 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.

```
"resources": {
.
    "driver": {
      "name": "TaintDetector",
      "rules": {[
       "CA1001": {
          "id": "CA1001TD0001",
          "name": "UntrustedDataStoredInDatabase",
          "shortDescription": {
            "text": "Types that own disposable fields should beData from an
untrusted source was stored in a database."
                 disposable."}
      }  },
    },     ...
    "CA1002": {  ],
      "taxa": [                   # Custom taxonomy (§3.19.3) for helper rules.
        {                         # A reportingDescriptor object (§3.49).
          "id": "CA1002HR0001",
          "name": "SubstrPropogatesTaint",
          "shortDescription": {
           "text": "Do not expose generic lists."
      }
    }
  }
}
Some tools useIf the same rule idinput 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.String.Substr is tainted,
```

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.

```
"resources": {
  "rules": {
    "CA1711-1": {
      "id": "CA1711",
      "messageStrings": {
        "default": "Rename type name {0} so that it does not end in '{1}'"
      }
    },
    "CA1711-2": {
      "id": "CA1711",
      "messageStrings": {
        "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 (§) by means of the result's `ruleId` property (§).

## 3.37 rule object

### 3.37.1 General

A `rule` object contains information that describes a rule. We refer to this information as "rule metadata."

### 3.37.21.1.1 Constraints

```
                         so is the return value."
            }
          },
          ...
        ]
      }
    },

  "results": [                        # See §3.14.23.
    {                                 # A result object §3.27.
      "ruleId": "TD0001",
      ...
      "codeFlows": [                  # See §3.27.18.
        {                             # A codeFlow object (§3.36).
          "threadFlows": [            # See §3.36.3.
            {                         # A threadFlow object (§3.37).
```

"locations": [     # See §3.37.6Either the `shortDescription` property (§) or the `fullDescription` property (§) or both **SHALL** be present.

### 3.37.3 id property

```
.
                ...
                {                     # A rulethreadFlowLocation object.
                  "location": {   # See §3.38.3.
                    "physicalLocation": {
                      "artifactLocation": {
                        "uri": "io/input.c",
                        "uriBaseId": "SRCROOT"
                      },
                      "region": {
                        "startLine": 32
                      }
                    }
                  },
                  "taxa": [
                    {         # A reportingDescriptorReference object (§3.52).
                      "id": "HR0001",
                      "index": 0
                    }
                  ]
                },
                ...
              ]
            }
          ]
```

  

```
          }
        ]
      }
    ]
}
```

# 3.39 graph object

## 3.39.1 General

A `graph` object represents a directed graph, a network of nodes and directed edges that describes some aspect of the structure of the code (for example, a call graph). `graph` objects **MAY** be defined both at the run level in `run.graphs` (§3.14.20) and at the result level in `result.graphs` (§3.27.19).

A path through a graph, called a "graph traversal," is represented by a `graphTraversal` object (§3.42).

## 3.39.2 description property

A `graph` object **MAY** contain a property named `description` whose value is a `message` object (§3.11) that describes the graph.

## 3.39.3 nodes property

A `graph` object **MAY** contain a property named `nodes` whose value is an array of zero or more unique (§3.7.3) `node` objects (§3.40) which represent the nodes of the graph.

## 3.39.4 edges property

~~**SHALL**~~A `graph` object **MAY** contain a property named `edges` whose value is an array of zero or more unique (§3.7.3) `edge` objects (§3.41) which represent the edges of the graph.

# 3.40 node object

## 3.40.1 General

A `node` object represents a node in the graph represented by the containing `graph` object (§3.39), which we refer to as `theGraph`.

## 3.40.2 id property

A `node` object **SHALL** contain a property named `id` whose value is a string that uniquely identifies the node within `theGraph`. `id` **SHALL** be unique among all nodes in `theGraph`, regardless of nesting (see §3.40.5).

> EXAMPLE: This graph is invalid because two nodes have the same `id`, even though the nodes are within unrelated nested graphs.

```
{                                # A graph object (§3.39).
  "nodes": [                     # See §3.39.3.
    {                            # A node object.
      "id": "n1",
      "children": [              # See §3.40.5.
        {
          "id": "n3"
        }
      ]
    },
    {
      "id": "n2",
```

```
      "children": [
        {
          "id": "n3"          # INVALID: duplicate id.
        }
      ]
    }
  ],
  ...
}
```

### 3.40.3 label property

A `node` object **MAY** contain a property named `label` whose value is a `message` object (§3.11) that provides a short description of the node.

### 3.40.4 location property

A `node` object **SHOULD** have a property named `location` whose value is a `location` object (§3.28) that specifies the location associated with the node.

### 3.40.5 children property

A `node` object **MAY** contain a property named `children` whose value is an array of zero or more unique (§3.7.3) `node` objects, referred to as "child nodes."

Child nodes are logically subordinate to their containing node, and form a "nested graph" within that node.

## 3.41 edge object

### 3.41.1 General

An `edge` object represents a directed edge in the graph represented by `theGraph`.

### 3.41.2 id property

An `edge` object **SHALL** contain a property named `id` whose value is a string that uniquely identifies the edge within `theGraph`.

### 3.41.3 label property

An `edge` object **MAY** contain a property named `label` whose value is a `message` object (§3.11) that provides a short description of the edge.

### 3.41.4 sourceNodeId property

An `edge` object **SHALL** contain a property named `sourceNodeId` whose value is a string that identifies the source node (the node at which the edge starts). It **SHALL** equal the `id` property (§3.40.2) of one of the `node` objects (§3.40) in `theGraph`. It **MAY** equal the id of any node within `theGraph`, regardless of nesting (see §3.40.5).

> EXAMPLE: In this example, an edge connects two nodes defined in unrelated nested graphs.

```
{                              # A graph object (§3.39).
  "nodes": [                   # See §3.39.3.
    {                          # A node object.
      "id": "n1",
      "children": [            # See §3.40.5.
        {
```

```
          "id": "n3"
        }
      ]
    },
    {
      "id": "n2",
      "children": [
        {
          "id": "n4"
        }
      ]
    }
  ],
  "edges": [                  # See §3.39.4.
    {
      "sourceNodeId": "n3",    # Source node and target node are in separate
      "targetNodeId": "n4"     # nested graphs: ok.
    }
  ],
  ...
}
```

### 3.41.5 targetNodeId property

An `edge` object **SHALL** contain a property named `targetNodeId` whose value is a string that identifies the target node (the node at which the edge ends). It **SHALL** equal the `id` property (§3.40.2) of one of the `node` objects (§3.40) in `theGraph`. It **MAY** equal `sourceNodeId` (§3.41.4).

## 3.42 graphTraversal object

### 3.42.1 General

A `graphTraversal` object represents a "graph traversal," that is, a path through a graph specified by a sequence of connected "edge traversals," each of which is represented by an `edgeTraversal` object (§3.43). For an example, see §3.42.8.

### 3.42.2 Constraints

Exactly one of the `resultGraphIndex` property (§3.42.3) and the `runGraphIndex` property (§3.42.4) **SHALL** be present.

### 3.42.3 resultGraphIndex property

If a `graphTraversal` object represents the traversal of a `graph` object (§3.39) that resides in `theResult.graphs` (§3.27.19), the `graphTraversal` object **SHALL** contain a property named `resultGraphIndex` whose value is the array index (§3.7.4) within `theResult.graphs` of that `graph` object.

### 3.42.4 runGraphIndex property

If a `graphTraversal` object represents the traversal of a `graph` object (§3.39) that resides in `theRun.graphs` (§3.14.20), the `graphTraversal` object **SHALL** contain a property named `runGraphIndex` whose value is the array index within `theRun.graphs` of that `graph` object.

### 3.42.5 description property

A `graphTraversal` object **MAY** contain a property named `description` whose value is a `message` object (§3.11) that describes the graph traversal.

  

### 3.42.6 initialState property

A `graphTraversal` object **MAY** contain a property named `initialState` whose value is an object (§3.6) each of whose properties is a `multiformatMessageString` object (§3.12) that represents the value of a relevant item at the point of entry to the graph. This property, together with `edgeTraversal.finalState` (§3.43.4), enables a SARIF viewer to present a debugger-like "watch window" experience as the user traverses a graph.

This property **SHOULD NOT** include items whose value remains constant throughout the traversal. Such items **SHOULD** be stored in the `immutableState` property (§3.42.7).

For details of how properties within a "state" object are represented, see EXAMPLE 1 in §3.38.9.

### 3.42.7 immutableState property

A `graphTraversal` object **MAY** contain a property named `immutableState` whose value is an object (§3.6) each of whose properties is a `multiformatMessageString` object (§3.12) that represents the value of a relevant item that remains constant throughout the traversal.

> EXAMPLE: In this example, `immutableState` holds the value of a global variable that remains constant throughout the traversal.

```
{                                           # A graphTraversal object.
  "immutableState": {
    "MaxFiles": {
      "text": "1000"
    }
  }
}
```

### 3.42.8 edgeTraversals property

A `graphTraversal` object **MAY** contain a property named `edgeTraversals` whose value is an array of zero or more `edgeTraversal` objects (§3.43) which together represent the sequence of edges traversed during this graph traversal.

The `edgeTraversal` objects **SHALL** be connected end to end; that is, the target node of every traversed edge except the last **SHALL** equal the source node of the next edge.

> EXAMPLE: In this example, the `graphTraversal` contains two `edgeTraversal` objects. The id of the first traversed edge is `"e1"`, which connects node `"n1"` to node `"n2"`. The id of the second traversed edge is `"e3"`, which connects node `"n2"` to node `"n4"`. This is a valid graph traversal because the target node of each traversed edge is the source node of the next.
>
> This example also demonstrates the usage of `graphTraversal.initialState` (§3.42.6) and `edgeTraversal.finalState` (§3.43.4).

```
{                                           # A result object (§3.27).
  "graphs": [                               # See §3.27.19.
    {                                       # A graph object (§3.39).
      "nodes": [                            # See §3.39.3.
        { "id": "n1" },                     # A node object (§3.40).
        { "id": "n2" },
        { "id": "n3" },
        { "id": "n4" }
      ],

      "edges": [                            # See §3.39.4.
        {                                   # An edge object (§3.41).
          "id": "e1",                       # See §3.41.2.
          "sourceNodeId": "n1",             # See §3.41.4.
```

  

```
          "targetNodeId": "n2"             # See §3.41.5.
        },
        {
          "id": "e2",
          "sourceNodeId": "n2",
          "targetNodeId": "n3"
        },
        {
          "id": "e3",
          "sourceNodeId": "n2",
          "targetNodeId": "n4"
        }
      ]
    }
  ],

  "graphTraversals": [                     # See §3.27.20.
    {                                      # A graphTraversal object (§3.42).
      "resultGraphIndex": 0,               # See §3.42.3.

      "initialState": {                    # See §3.42.6.
        "x": {
          "text": "1"
        },
        "y": {
          "text": "2"
        },
        "x + y": {
          "text": "3"
        }
      },

      "edgeTraversals": [                  # See §3.42.8.
        {                                  # An edgeTraversal object (§3.43).
          "edgeId": "e1",                  # See §3.43.2.

          "finalState": {                  # See §3.43.4.
            "x": {
              "text": "4"
            },
            "y": {
              "text": "2"
            },
            "x + y": {
              "text": "6"
            }
          }
        },
        {
          "edgeId": "e3",

          "finalState": {
            "x": {
              "text": "4"
            },
            "y": {
              "text": "7"
            },
            "x + y": {
              "text": "11"
            }
          }
        }
      ]
```

  

```
    }
  ]
}
```

# 3.43 edgeTraversal object

## 3.43.1 General

An `edgeTraversal` object represents the traversal of a single edge during a graph traversal.

## 3.43.2 edgeId property

An `edgeTraversal` object **SHALL** contain a property named `edgeId` whose value is a string which equals the `id` property (§3.41.2) of one of the `edge` objects (§3.41) in the graph identified by the `resultGraphIndex` property (§3.42.3) or the `runGraphIndex` property (§3.42.4a string) of the containing `graphTraversal` object (§3.42).

## 3.43.3 message property

An `edgeTraversal` object **MAY** contain a property named `message` whose value is a `message` object (§3.11) that contains a message to display to the user as the edge is traversed.

## 3.43.4 finalState property

An `edgeTraversal` object **MAY** contain a property named `finalState` whose value is an object (§3.6) each of whose properties is a `multiformatMessageString` object (§3.12) that represents the value of a relevant item after the edge has been traversed.

> NOTE: This property, together with `graphTraversal.initialState` (§3.42.6), enables a viewer to present a debugger-like “watch window” experience as the user traverses a graph.

A SARIF viewer **SHALL** display only those properties that are explicitly present in the `finalState` property of the current `edgeTraversal`. It **SHALL NOT** assume that properties present in previous steps are still present with unchanged values.

For details of how properties within a “state” object are represented, see §3.38.9.

## 3.43.5 stepOverEdgeCount property

An `edgeTraversal` object **MAY** contain a property named `stepOverEdgeCount` whose value is a non-negative integer specifying the number of edges a user can step over.

This property is intended to enable a viewing experience in which the user can either step over or step into the traversal of a nested graph (§3.40.5). Therefore, this property **SHOULD** be specified only on an edge that leads from a node to one of its child nodes, and its value **SHOULD** be the number of edges the user would need to traverse to return to the current nesting level.

If this property is present, a SARIF viewer **MAY** provide a visual cue informing the user that they have the option of either stepping over the current edge and into the nested graph, or of stepping over the entire traversal of the nested graph.

> EXAMPLE: This example defines a graph containing two nested graphs, the first representing code locations in function `A` and the second representing locations in function `B`. Node `na2` in function `A` represents a call to function `B`.
>
> The example defines a graph traversal consisting of a set of edge traversals which start at node `"na1"` in function `A`, call into function `B`, and ultimately return to and continue execution in function `A`.

  

Suppose the user executes the first edge traversal, which traverses edge `ea1`.The next edge traversal has a `stepOverEdgeCount` property value of 4. Therefore, the SARIF viewer informs her that she can now choose to either step into function `B` by traversing edge `"eab"`, or step over the function call by traversing 4 edges, the last of which (edge `"eba"`) returns to function `A` at node `"na3"`.

If she chooses to enter the nested graph, she will visit the following nodes, in this order:

```
[ na1, na2, nb1, nb2, nb3, na3, na4 ]
```

If she chooses not to enter the nested graph, the traversal of the edges

```
[ eab, eb1, eb2, eba ]
```

will be collapsed into a single "step over." As a result, she will visit the following nodes, in this order:

```
[ na1, na2, na3, na4 ]
```

```
{                                                  # A result object (§3.27).
  "graphs": [                                      # See §3.27.19.
    {                                              # A graph object (§3.39).
      "nodes": [
        {
          "id": "functionA",
          "children": [
            { "id": "na1" },
            { "id": "na2", "label": "Call functionB" },
            { "id": "na3" },
            { "id": "na4" }
          ]
        },
        {
          "id": "functionB",
          "nodes": [
            { "id": "nb1" },
            { "id": "nb2" },
            { "id": "nb3" }
          ],
        }
      ]
      "edges": [
        { "id": "ea1", "sourceNodeId": "na1", "targetNodeId": "na2" },
        { "id": "ea2", "sourceNodeId": "na2", "targetNodeId": "na3" },
        { "id": "eab", "sourceNodeId": "na2", "targetNodeId": "nb1" },
        { "id": "ea3", "sourceNodeId": "na3", "targetNodeId": "na4" },
        { "id": "eb1", "sourceNodeId": "nb1", "targetNodeId": "nb2" },
        { "id": "eb2", "sourceNodeId": "nb2", "targetNodeId": "nb3" },
        { "id": "eba", "sourceNodeId": "nb3", "targetNodeId": "na3" }
      ]
    }
  ],

  "graphTraversals": [                             # See §3.27.20.
    {                                              # A graphTraversal object (§3.42).
      "resultGraphIndex": 0,                       # The graph being traversed.
      "edgeTraversals": [
        { "edgeId": "ea1" },
        {
          "edgeId": "eab",
          "stepOverEdgeCount": 4
        },
        { "edgeId": "eb1" },
        { "edgeId": "eb2" },
        { "edgeId": "eba" },
```

```
        { "edgeId": "ea3" }
      ]
    }
  ]
}
```

# 3.44 stack object

## 3.44.1 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.

## 3.44.2 message property

A `stack` object **MAY** contain a property named `message` whose value is `message` object (§3.11) relevant to this call stack.

## 3.44.3 frames property

A stack object **SHALL** contain a property named `frames` whose value is an array of zero or more `stackFrame` objects (§3.45). 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.

# 3.45 stackFrame object

## 3.45.1 General

A `stackFrame` object describes a single stack frame within a call stack (§3.44).

## 3.45.2 location property

A `stackFrame` object **MAY** contain a property named `location` whose value is a `location` object (§3.28) specifying the location to which this stack frame refers.

If location information is unavailable (as it might be, for example, when stepping from application code into library code or operating system code), `location` **SHOULD** be present and **SHOULD** contain a `message` property (§3.28) (for example, with a message string `"Call into external code"`).

## 3.45.3 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.

## 3.45.4 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.

  

### 3.45.5 parameters property

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

## 3.46 webRequest object

### 3.46.1 General

A `webRequest` object describes an HTTP request [RFC7230]. The response to the request is described by a `webResponse` object (§3.47).

> NOTE 1: This object is primarily useful to web analysis tools.

A `webRequest` object does not need to represent a valid HTTP request.

> NOTE 2: This allows an analysis tool that intentionally sends invalid HTTP requests to use the `webRequest` object.

### 3.46.2 index property

Depending on the circumstances, a `webRequest` object either **MAY, SHALL NOT**, or **SHALL** contain a property named `index` whose value is the array index (§3.7.4) within `theRun.webRequests` (§3.14.21) of a `webRequest` object that provides the properties for `thisObject`. We refer to the object in `theRun.webRequests` as the "cached object."

If `thisObject` is an element of `theRun.webRequests`, then `index` **MAY** be present. If present, its value **SHALL** be the index of `thisObject` within `theRun.webRequests`.

Otherwise, if `theRun.webRequests` is absent, or if it does not contain a cached object for `thisObject`, then `index` **SHALL NOT** be present.

Otherwise (that is, if `thisObject` belongs to a result, and `theRun.webRequests` contains a cached object for `thisObject`), then `index` **SHALL** be present, and its value **SHALL** be the array index within `theRun.webRequests` of the cached object.

If `index` is present, `thisObject` **SHALL** take all properties present on the cached object. If `thisObject` contains any properties other than `index`, they **SHALL** equal the corresponding properties of the cached object.

> NOTE 1: This allows a SARIF producer to reduce the size of the log file by reusing the same `webRequest` object in multiple results.

> NOTE 2: For examples of the use of an `index` property to locate a cached object, see §3.38.2.

### 3.46.3 protocol property

A `webRequest` object **SHOULD** contain a property named `protocol` whose value is a string containing the name of the web protocol used in the request, found on the HTTP request line.

> EXAMPLE: `"protocol": "HTTP"`

### 3.46.4 version property

A `webRequest` object **SHOULD** contain a property named `version` whose value is a string containing the version of the web protocol used in the request, found on the HTTP request line.

> EXAMPLE: `"version": "1.1"`

  

### 3.46.5 target property

A `webRequest` object **SHOULD** contain a property named `target` whose value is a string containing the target of the request, found on the HTTP request line, in the form defined by §5.3 ("Request Target") of the HTTP standard [RFC7230].

### 3.46.6 method property

A `webRequest` object **SHOULD** contain a property named `method` whose value is a string containing the HTTP method used in the request, found on the HTTP request line. The string **SHOULD** be one of the values `"GET"`, `"PUT"`, `"POST"`, `"DELETE"`, `"PATCH"`, `"HEAD"`, `"OPTIONS"`, `"TRACE"`, or `"CONNECT"`.

### 3.46.7 headers property

A `webRequest` object **SHOULD** contain a property named `headers` whose value is an object (§3.6) whose property names are the names of the HTTP headers in the request (for example, `"Content-Type"`) and whose corresponding values are the header values (for example, `"text/plain; charset=ascii"`).

### 3.46.8 parameters property

A `webRequest` object **MAY** contain a property named `parameters` whose value is an object (§3.6) whose property names are the names of the parameters in the request and whose corresponding values are the values of those parameters.

> NOTE: The `parameters` property exists as a convenience for the log file consumer. If it is absent, the consumer can parse the parameters from `body` (§3.46.9), in the case of a forms post, or from the query portion of `uri` (§3.46.5).

### 3.46.9 body property

A `webRequest` object **MAY** contain a property named `body` whose value is an `artifactContent` object (§3.3) containing the body of the request.

If the request body is entirely textual, `body.text` (§3.3.2) **SHOULD** be present. If present, it **SHALL** contain the request body, transcoded to UTF-8 if necessary.

> NOTE 1: The transcoding is required because all textual content in a SARIF log file is represented in UTF-8 (see §3.1).

> NOTE 2: If necessary, the character encoding actually used in the request can be deduced from the value of the `Content-Type` header (see §3.46.7), for example, `"text/plain; charset=ascii"`.

If the request body is entirely textual, `body.binary` (§3.3.3) **MAY** be present. If present, it **SHALL** contain the MIME Base64 encoding [RFC2045] of the body as it was actually transmitted.

If the request body consists partially or entirely of binary data, `body.binary` **SHALL** be present and **SHALL** contain the MIME Base64 encoding of the body. In this situation, `body.text` **SHALL** be absent.

## 3.47 webResponse object

### 3.47.1 General

A `webResponse` object describes the response to an HTTP request [RFC7230]. The request itself is described by a `webRequest` object (§3.46).

> NOTE: This object is primarily useful to web analysis tools.

A `webResponse` object does not need to represent a valid HTTP response.

  

NOTE 2: This allows an analysis tool to describe a situation where a server produces an invalid response.

### 3.47.2 index property

Depending on the circumstances, a `webResponse` object either **MAY, SHALL NOT**, or **SHALL** contain a property named `index` whose value is the array index (§3.7.4) within `theRun.webResponses` (§3.14.22) of a `webResponse` object that provides additional properties for `thisObject`. We refer to the object in `theRun.webResponses` as the “cached object.”

If `thisObject` is an element of `theRun.webResponses`, then `index` **MAY** be present. If present, its value **SHALL** be the index of `thisObject` within `theRun.webResponses`.

Otherwise, if `theRun.webResponses` is absent, or if it does not contain a cached object for `thisObject`, then `index` **SHALL NOT** be present.

Otherwise (that is, if `thisObject` belongs to a result, and `theRun.webResponses` contains a cached object for `thisObject`), then `index` **SHALL** be present, and its value **SHALL** be the array index within `theRun.webResponses` of the cached object.

If `index` is present, `thisObject` **SHALL** take all properties present on the cached object. If `thisObject` contains any properties other than `index`, they **SHALL** equal the corresponding properties of the cached object.

NOTE 1: This allows a SARIF producer to reduce the size of the log file by reusing the same `webResponse` object in multiple results.

NOTE 2: For examples of the use of an `index` property to locate a cached object, see §3.38.2.

### 3.47.3 protocol property

A `webResponse` object **SHOULD** contain a property named `protocol` whose value is a string containing the name of the web protocol used in the response, found on the HTTP status line.

EXAMPLE: `"protocol": "HTTP"`

### 3.47.4 version property

A `webResponse` object **SHOULD** contain a property named `version` whose value is a string containing the version of the web protocol used in the response, found on the HTTP status line.

EXAMPLE: `"version": "1.1"`

### 3.47.5 statusCode property

A `webResponse` object **SHOULD** contain a property named `statusCode` whose value is an integer containing the status code that describes the result of the request, found on the HTTP status line.

EXAMPLE: `"statusCode": 200`

### 3.47.6 reasonPhrase property

A `webResponse` object **SHOULD** contain a property named `reasonPhrase` whose value is a string containing the textual description of the `statusCode` (§3.47.5) found on the HTTP status line.

EXAMPLE: `"reasonPhrase": "OK"`

If `noResponseReceived` (§3.47.9) is `true`, then `reasonPhrase` **SHOULD** instead contain a string describing the reason that no response was received.

  

### 3.47.7 headers property

A `webResponse` object **SHOULD** contain a property named `headers` whose value is an object (§3.6) whose property names are the names of the HTTP headers in the response (for example, `"Content-Type"`) and whose corresponding values are the header values (for example, `"text/plain; charset=ascii"`).

### 3.47.8 body property

A `webResponse` object **MAY** contain a property named `body` whose value is an `artifactContent` object (§3.3) containing the body of the response.

If the response body is entirely textual, `body.text` (§3.3.2) **SHOULD** be present. If present, it **SHALL** contain the response body, transcoded to UTF-8 if necessary.

> NOTE 1: The transcoding is required because all textual content in a SARIF log file is represented in UTF-8 (see §3.1).

> NOTE 2: If necessary, the character encoding actually used in the response can be deduced from the value of the `Content-Type` header (see §3.47.7), for example, `"text/plain; charset=ascii"`.

If the response body is entirely textual, `body.binary` (§3.3.3) **MAY** be present. If present, it **SHALL** contain the MIME Base64 encoding [RFC2045] of the body as it was actually transmitted.

If the response body consists partially or entirely of binary data, `body.binary` **SHALL** be present and **SHALL** contain the MIME Base64 encoding of the body. In this situation, `body.text` **SHALL** be absent.

### 3.47.9 noResponseReceived property

If no response to the HTTP request was received (for example, because of a network failure), the `webResponse` object **SHALL** contain a property named `noResponseReceived` whose value is a Boolean `true`. If a response was received, `noResponseReceived` **SHALL** either be present with the value `false`, or absent, in which case it defaults to `false`.

If `noResponseReceived` is `true`, then `reasonPhrase` (§3.47.6), which normally contains the reason phrase from the HTTP response line, **SHOULD** instead contain a string describing the reason that no response was received.

## 3.48 resultProvenance object

### 3.48.1 General

A `resultProvenance` object contains information about the how and when `theResult` was detected.

> NOTE: This information is useful to various human and automated participants in an engineering system. For example:
>
> - A build engineer might use the information to understand the specific tool invocation that produced the result, for example, if the violated rule should not have been configured to run at all.
> - A developer reviewing results might use the information to determine how long an issue has existed in the code.
> - A result management system might be responsible for associating logically identical results from one run to the next, making it possible for the developer to determine how long the result has existed. Such a result management system might populate this information.

  

### 3.48.2 firstDetectionTimeUtc property

A `resultProvenance` object **MAY** contain a property named `firstDetectionTimeUtc` whose value is a string in the format specified in §3.9, specifying the UTC date and time at which the result was first detected. It **SHOULD** specify the start time of the run in which the result was first detected, as opposed to, for example, the time within the run at which the result was actually generated.

> NOTE: Using the run's start time makes it possible to group together results that were first detected in the same run.

### 3.48.3 lastDetectionTimeUtc property

A `resultProvenance` object **MAY** contain a property named `lastDetectionTimeUtc` whose value is a string in the format specified in §3.9, specifying the UTC date and time at which the result was most recently detected. It **SHOULD** specify the start time of the run in which the result was most recently detected, as opposed to, for example, the time within the run at which the result was actually generated.

> NOTE: Using the run's start time makes it possible to group together results that were detected in the same run.

If `lastDetectionTimeUtc` is absent, its default value **SHALL** be determined as follows:

1. If `run.invocations` is present, and if the `startTimeUtc` property (§3.20.7) is present on any of the `invocation` objects (§3.20) in that array, then the default is the earliest of those times.
2. Otherwise, there is no default.

### 3.48.4 firstDetectionRunGuid property

A `resultProvenance` object **MAY** contain a property named `firstDetectionRunGuid` whose value is a GUID-valued string (§3.5.3) which **SHALL** equal the `automationDetails.guid` property (§3.14.3, §3.17.4) of the run in which `theResult` was first detected (either the current run or some previous run).

### 3.48.5 lastDetectionRunGuid property

A `resultProvenance` object **MAY** contain a property named `lastDetectionRunGuid` whose value is a GUID-valued string (§3.5.3) which **SHALL** equal the `automationDetails.guid` property (§3.14.3, §3.17.4) of the run in which `theResult` was most recently detected (either the current run or some previous run).

### 3.48.6 invocationIndex property

If `theRun.invocations` (§3.14.11) is present, a `resultProvenance` object **MAY** contain a property named `invocationIndex` whose value is the array index (§3.7.4) within the `invocations` property of the `invocation` object (§3.20) that describes the tool invocation as a result of which `theResult` was detected.

If `theRun.invocations` is absent, `invocationIndex` **SHALL** be absent.

> NOTE 1: The purpose of this property is to allow a result to be associated with the tool invocation that produced it.

If `invocationIndex` is absent and `theRun.invocations` is present and contains a single element, it **SHALL** default to 0; otherwise it **SHALL** default to -1, which indicates that the value is unknown (not set).

> NOTE 2: This provides a sensible default in the common case where there is only a single tool invocation in the run.

  

### 3.48.7 conversionSources property

Some analysis tools produce output files that describe the analysis run as a whole; we refer to these as "per-run" files. Some tools produce one or more output files for each result; we refer to these as "per-result" files. Some tools produce both per-run and per-result files.

A `resultProvenance` object **MAY** contain a property named `conversionSources` whose value is an array of zero or more unique (§3.7.3) `physicalLocation` objects (§3.29).

If `theResult` was produced by a converter, and if the analysis tool whose output was converted to SARIF produced any per-result files for this result, then the `physicalLocation` objects in the array **SHALL** specify the relevant portions of the per-result files for this result.

Otherwise (that is, if the `run` object was not produced by a converter, or if there were no per-run files for this result), then if `conversionSources` is present, its value **SHALL** be an empty array.

Per-run files are handled by the `conversion.analysisToolLogFiles` property (§3.22.4).

NOTE: This property is intended to be useful to developers of converters, to help them debug the conversion from the analysis tool's native output format to the SARIF format.

EXAMPLE: Given this analysis tool's output file:

```
<?xml version="1.0" encoding="UTF-8"?>
<problems>
  <problem>
    <file></file>
    <line>242</line>
    ...
    <problem_class ...>Assertions</problem_class>
    ...
    <description>Assertions are unreliable. ...</description>
  </problem>
</problems>
```

a SARIF converter might transform it into the following SARIF log file:

```
{
  ...
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "CodeScanner"
        }
      },
      "conversion": {  # A conversion object (see §3.22).
        ...
      },
      "results": [
        {
          "ruleId": "Assertions",
          "message": {
            "text": "Assertions are unreliable. ..."
          },
          ...
          "provenance": {              # See §3.27.29.
            "conversionSources": [     # An array of physicalLocation objects
              {                        # (§3.29).
                "artifactLocation": {  # See §3.29.3.
                  "uri": "CodeScanner.log",
                  "uriBaseId": "$LOGSROOT"
                },
                "region": {            # See §3.29.4.
                  "startLine": 3,
```

  

```
                  "startColumn": 3,
                  "endLine": 12,
                  "endColumn": 13,
                  "snippet": {
                    "text": "<problem>\n ... \n  </problem>"
                  }
                }
              }
            ],
            ...
          }
        }
      ]
    }
  ]
}
```

## 3.49 reportingDescriptor object

### 3.49.1 General

A `reportingDescriptor` object contains information that describes a "reporting item" generated by a tool. A reporting item is either a result produced by the tool's analysis (see §3.27), or a notification of a condition encountered by the tool (§3.58). We refer to this descriptive information as "reporting item metadata." When referring to the metadata that describes a result, we use the more specific term "rule metadata."

Some of the properties of the `reportingDescriptor` object are interpreted differently depending on whether the object represents a rule or a notification. The description of each property will specify any such differences.

### 3.49.2 Constraints

Either the `shortDescription` property (§3.49.9) or the `fullDescription` property (§3.49.10) or both **SHOULD** be present.

### 3.49.3 id property

A `reportingDescriptor` object **SHALL** contain a property named `id` whose value is a string. In the case of a rule, `id` **SHALL** contain a stable identifier for the rule and **SHOULD** be opaque. In the case of a notification, `id` does not need be a stable, opaque identifier ~~for the rule~~; it **MAY** be a user-readable identifier.

EXAMPLE: `"id": "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.

### ~~3.37.4~~3.49.4 ~~name~~deprecatedIds property

A ~~`rule`~~ `reportingDescriptor` object **MAY** contain a property named ~~`name`~~`deprecatedIds` whose value is an array of zero or more unique (§3.7.3) strings each of which contains an id (see §3.49.3) by which this reporting item was known in some previous version of the analysis tool.

> NOTE: This property is most useful for rules. It addresses the scenario where rule ids change from one version of a tool to the next. For example, a tool developer might decide that a rule is too general, covering too many concepts. In the next version of the tool, the tool developer might break this rule into a set of more specific rules.
>
> Now the result management system has the problem of matching results between the newer and the older versions of the tool. `deprecatedIds` solves this problem.
>
> EXAMPLE: In this example, version 1 of an analysis tool defines rule `CA1000`. A run of this tool finds two results. The result management system decides that neither result was previously detected, so it marks them as with `"baselineState": "new"` (§3.27.24), producing this log:

```
{
  "tool": {
    "driver": {
      "name": "CodeScanner",
      "version": "1",
      "rules": [
        {
          "id": "CA1000",
          ...
        }
      ]
    }
  },
  "results": [
    {
      "ruleId": "CA1000",
      "rule": {
        "index": 0
      },
      "baselineState": "new",
      ...
    },
    {
message      "ruleId": "CA1000",
      "rule": {
        "index": 0
      },
      "baselineState": "new",
      ...
    }
  ]
}
```

> The engineering team decides that these results are false positive, so they add in-source suppressions, for example (in C#):

```
[SuppressMessage("CA1000", ...)]
...
[SuppressMessage("CA1000", ...)]
```

> Now the tool developers decide that rule `CA1000` is too broad, so in version 2 of the tool, they divide it into two new rules, `CA1001` and `CA1002`. The engineering team runs the new tool, and the result management system performs result matching, producing this log:

  

```
{
  "tool": {
    "driver": {
      "name": "CodeScanner",
      "version": "2",
      "rules": [
        {
          "id": "CA1001",
          "deprecatedIds": [
            "CA1000"
          ],
          ...
        },
        {
          "id": "CA1002",
          "deprecatedIds": [
            "CA1000"
          ],
          ...
        }
      ]
    }
  },
  "results": [
    {
      "ruleId": "CA1001",
      "rule": {
        "index": 0
      },
      "baselineState": "existing",
      "suppressions": [
        {
          "kind": "inSource"
        }
      ],
      ...
    },
    {
      "ruleId": "CA1002",
      "rule": {
        "index": 1
      },
      "baselineState": "existing",
      "suppressions": [
        {
          "kind": "inSource"
        }
      ],
      ...
    }
  ]
}
```

There are a few things to notice:

- In `tool.driver.rules`, each of the new rules is associated with its id from the previous tool version.
- As a result, the analysis tool can determine that the in-source suppressions still apply, even though the rule ids have changed, so it correctly marks each result with `"kind": "inSource"`.

  

- Furthermore, the result management system can determine that these are the same results it saw in the previous run, so it correctly marks them with `"baselineState": "existing"`.

### 3.49.5 guid property

A `reportingDescriptor` object **MAY** contain a property named `guid` whose value is a GUID-valued string (§3.5.3) that uniquely identifies the descriptor.

### 3.49.6 deprecatedGuids property

A `reportingDescriptor` object **MAY** contain a property named `deprecatedGuids` whose value is an array of zero or more unique (§3.7.3) GUID-valued strings (§3.5.3) each of which was used by a previous version of the tool as the value of the `guid` property (§3.49.5(§) for this object.

### 3.49.7 name property

A `reportingDescriptor` object **MAY** contain a property named `name` whose value is a localizable string (§3.5.1) containing a rulean identifier that is understandable to an end user. If the `name` of a rule contains implementation details that change over time, a tool author might alter a rule's name (while leaving the stable `id` property (§3.49.3) unchanged).

NOTE 1: A rule name is suitable in contexts where a readable identifier is preferable and where the lack of stability is not a concern.

~~NOTE 2: The `name` property is represented as a `message` object rather than as a string because it is intended to be understandable to an end user, so tool vendors might want to localize it.~~

EXAMPLE:

```
{                              # A rule object
  "name": {
```

```
    "text": "SpecifyMarshalingForPInvokeStringArguments"
```

### 3.49.8 deprecatedNames property

A `reportingDescriptor` object **MAY** contain a property named `deprecatedNames` whose value is an array of zero or more unique (§3.7.3) localizable (§3.5.1) strings each of which was used by a previous version of the tool as the value of the `name` property (§3.49.7) for this object.

```
The array elements SHALL occur in the same order in every translation (§3.19.3  }
}
```

).

### ~~3.37.5~~3.49.9 shortDescription property

A ~~rule~~`reportingDescriptor` object **MAY** contain a property named `shortDescription` whose value is a ~~message~~localizable `multiformatMessageString` object (§3.12, §3.12.2) that provides 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.~~

reporting item. The `shortDescription` property **SHOULD** be a single sentence that is understandable when visible space is limited to a single line of text.

EXAMPLE:

```
{                              # A rulereportingDescriptor object
  "shortDescription": {
```

  

```
    "text": "Specify marshaling for P/Invoke string arguments"."
  }
}
```

## 3.37.63.49.10 fullDescription property

A `rule``reportingDescriptor` object **SHOULD** contain a property named `fullDescription` whose value is a `message`localizable `multiformatMessageString` object (§3.12, §3.12.2) that comprehensively describes the rulereporting item.

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

The first sentencebeginning of `fullDescription` (for example, its first sentence) **SHOULD** provide a concise description of the rulereporting item, 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 `shortDescription` (§3.49.9). Tools that do not construct `fullDescription` in this way **SHOULD** provide a value for `shortDescription`, because otherwise,.

> NOTE:The rationale for this guidance is that in the initial portionabsence of `shortDescription`, a viewer with limited display space might display a truncated version of `fullDescription` that a viewer displays where available space is limited , for example, the first sentence (if a sentence is identifiable), the first paragraph, or the first 100 characters. If this guidance is not followed, that truncated version might not be understandable.

## 3.49.11 messageStrings property

## 3.37.7 messageStrings property

A `rule``reportingDescriptor` object **MAY** contain a property named `messageStrings` whose value is a JSONan object (§3.6) consisting of a set of properties with arbitrary names.

The value of, each property **SHALL** be a plain text message stringof whose values is a localizable `multiformatMessageString` object (§3.12, §3.12.2). As with any message string, it **MAY** contain placeholders (§) and embedded links (§).

TheIf the `reportingDescriptor` object defines a rule, the set of property names appearing in the `messageStrings` property **SHALL** contain at least the set of strings which occur as values of `result.ruleMessageId``message.id` properties (§3.27.11, §3.11.10) in the current `run` object. The `messageStrings` property **MAY** contain additional properties whose names do not appear as the value of the `result.ruleMessageId``message.id` property for any `result` object in the `run`.

If the `reportingDescriptor` object describes a notification, the set of property names appearing in the `messageStrings` property **SHALL** contain at least the set of strings which occur as values of `notification.message.id` for any `notification` object in the run.

> NOTE: Additional properties are permitted in the `messageStrings` property for the convenience of tool vendors, who might find it easier to emit the entire set of messages supported by a ruledefined in the reporting metadata, rather than restricting it to those messages that happen to appear in the log file.

> EXAMPLE:

```
{                          # A rulereportingDescriptor object for a rule.
  "messageStrings": {
    "objectCreation":  {   # A multiformatMessageString object (§3.12).
      "text": "{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": {
        "text": "{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."
  }      }
}
```

### 3.37.8 richMessageStrings property

If a `rule` object contains a `messageStrings` property (§), it **MAY** also contain a property named `richMessageStrings` whose value is a JSON object (§) consisting of a set of properties with arbitrary names.

The value of each property **SHALL** be a rich text message string (§). As with any message string, it **MAY** contain placeholders (§) and embedded links (§).

The rules governing the set of property names appearing in the `richMessageStrings` property are the same as those for the `messageStrings` property.

SARIF consumers that cannot render rich text **SHALL** ignore the `richMessageStrings` property and use the `messageStrings` property instead. For this reason, every property name that appears in the `richMessageStrings` property **SHALL** also appear in the `messageStrings` property. SARIF consumers that can render rich text **SHOULD** use the `richMessageStrings` property, assuming they take appropriate measures to address security issues such as those discussed in §.

```
    }
}
```

### 3.37.93.49.12 helpUri property

A `rule``reportingDescriptor` object **MAY** contain a property named `helpUri` whose value is a localizable string (§3.5.1) containing the absolute URI [RFC3986] of the primary documentation for the rulereporting item.

> NOTE 1: The documentation might include examples, contact information for the rule authors, and links to additional information about the rule.

> NOTE 2: This property is localizable so that help information in different languages can be viewed at different URIs.

### 3.37.103.49.13 help property

A `rule``reportingDescriptor` object **MAY** contain a property named `help` whose value is a messagelocalizable `multiformatMessageString` object (§3.12, §3.12.2) which provides the primary documentation for the rulereporting item.

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

### 3.37.113.49.14 configurationdefaultConfiguration property

A `rule``reportingDescriptor` object **MAY** contain a property named `configuration``defaultConfiguration` whose value is a `ruleConfiguration``reportingConfiguration` object (§3.50).

If this property is absent, it **SHALL** be taken to be present, and its properties **SHALL** be taken to have the default values specified in §3.50.

### ~~3.37.12 properties property~~

~~A~~The rule- or notification-specific configuration parameters for a `reportingDescriptor`, if any, **SHALL NOT** be stored in its property bag (§3.8) Rather, they **SHALL** be stored in `defaultConfiguration.parameters` (§3.50.5).

### 3.49.15 relationships property

A `reportingDescriptor` object **MAY** contain a property named ~~`properties`~~`relationships` whose value is an array of zero or more unique (§3.7.3) `reportingDescriptorRelationship` objects (§3.53~~a property bag (§). This allows tools~~) each of which declares one or more directed relationships from `thisObject` to ~~include information about the rule that is not explicitly~~another `reportingDescriptor` object, which we refer to as `theTarget`, specified by `reportingDescriptorRelationship.target` (§3.53.2). The natures of the relationships between `thisObject` and `theTarget` are specified by `reportingDescriptorRelationship.kinds` (§3.53.3~~in the SARIF format.~~).

~~This property **SHALL NOT** be used to hold rule configuration information. Use the `ruleConfiguration.parameters` property (§) for that.~~

## ~~3.38~~3.50 ~~ruleConfiguration~~reportingConfiguration object

### ~~3.38.1~~3.50.1 General

A ~~`ruleConfiguration`~~`reportingConfiguration` object contains ~~rule configuration~~ the information in a `reportingDescriptor` (§3.49~~, that is, information about the rule~~) that a SARIF producer can modify at runtime, before executing its scan. We refer to the `reportingDescriptor` object whose configuration is established or modified by a `reportingConfiguration` object as `theDescriptor`.

When a `reportingConfiguration` object appears as the value of `theDescriptor.defaultConfiguration` (§3.49.14~~For example, if the rule~~ ), it specifies ~~a maximum source file line length, its~~ `theReportingDescriptor`'s default configuration. When a `reportingConfiguration` object appears as the value of `configurationOverride.configuration` (§3.51.3 ~~information might specify~~), it overrides the default values in the `reportingDescriptor` identified by `configurationOverride.descriptor` (§3.51.2~~maximum permitted line length.~~).

For an example, see §3.50.5.

### ~~3.38.2~~3.50.2 enabled property

A ~~`ruleConfiguration`~~`reportingConfiguration` object **MAY** contain a property named `enabled` whose value is a Boolean that specifies whether the ~~rule will be evaluated~~condition described by `theDescriptor` was checked for during the scan.

If this property is absent, it **SHALL** default to `true`.

> EXAMPLE: In this example, a tool allows the user to enable or disable rules or notifications:

```
SecurityScanner --disable "SEC4002,SEC4003" --enable SEC6012
```

### ~~3.38.3~~3.50.3 ~~defaultLevel~~level property

A ~~`ruleConfiguration`~~`reportingConfiguration` object **MAY** contain a property named ~~`defaultLevel`~~`level` whose value is one of the strings `"warning"`, `"error"`, `"note"`, or `"`~~`open`~~`none"`, with the same meanings as when those strings appear as the value of `result.level` (§3.27.10) or `notification.level` (§3.58.6).

  

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

The value of this property If theDescriptor describes a rule, then if level is present, it **SHALL** provide the value for the level property of any result object (§3.27) whose ruleIndex (§3.27.6) or rule property (§3.27.7ruleId property (§) refers to this rule configuration ), either explicitly supplied or inferred from its default, identifies theDescriptor and which does not itself specify a level property. For details of the configuration property resolution procedure, see §3.27.10 (which illustrates the procedure for the specific case of the result.level property).

If theDescriptor describes a notification, then if level is present, it **SHALL** provide the value for the level property of any notification object (§3.58) whose descriptor property (§3.58.2) identifies theDescriptor and which does not itself specify a level property.

EXAMPLE: In this example, a tool allows the user to override a rule'srule or notification's default level:

```
WebScanner --level "WEB1002:error,WEB1005:warning"
```

### 3.38.41.1.1 parameters property

### 3.50.4 rank property

A reportingConfiguration object **MAY** contain a property named rank whose value is a number between 0.0 and 100.0 inclusive, with the same interpretation as the value of the result.rank (§3.27.25).

If rank is absent, it **SHALL** default to -1.0, which indicates that the value is unknown (not set).

If theDescriptor describes a rule, then if rank is present, it **SHALL** provide the value for the rank property of any result object (§3.27) whose ruleIndex (§3.27.6) or rule property (§3.27.7ruleConfiguration), either explicitly supplied or inferred from its default, identifies theDescriptor and which does not itself specify a rank property.

rank is not applicable to notifications.

### 3.50.5 parameters property

A reportingConfiguration object **MAY** contain a property named parameters whose value is a property bag (§3.8). This allows a reportingDescriptor object (§3.49rule) to define configuration information that is specific to that ruledescriptor.

EXAMPLE: In this example, a rule that specifies the maximum permitted source line length is parameterized by the maximum length.

```
{                                   # A rulereportingDescriptor object
(§3.49.)).
  "id": "SA2707",
  "name": {
    "text": "LimitSourceLineLength"
  },
  "shortDescription": {
    "text": "Limit source line length for readability."
  },
  "configurationdefaultConfiguration": {
    "enabled": true,
    "defaultLevellevel": "warning",
    "parameters": {
      "maxLength": 120
    }
  }
}
```

  

The rule provides a default value, but the tool allows the user to override it:

```
StyleScanner *.c --rule-config "SA2707:maxLength=80"
```

## 3.51 configurationOverride object

### 3.51.1 General

A `configurationOverride` object modifies the effective runtime configuration of a specified `reportingDescriptor` object (§3.49), which we refer to as `theDescriptor`.

NOTE: Together with `toolComponent.rules` (§3.19.23), the `configurationOverride` object allows the SARIF consumer to determine exactly how the tool's analysis rules were configured during the run. This is useful in compliance scenarios where, for example, an auditor might want to confirm that a particular rule was reconfigured from a warning to an error. It might also be useful for reproducing a run.

The `configurationOverride` object's `descriptor` property (§3.51.2) identifies `theDescriptor`. Its `configuration` property (§3.51.3) overrides the values specified in `theDescriptor.defaultConfiguration` (§3.49.14).

EXAMPLE: In this example, rule `CA2101` is treated as a warning rather than an error.

```
{                                                 # A run object (§3.14).
  "tool": {                                       # See §3.14.6.
    "driver": {                                   # See §3.18.2.
      "name": "CodeScanner",
      "rules": [                                  # See §3.19.23.
        {                                         # A reportingDescriptor object
          "id": "CA2101",                         #   (§3.49).
          "defaultConfiguration": {
            "level": "error"
          }
        }
      ]
    }
  },

  "invocations": [                                # See §3.14.11.
    {                                             # An invocation object (§3.20).
      "ruleConfigurationOverrides": [             # See §3.20.5.
        {                                         # A configurationOverride object
                                                  #   (§3.51).
          "descriptor": {                         # See §3.51.2.
            "index": 0
          },
          "configuration": {                      # See §3.51.3.
            "level": "warning"
          }
        }
      ],
      ...
    }
  ]
}
```

### 3.51.2 descriptor property

A `configurationOverride` object **SHALL** contain a property named `descriptor` whose value is a `reportingDescriptorReference` object (§3.52) that identifies the `reportingDescriptor` (§3.49) whose runtime configuration is to be modified, which we refer to as `theDescriptor`.

  

### 3.51.3 configuration property

A `configurationOverride` object **SHALL** contain a property named `configuration` whose value is a `reportingConfiguration` object (§3.50) each of whose properties overrides the corresponding property in `theDescriptor.defaultConfiguration` (§3.49.14). If any property of `configuration` is absent, the corresponding property of `theDescriptor.defaultConfiguration` is respected.

## 3.52 reportingDescriptorReference object

### 3.52.1 General

A `reportingDescriptorReference` object identifies a particular `reportingDescriptor` object (§3.49), which we refer to as `theDescriptor`, among all `reportingDescriptor` objects defined by `theTool`, including those defined by `theTool.driver` (§3.18.2) and `theTool.extensions` (§3.18.3).

In some cases, there is no `reportingDescriptor` object associated with a `reportingDescriptorReference` object. In that case, the `reportingDescriptorReference` object **SHALL** contain only the `id` property (§3.52.4), and `theDescriptor` does not exist.

EXAMPLE: In this example, a tool emits a tool execution notification that refers to a rule. The tool does not provide rule metadata. Therefore, `associatedRule` (§3.58.3) contains only an `id` property, whose value is the id of the rule that failed. Similarly, the tool does not provide metadata about its notifications, so `"descriptor"` (§3.58.2) contains only the id of the notification.

```
{                                                  # An invocation object (§3.20).
  "toolExecutionNotifications": [                  # See §3.20.21.
    {                                              # A notification object (§3.58).
      "descriptor": {                              # See §3.58.2.
        "id": "CTN9999"
      },
      "associatedRule": {                          # See §.3.58.3
        "id": "C2001"
      },
      "level": "error",
      "message": {
        "text": "Exception evaluating rule 'C2001'. Rule disabled;
                 run continues."
      }
    }
  ]
}
```

### 3.52.2 Constraints

If metadata is present, at least one of `index` (§3.52.5) and `guid` (§3.52.6) **SHALL** be present. If both are present, they **SHALL** identify the same `reportingDescriptor` object (§3.49).

### 3.52.3 reportingDescriptor lookup

`theDescriptor` **SHALL** be located within the `toolComponent` object (§3.19) identified by the `toolComponent` property (§3.52.7), which we refer to as `theComponent`. The procedure for looking up a `toolComponent` from a `toolComponentReference` is described in §3.54.2.

`theDescriptor` **SHALL** be located either within `theComponent.rules` (§3.19.23) or `theComponent.notifications` (§3.19.24), according to this table:

  

| If the reportingDescriptorReference occurs in: | … then theDescriptor is an element of: |
|---|---|
| invocation.ruleConfigurationOverrides (§3.20.5) | rules |
| invocation.notificationConfigurationOverrides (§0) | notifications |
| result.rule (§3.27.7) | rules |
| notification.descriptor (§3.58.2) | notifications |
| notification.associatedRule (§3.58.3) | rules |

### 3.52.4 id property

A reportingDescriptorReference object **MAY** contain a property named id whose value is a hierarchical string (§3.5.4) that either equals theDescriptor.id (§3.49.3) or equals theDescriptor.id plus one additional hierarchical component.

> NOTE: This property does not participate in the lookup, but its presence improves the readability of the log file at the expense of increased file size.

If id is absent and theResult.ruleId (§3.27.5) is present, then id **SHALL** default to theResult.ruleId. If both are present, they **SHALL** be equal.

For more information about the semantics of id when theDescriptor is a rule, in particular the usage of the hierarchical components of id, see the description of result.ruleId (§3.27.5).

> EXAMPLE: In this example, the first result object is valid because rule.id (inherited from ruleId) equals theDescriptor.id. The second result object is also valid because rule.id (this time specified directly) equals theDescriptor.id plus one additional hierarchical component ("ghi"). The third result object is invalid because theDescriptor.id is not a "component-wise" prefix of rule.id. The fourth result object is invalid because ruleId does not equal rule.id.

```
{                              # A run object (§3.14).
  "tool": {                    # See §3.14.6.
    "driver": {                # See §3.18.2.
      "name": "CodeScanner",
      "rules": [               # See §3.19.23.
        {                      # A reportingDescriptor object (§3.49).
          "id": "abc/def",     # See §3.49.3.
          ...
        },
        ...
      ]
    }
  },
  "results": [                 # See §3.14.23.
    {                          # A result object (§3.27).
      "ruleId": "abc/def",     # See §3.27.5.
      "rule": {
        "index": 0
      },
    },
    {
      "rule": {
        "id": "abc/def/ghi",
        "index": 0
      }
    },
```

  

```
    {
      "rule": {
        "id": "abc/defg",      # INVALID: theDescriptor.id is not a
        "index": 0             #   "component-wise" prefix of id.
      }
    },
    {
      "ruleId": "abc/def",
      "rule": {
        "id": "abc/defg/hij", # INVALID: Not equal to ruleId.
        "index": 0
      }
    }
  ]
}
```

#### 3.52.5 index property

A `reportingDescriptorReference` object **MAY** contain a property named `index` whose value is the array index (§3.7.4) into `theComponent.rules` (§3.19.23) or `theComponent.notifications` (§3.19.24), according to the table in §3.52.3.

> EXAMPLE 1: In this example, there is more than one rule with id `CA1711`. `index` uniquely specifies the relevant rule, whether or not there are multiple rules with the same id.

```
{                              # A run object (§3.14).
  "tool": {                    # See §3.14.6.
    "driver": {                # See §3.18.2.
      "name": "CodeScanner",
      "rules": [               # See §3.19.23.
        {                      # A reportingDescriptor object (§3.49).
          "id": "CA1711",      # See §3.49.3.
          ...
        },
        {                      # Another reportingDescriptor with the same id.
          "id": "CA1711",      #  rule.index points to this one.
          ...
        }
      ]
    }
  },
  "results": [                 # See §3.14.23.
    {                          # A result object (§3.27).
      "ruleId": "CA1711",      # See §3.27.5.

                               # A reportingDescriptorReference object.
      "rule": {
        "index": 1
      }
    }
  ]
}
```

If `index` is absent and `theResult.ruleIndex` (§3.27.6) is present, `index` **SHALL** default to `theResult.ruleIndex`. If both are present, they **SHALL** be equal.

#### 3.52.6 guid property

A `reportingDescriptorReference` object **MAY** contain a property named `guid` whose value is a GUID-valued string (§3.5.3) equal to `theDescriptor.guid` (§3.49.5).

  

### 3.52.7 toolComponent property

A `reportingDescriptorReference` object **MAY** contain a property named `toolComponent` whose value is a `toolComponentReference` object (§3.54) that identifies `theComponent`.

If `toolComponent` is absent, `theComponent` shall be taken to be `theTool.driver` (§3.18.2).

## 3.53 reportingDescriptorRelationship object

### 3.53.1 General

A `reportingDescriptorRelationship` object specifies one or more directed relationships from one `reportingDescriptor` object (§3.49), which we refer to as `theSource`, to another one, which we refer to as `theTarget`.

`reportingDescriptorRelationship` objects appear as elements of the `reportingDescriptor.relationships` array (§3.49.15). The `reportingDescriptor` object containing this property is `theSource`.

`reportingDescriptorRelationship` objects are useful in various scenarios:

1. In relating analysis rules to taxonomic categories (“taxa”; see §3.19.3).

   EXAMPLE 1: In this example, the definition of rule `CA1000` states that every result that violates this rule falls into the taxonomic category (“taxon”) specified by ID 327 of the Common Weakness Enumeration [CWE™]:

```
{                               # A run object (§3.14).
  "tool": {                     # See §3.14.6.
    "driver": {                 # See §3.18.2.
      "name": "CodeScanner",
      "rules": [                # See §3.19.23.
        {                       # A reportingDescriptor object (§3.49).
          "id": "CA1000",
          "relationships": [
            {                   # A reportingDescriptorRelationship object.
              "target": {       # See §3.53.2.
                "id": "327",
                "guid": "33333333-0000-0000-0000-111111111111",
                "toolComponent": {
                  "name": "CWE",
                  "guid": "33333333-0000-0000-0000-000000000000",
                }
              },
              "kinds": [
                "superset"
              ]
            }
          ]
        }
      ]
    }
  },

  "taxonomies": [
      "name": "CWE",
      "guid": "33333333-0000-0000-0000-000000000000",
      ...
      "taxa": [
        {
          "id": "327",
          "guid": "33333333-0000-0000-0000-111111111111",
          "name": "BrokenOrRiskyCryptographicAlgorithm",
```

  

```
            ...
          },
          ...
        ]
      }
    ],

    ...
}
```

2. In relating one analysis rule to another.

   EXAMPLE 2: In this example, the definition of rule `CA1000` states that every violation of this rule will lead to a violation of rule `CA2000`.

```
{                              # A run object (§3.14).
  "tool": {                    # See §3.14.6.
    "driver": {                # See §3.18.2.
      "name": "CodeScanner",
      "rules": [               # See §3.19.23.
        {                      # A reportingDescriptor object (§3.49).
          "id": "CA1000",
          "guid": "11111111-0000-0000-0000-000000000001"
          "relationships": [
            {                  # A reportingDescriptor object.
              "target": {      # See §3.53.2.
                "id": "CA2000",
                "guid": "11111111-0000-0000-0000-000000000002",
              },
              "kinds": [
                "willFollow"
              ]
            }
          ]
        },
        {
          "id": "CA2000",
          "guid": "11111111-0000-0000-0000-000000000002"
          ...
        }
      ]
    }
  },
  ...
```

## 3.53.2 target property

A `reportingDescriptorRelationship` object **SHALL** contain a property named `target` whose value is a `reportingDescriptorReference` object which identifies `theTarget` (see §3.53.1).

## 3.53.3 kinds property

A `reportingDescriptorRelationship` object **MAY** contain a property named `kinds` whose value is an array of one or more unique (§3.7.3) strings each of which specifies a relationship between `theSource` and `theTarget` (see §3.53.1). If `kinds` is absent, it **SHALL** default to `[ "relevant" ]` (see below for the meaning of `"relevant"`).

When possible, SARIF producers **SHOULD** use the following values, with the specified meanings.

- `"equal"`: `theTarget` identifies essentially the same set of items as does `theSource` (for example, a taxonomic category that identifies the same set of results as this rule).

- `"superset"`: `theTarget` identifies a superset of the items identified by `theSource` (for example, a taxonomic category that identifies a superset of the results identified by this rule).
- `"subset"`: `theTarget` identifies a subset of the items identified by `theSource` (for example, a taxonomic category that identifies a subset of the results identified by this rule)
- `"disjoint"`: The sets of items identified by `theTarget` does not intersect with the set of items identified by `theSource`.
- `"incomparable"`: The sets of items identified by `theTarget` intersects with the set of items identified by `theSource` but is neither a superset nor a subset.
- `"canFollow"`: Items identified by `theTarget` can be caused by, or occur downstream of, items identified by `theSource`.
- `"canPrecede"`: Items identified by `theSource` can be caused by, or occur downstream of, items identified by `theTarget`.
- `"willFollow"`: Items identified by `theTarget` will be caused by, or occur downstream of, items identified by `theSource`.
- `"willPrecede"`: Items identified by `theSource` will be caused by, or occur downstream of, items identified by `theTarget`.
- `"relevant"`: `theTarget` is relevant to `theSource` in a way not covered by other relationship kinds.

If none of these values are appropriate, a SARIF producer **MAY** use any value.

NOTE 1: Although `"relevant"` is a catch-all for any relationship not described by the other values, a producer might still wish to define its own more specific values.

NOTE 2: The values `"equal"` and `"superset"` are special in that they allow certain elements of `result.taxa` (§3.27.8) to be elided. See §3.27.8, paragraph 2, for more information on this point.

### 3.53.4 description property

A `reportingDescriptorRelationship` object **MAY** contain a property named `description` whose value is a `message` object (§3.11) that describes the relationship.

## 3.54 toolComponentReference object

### 3.54.1 General

A `toolComponentReference` object identifies a particular `toolComponent` object (§3.19), either `theTool.driver` (§3.18.2) or an element of `theTool.extensions` (§3.18.3). We refer to the identified `toolComponent` object as `theComponent`.

### 3.54.2 toolComponent lookup

If neither `index` (§3.54.4) nor `guid` (§3.54.5) is present, `theComponent` **SHALL** be `theTool.driver` (§3.18.2).

If `index` is present, `theComponent` **SHALL** be the object at array index `index` within `theTool.extensions` (§3.18.3).

If `index` is absent and `guid` is present, `theComponent` **SHALL** be either `theTool.driver` or an element of `theTool.extensions`, whichever one has a matching `guid` property.

### 3.54.3 name property

A `toolComponentReference` object **MAY** contain a property named `name` whose value is a string equal to `theComponent.name` (§3.19.8).

  

NOTE: This property does not participate in the lookup, but its presence improves the readability of the log file at the expense of increased file size.

### 3.54.4 index property

If `theComponent` is an element of `theTool.extensions` (§3.18.3), a `toolComponentReference` object **MAY** contain a property named `index` whose value is the array index (§3.7.4) of that element. Otherwise, `index` SHALL be absent.

### 3.54.5 guid property

A `toolComponentReference` object **MAY** contain a property named `guid` whose value is a GUID-valued string (§3.5.3) equal to `theComponent.guid` (§3.19.6).

## ~~3.39~~3.55 fix object

### ~~3.39.1~~3.55.1 General

A `fix` object represents a proposed fix for the problem indicated by ~~the containing `result` object (§).~~ `theResult`. It specifies a set of ~~files~~artifacts to modify. For each ~~file~~artifact, it specifies regions to remove, and provides new ~~file~~ content to insert.

EXAMPLE:

```
{                                   # A result object (§3.27).
  "fixes": [                        # See §3.27.30.
    {                               # A fix": { object.
      "description": {              # See §3.55.2.
        "text": "Private member names begin with '_'"
      },
      "fileChanges": [      "artifactChanges": [          # See §3.55.3.
        {                           # A fileChange         {
# An artifactChange object (§3.56).
          ...
        }
      ]
    }
  ],
  ...
}
```

### ~~3.39.2~~3.55.2 description property

A `fix` object **SHOULD** contain a property named `description` whose value is a `message` object (§3.11) that describes the proposed fix.

NOTE: The purpose of the `description` property is to enable a SARIF viewer to present the proposed fix to the end user.

EXAMPLE:

```
"fix": {
  "description": {
    "text": "Combine declaration and initialization of variable 'x'."
  },
  ...
}
```

  

### 3.39.33.55.3 fileChangesartifactChanges property

A `fix` object **SHALL** contain a property named `fileChanges``artifactChanges` whose value is an array of one or more unique (§3.7.3`fileChange`) `artifactChange` objects (§).

## 3.40 fileChange object

### 3.40.11.1.1 General

3.56A `fileChange` object represents a change) each of which describes the changes to a single fileartifact that are necessary to effect the fix.

EXAMPLE:

{NOTE: `artifactChanges` is an array because a fix might require changes to multiple artifacts.

The array elements **SHALL** refer to distinct artifacts.

EXAMPLE 1: In this example, two `artifactChange` objects make identical changes (commenting out the first line) in two distinct C-language files, `src/a.c` and `src/b.c`.

```
{                                         # A fix object.
  "artifactChanges": [
    {                                     # An artifactChange object (§3.56).
  "fileChanges": [        "artifactLocation": {          # See §3.56.2.
    {
      "fileLocation": {           # See §.
        "uri": "src/a.hc"
      },
      "replacements": [                   # See §3.56.3.
        {                                 # A replacement object (§3.57).
          "deletedRegion": {              # See §3.57.3.
            "startLine": 1,
            "startColumn": 1,
            "endColumn": 1
          },
          "insertedContent": {            # See §3.57.4.
            "text": "// "
          }
        }
      }
    },
    {
      "artifactLocation": {
        "uri": "src/b.c"
      },
      "replacements": [
        {
          "deletedRegion": {
            "startLine": 1,
            "startColumn": 1,
            "endColumn": 1
          },
          "insertedContent": {
            "text": "// "
          }
        }
      }
    }
  ]
}
```

EXAMPLE 2: This example represents invalid SARIF because the two `artifactChange` objects refer to the same file, `src/a.c`. It is invalid even though the `artifactChange` objects are distinguished by their `replacements` properties.

```
{                                    # A fix object.
  "artifactChanges": [
    {                                # An artifactChange object (§3.56).
      "artifactLocation": {          # See §3.56.2.
        "uri": "src/a.c"
      },
      "replacements": [              # See §3.56.3.
        {                            # A replacement object (§3.57).
          "deletedRegion": {         # See §3.57.3.
            "startLine": 1,
            "startColumn": 1,
            "endColumn": 1
          },
          "insertedContent": {       # See §3.57.4.
            "text": "// "
          }
        }
      }
    },
    {
      "artifactLocation": {
        "uri": "src/a.c"             # Invalid: refers to the same file.
      },
      "replacements": [
        {
          "deletedRegion": {
            "startLine": 2,          # Invalid even though it affects a
            "startColumn": 1,        #  different line.
            "endColumn": 1
          },
          "insertedContent": {
            "text": "// "
          }
        }
      }
    }
  ]
}
```

# 3.56 artifactChange object

## 3.56.1 General

An `artifactChange` object represents a change to a single artifact.

EXAMPLE:

```
{                             # A fix object (§3.55).
  "artifactChanges": [        # See §3.55.3.
    {
      "artifactLocation": {   # See §3.56.2.
        "uri": "a.h"
      },
      "replacements": [       # See §3.56.3.
        {                     # A replacement object (§3.57).
          ...
        },
        {                     # Another replacement object.
          ...
```

```
        }
      ]
    }
  ]
}
```

### ~~3.40.2~~3.56.2 ~~fileLocation~~artifactLocation property

~~A `fileChange`~~An `artifactChange` object **SHALL** contain a property named ~~`fileLocation`~~`artifactLocation` whose value is ~~a `fileLocation`~~an `artifactLocation` object (§3.4) that represents the location of the ~~file~~artifact.

### ~~3.40.3~~3.56.3 replacements property

~~A `fileChange`~~An `artifactChange` object **SHALL** contain a property named `replacements` whose value is an array of one or more `replacement` objects (§3.57~~),~~) each of which represents the replacement of a single region of the ~~file~~artifact specified by the ~~`fileLocation`~~`artifactLocation` property (§3.56.2).

## ~~3.41~~3.57 replacement object

### ~~3.41.1~~3.57.1 General

A `replacement` object represents the replacement of a single region of ~~a file~~an artifact. If the region's length is zero, it represents an insertion point.

If a replacement object specifies both the removal of a region by means of the `deletedRegion` property (§3.57.3) and the insertion of new ~~file~~ content by means of the `insertedContent` property (§3.57.4), then the effect of the replacement **SHALL** be as if the removal were performed before the insertion.

If a single ~~`fileChange`~~`artifactChange` object (§3.56) 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 (§3.56.3). The `deletedRegion` property of each `replacement` object **SHALL** specify the location of the replacement in the unmodified ~~file~~artifact.

> EXAMPLE 1: Suppose ~~a `fileChange`~~an `artifactChange` object contains a `replacements` property whose value is the following array of `replacement` objects:

```
"fileChanges""artifactChanges": [
  {
    "deletedRegion": {
      "byteOffset": 12,
      "byteLength": 5
    },
    "insertedContent": {
      "binary": "ZXhhbXBsZQ=="
    }
  },
  {
    "deletedRegion": {
      "byteOffset": 20,
      "byteLength": 3
    }
  },
  {
    "deletedRegion": {
      "byteOffset": 312,
      "byteLength": 0
    },
    "insertedContent": {
      "binary": "ZXhhbXBsZQ=="
```

  

```
      }
    }
]
```

The first replacement object removes 5 bytes starting at offset 12; that is, it removes bytes 12–16. Then it inserts the 7 bytes specified by the MIME Base64-encoded string in the insertedContent.binary property 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. of the contents after the first change. Since the insertedContent property is absent, no content is inserted in place of the deleted bytes.

In the third replacement object, the length of the region specified by the deletedRegion property is zero, so the region represents an insertion point. The 7 bytes specified by the insertedContent.binary property are inserted at offset 312 with respect to the unmodified ~~file~~artifact.

A replacement object can represent either a textual replacement or a binary replacement, depending on whether the deletedRegion property (§3.57.3) specifies a text region (§3.30.2) or a binary region (§3.30.3).

EXAMPLE 2: In this example, the replacements property specifies a replacement in a text file.

```
"replacements": [
  {
    "deletedRegion": { # The region object represents a text region (§3.30.2).
      "startLine": 12,
      "startColumn": 5,
      "endColumn": 9
    },
    "insertedContent": {
      "text": "example" # The insertedContent property contains a text
    }                  # property
    }                          # instead of a binary property.
  }
]
```

When performing a replacement in a text ~~file~~artifact, the SARIF producer **SHOULD** specify a text replacement rather than a binary replacement. This allows the SARIF producer to specify the region without regard to whether the ~~file~~artifact starts with a byte order mark (BOM).

## ~~3.41.2~~3.57.2 Constraints

If the deletedRegion property (§3.57.3) specifies a text region (§3.30.2) and the insertedContent property (§3.57.4) is present, then the insertedContent property **SHOULD** contain a text property (§3.3.2).

If the deletedRegion property specifies a binary region (§3.30.3) and the insertedContent property is present, then the insertedContent property **SHALL** contain a binary property (§3.3.3).

Although it is possible to construct a replacement object that neither removes nor adds any content, a replacement object **SHOULD** have a material effect on the target artifact, either because deletedRegion denotes a non-empty region to delete, or because insertedContent specifies non-empty content to insert, or both.

## ~~3.41.3~~3.57.3 deletedRegion property

A replacement object **SHALL** contain a property named deletedRegion whose value is a region object (§3.30) specifying the region to delete.

  

If the length of the region specified by `deletedRegion` is zero, then `deletedRegion` specifies an insertion point, and the SARIF consumer performing the replacement **SHALL NOT** remove any file content.

### 3.41.43.57.4 insertedContent property

A `replacement` object **MAY** contain a property named `insertedContent` whose value is a `fileContent`an `artifactContent` object (§3.3) that specifies the content to insert in place of the region specified by the `deletedRegion` property (or at the point specified by `deletedRegion`, if `deletedRegion` has a length of zero and therefore specifies an insertion point).

If the inserted content is specified as text, the text **SHALL** be transcoded from UTF-8 (the encoding of all text in all SARIF log files) to the encoding of the target artifact before being inserted.

> NOTE: This implies that a text fix cannot be safely applied unless the target artifact's encoding is known.

If `insertedContent` is absent or its properties specify content whose length is zero, the SARIF consumer performing the replacement **SHALL NOT** insert any content.

## 3.423.58 notification object

### 3.42.13.58.1 General

A `notification` object describes a condition encountered induring the courseexecution of running an analysis tool which is relevant to the operation of the tool itself, as opposed to being relevant to a filean artifact being analyzed by the tool. Conditions relevant to filesartifacts being analyzed by a tool are represented by `result` objects (§3.27).

### 3.42.23.58.2 iddescriptor property

A `notification` object **MAYSHOULD** contain a property named `id``descriptor` whose value is a `reportingDescriptorReference` object (§3.52string containing an identifier for the condition that was encountered.

NOTE: In contrast to rule identifiers (see `rule.id`, §), which must be stable and opaque, ) that identifies this 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.

If the `reportingDescriptor` object (§3.49ruleId) `theDescriptor` to which `descriptor` refers exists (that is, if `theTool` contains a `reportingDescriptor` object that describes this notification), then `descriptor` **SHOULD** refer to `theDescriptor`.

> NOTE: If `theDescriptor` exists but `descriptor` does not refer to it, a SARIF consumer will not be able to locate the metadata for this notification.

### 3.42.33.58.3 associatedRule 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``associatedRule` whose value is a `reportingDescriptorReference` object (§3.52string containing the stable, unique identifier of the rule (§).

If there is more than one rule with the desired id, and if) that identifies the containing `run` object (§) contains a `resources.rules` property (§, §), then instead of containing the rule id, `ruleId` **SHALL** contain a string that equals one of the property names in `resources.rules`. To improve the readability of the log file, this property name **SHOULD** be formed by appending a suffix to the rule id. In this case, the "id" property (§) of the specified rule object (§) **SHALL** contains the actual rule idrule.

EXAMPLE: In this example, there is more than one rule with id CA1711. The SARIF producer sets ruleId to a value that specifies which of the rules with that id is meant. That value is formed by appending the suffix "-1" to the rule id. The rule id is specified by resources.rules["CA1711-1"].idassociatedRule.index uniquely specifies the relevant rule.

```
{                                                      # A run object (§3.14).
  "tool": {                                            # See §3.14.6.
    "driver": {                                        # See §3.18.2.
      "name": "CodeScanner",
      "rules": [                                       # See §3.19.23.
        {                                              # A runreportingDescriptor object
(§3.49).
          "id": "CA1711",
          ...
        },
        {                                              # Another reportingDescriptor object
          "id": "CA1711",                              #  with the same id. associatedRule.id
          ...                                          #  identifies this one.
        }
      ]
    }
  },
  "invocations": [                                      # See §3.14.11.
    {                                                   # An invocation object (§3.20).
      "configurationNotifications": [ toolConfigurationNotifications": [ #
See §3.20.22.
        {                                               # A notification object (§3.58.).
          "descriptor": {
            "id": "CFG0001","
          },
          "message": {
            "text": "Rule configuration is missing."
          },
          "ruleId": "CA1711-1"                          # Specifies a property name within "rules".
        }
      ],
    }
  ],

  "resources": {                                        # See §.
    "rules": {                                          # See §.
      "CA1711-1": {                                     # A rule object (§).
          "associatedRule": {
            "id": "CA1711",
        ...     "index": 1
      },    }
      "CA1711-2": {                                     # Another rule object with the same id.
        "id": "CA1711",
        ...
      }
    }
  }
}
physicalLocation        }
      ],
      ...
    }
  ]
}
```

  

### 3.42.43.58.4 locations property

If the condition described by the `notification` object is relevant to a particular file locationone or more locations, the `notification` object SHOULD**MAY** contain a property named `physicalLocationlocations` whose value is an array of zero or more unique (§3.7.3a `physicalLocation` object (§) `location` objects (§3.28) that identifies the relevant locationidentify those locations.

### 3.42.53.58.5 message property

A `notification` object **SHALL** contain a property named `message` whose value is a `message` object (§3.11) that describes the condition that was encountered. See §3.11.7

NOTE: The for the procedure for looking up a message string from a `message` object, in the particular, for the case where the `message` object occurs as the value of `notification.message` property will typically not contain a `richText` (§) or `richMessageId` (§) property because tool notifications typically appear on the console, where rich text is not supported.

### 3.42.63.58.6 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 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.
- `"none"`: This is a trace notification (typically, debug output from the tool).

If `level` is absent, it **SHALL** default to the value determined by the procedure defined for `result.level` (§3.27.10pass`level` property is absent, it **SHALL** be considered equivalent to the value `"warning"`.

), except throughout the procedure, replace `ruleConfigurationOverrides` with `notificationConfigurationOverrides`.

Analysis tools **SHOULD** treat notifications whose `level` property is `"error"` as failures and treat the entire run as having failed (for example, by settings the exit code to the value that the tool uses to indicate failure, typically a non-zero value).

Because a notification whose `level` property is `"error"` describes a failed run, an analysis tool **SHALL NOT** override the severity of such a notification.

### 3.42.73.58.7 threadId property

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

### 3.42.83.58.8 timetimeUtc property

A `notification` object **MAY** contain a property named `timetimeUtc` whose value is a string in the format specified §3.9, specifying the UTC date and time at which the analysis tool generated the notification. The string **SHALL** be in the format specified by (§).

  

### ~~3.42.9~~3.58.9 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 (§3.59).

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

### ~~3.42.10 properties property~~

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

## ~~3.43~~3.59 exception object

### ~~3.43.1~~3.59.1 General

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

### ~~3.43.2~~3.59.2 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.

### ~~3.43.3~~3.59.3 message property

An `exception` object **SHOULD** contain a property named `message` whose value is a string ~~containing a plain text message 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 ~~would~~might populate `message` ~~from~~with the string returned from the `what()` method of any object derived from `std::exception`.

EXAMPLE 2: C#: The tool ~~would~~might populate `message` with the value returned from the ~~`Message` property~~`ToString()` method of ~~any object derived from~~ the `System.Exception` object, or (less informatively) from that object's `Message` property.

~~NOTE: The `exception.message` property is not a message object (§) because exception messages, appearing as they do in typical languages and operating systems, are inherently plain text, and require no arguments (§).~~

### ~~3.43.4~~3.59.4 stack property

An `exception` object **MAY** contain a property named `stack` whose value is a `stack` object (§3.44) that describes the sequence of function calls leading to the exception.

  

### 3.43.53.59.5 innerExceptions property

An `exception` object **MAY** contain a property named `innerExceptions` whose value is an array of onezero or more `exception` objects, each of which is considered 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.

# 4 External property file format

## 4.1 General

External property files (see §3.15.2) conform to a schema distinct from that of the root file. External property files contain information that makes it possible for a consumer to determine which properties are contained in the file, to parse their contents, and to associate the external properties with the run to which they belong.

An external property file **SHALL** contain one or more externalized properties. A SARIF consumer **SHALL** treat the value of an externalized property exactly as if it had appeared inline in the root file as the value of the corresponding property.

## 4.2 External property file naming convention

The file name of an external property file **SHOULD** end with the extension "`.sarif-external-properties`".

EXAMPLE 1: `scan-results.sarif-external-properties`

The file name **MAY** end with the additional extension "`.json`".

EXAMPLE 2: `scan-results.sarif-external-properties.json`

## 4.3 externalProperties object

### 4.3.1 General

The top-level element of an external property file **SHALL** be an object which we refer to as an `externalProperties` object.

EXAMPLE: In this example, `run.artifacts` and `run.properties` have been externalized to a file with these contents. Note that `run.properties` has been externalized under the property name `externalizedProperties`, as explained in §3.15.3.

```
{                              # An externalProperties object
  "$schema":                   # See §4.3.2.
    "https://raw.githubusercontent.com/oasis-tcs/sarif-
spec/master/Schemata/sarif-external-property-file-schema-2.1.0.json",

  "version": "2.1.0",          # See §4.3.3.

                               # See §4.3.4.
  "guid": "00001111-2222-3333-4444-555566667777",

                               # See §4.3.5.
  "runGuid": "88889999-AAAA-BBBB-CCCC-DDDDEEEEFFFF",

  "artifacts": {               # See §4.3.6.
    {
      "location": {
        "uri": "apple.png"
      },
      "mimeType": "image/png"
    },
    {
      "location": {
        "uri": "banana.png"
      },
```

  

```
        "mimeType": "image/png"
      }
    },

    "externalizedProperties": {
      "team": "Security Assurance Team"
    }
  }
```

### 4.3.2 $schema property

An `externalProperties` object **MAY** contain a property named `$schema` whose value is a string containing an absolute URI from which a JSON schema document describing the version of the external property file format to which this external property file conforms can be obtained.

If the `$schema` property is present, the JSON schema obtained from the specified URI **SHALL** describe the version of the external property file format corresponding to the SARIF version specified by the `version` property (§4.3.3).

> NOTE 1: The purpose of the `$schema` property is to allow JSON schema validation tools to locate an appropriate schema against which to validate the external property file. This is useful, for example, for tool authors who wish to ensure that external property files produced by their tools conform to the external property file format.

> NOTE 2: The SARIF external property file schema is available at https://raw.githubusercontent.com/oasis-tcs/sarif-spec/master/Schemata/sarif-external-property-file-schema-2.1.0.json.

### 4.3.3 version property

Depending on the circumstances, an `externalProperties` object either **SHALL** or **MAY** contain a property named `version` whose value is a string designating the version of the SARIF specification to which this external property file conforms. If present, this string **SHALL** have the value `"2.1.0"`.

If this `externalProperties` object is the root element of an external property file (see §3.15.2), then `version` **SHALL** be present.

Otherwise (that is, if this `externalProperties` object is an element of `theSarifLog.inlineExternalProperties` (§3.13.5)), then `version` **MAY** be present. If absent, it **SHALL** default to the value of `theSarifLog.version` (§3.13.2).

Although the order in which properties 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 external property file format if new versions are defined in the future.

### 4.3.4 guid property

An `externalProperties` object **SHOULD** contain a property named `guid` whose value is a GUID-valued string (§3.5.3) that equals the `guid` property (§3.16.4) of the corresponding `externalPropertyFileReference` object (§3.16) in the `run.externalPropertyFiles` property (§3.14.2) in the root file.

### 4.3.5 runGuid property

If the externalized properties contained in this `externalProperties` object are associated with a single `run` object (§3.14) `theRun`, and if `theRun` contains an `automationDetails.guid` property (§3.14.3, §3.17.4), the `externalProperties` object **MAY** contain a property named `runGuid` whose value is a GUID-valued string (§3.5.3) that equals `theRun.automationDetails.guid`. Otherwise (that is, if this externalProperties object is associated with more than one run object, or if theRun does not define automationDetails.guid), then runGuid **SHALL** be absent.

### 4.3.6 The property value properties

An externalProperties object **SHALL** contain zero or more externalized properties. The property names in this object, and the names of the corresponding externalized properties, are given in the table in §3.15.3.

The corresponding property values are the values of the externalized properties, exactly as they would have appeared had they occurred inline in the root file.

> NOTE 2: See the EXAMPLE in §4.3.1, where the externalized properties are run.artifacts and run.properties, the externalized value of run.artifacts is stored in a property named artifacts, and the externalized value of run.properties is stored in a property named externalizedProperties.

# ~~4~~5 Conformance

## ~~4.1~~5.1 Conformance targets

This specification defines requirements for the SARIF file format and for certain software components that interact with it. The entities (“conformance targets”) for which this specification defines requirements are:

- **SARIF log file**: A log file in the format defined by the SARIF ~~**resource file**: A SARIF file that contains only those elements related to resources~~specification.
- **SARIF producer**: A program which emits output in the SARIF format.
- **Direct producer**: An analysis tool which acts as a SARIF producer.
- ~~**Deterministic producer**: A SARIF producer which, given identical inputs, repeatedly produces an identical SARIF log file.~~
- **Converter**: A SARIF producer that transforms the output of an analysis tool from its native output format into the SARIF format.
- **SARIF post-processor**: A SARIF producer that transforms an existing SARIF log file into a new SARIF log file, for example, by removing or redacting security-sensitive elements.
- **SARIF consumer**: A program that reads and interprets a SARIF log file.
- **Viewer**: A SARIF consumer that reads a SARIF 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**: a software system that consumes the log files produced by analysis tools, produces reports that enable engineering 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.
- **Engineering system**: a software development environment within which analysis tools execute. It might include a build system, a source control system, a result management system, a bug tracking system, a test execution system, and so on.

The normative content in this specification defines requirements for SARIF log files, except for those normative requirements that are explicitly designated as defining the behavior of another conformance target.

## ~~4.2~~5.2 Conformance Clause 1: SARIF log file

A text file satisfies the “SARIF log file” conformance profile if:

- It conforms to the syntax and semantics defined in §3.

## ~~4.3~~5.3 Conformance Clause 2: SARIF ~~resource file~~producer

~~A text file satisfies the “SARIF resource file” conformance profile if:~~

- ~~Its name conforms to the convention defined in §, “”.~~
- ~~It contains only those elements defined in §.~~
- ~~Those elements that it does contain conform to the syntax and semantics defined in §, except as modified in §.~~

## ~~4.4 Conformance Clause 3: SARIF producer~~

A program satisfies the “SARIF producer” conformance profile if:

- It produces output in the SARIF format, according to the semantics defined in §3.
- It satisfies those normative requirements in §3 that are designated as applying to SARIF producers.

  

## 4.55.4 Conformance Clause 43: Direct producer

An analysis tool satisfies the “Direct producer” conformance profile if:

- It satisfies the “SARIF producer” conformance profile.
- It additionally satisfies those normative requirements in §3 that are designated as applying to “direct producers” or to “analysis tools”.
- It does not emit any objects, properties, or values which, according to §3, are intended to be produced only by converters.

## 4.65.5 Conformance Clause 5: ~~Deterministic producer~~Converter

~~An analysis tool or a converter satisfies the “Deterministic producer” conformance profile if:~~

- ~~It satisfies the “Direct producer” conformance profile or the “Converter” conformance profile, as appropriate.~~
- ~~It satisfies the normative requirements in Appendix F, “Producing deterministic SARIF log files”.~~

## ~~4.7 Conformance Clause 6: Converter~~

A converter satisfies the “Converter” conformance profile if:

- It satisfies the “SARIF producer” conformance profile.
- It additionally satisfies those normative requirements in §3 that are designated as applying to converters.
- It does not emit any objects, properties, or values which, according to §3, are intended to be produced only by direct producers.

## 4.85.6 Conformance Clause 76: SARIF post-processor

A SARIF post-processor satisfies the “SARIF post-processor” conformance profile if:

- It satisfies the “SARIF consumer” conformance profile.
- It satisfies the “SARIF producer” conformance profile.
- It additionally satisfies those normative requirements in §3 that are designated as applying to post-processors.

## 4.95.7 Conformance Clause 87: SARIF consumer

A consumer satisfies the “SARIF consumer” conformance profile if:

- It reads SARIF log files and interprets them according to the semantics defined in §3.
- It satisfies those normative requirements in §3 that are designated as applying to SARIF consumers.

## 4.105.8 Conformance Clause 98: Viewer

A viewer satisfies the “viewer” conformance profile if:

- It satisfies the “SARIF consumer” conformance profile.
- It additionally satisfies the normative requirements in §3 that are designated as applying to viewers.

## 4.115.9 Conformance Clause 109: Result management system

A result management system satisfies the “result management system” conformance profile if:

- It satisfies the “SARIF consumer” conformance profile.
- It additionally satisfies the normative requirements in §3 and Appendix B (“Use of fingerprints by result management systems”) that are designated as applying to result management systems.

  

## ~~4.12~~5.10 Conformance Clause ~~11~~10: Engineering system

An engineering system satisfies the “engineering system” conformance profile if:

- It satisfies the normative requirements in §3 that are designated as applying to engineering systems.

  

# Appendix A. (Informative) Acknowledgments

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

**Participants:**
Andrew Pardoe, Microsoft
Chris Meyer, Microsoft
Chris Wysopal, CA Technologies
David Keaton, Individual
Douglas Smith, Kestrel Technology
Duncan Sparrell, sFractal Consulting LLC
Everett Maus, Microsoft (participated on behalf of Microsoft; now at Google)
Harleen Kaur Kohli, Microsoft
Hendrik Buchwald, RIPS Technologies
Henny Sipma, Kestrel Technology
~~Jim~~James A. Kupsch, SWAMP Project, University of Wisconsin
Jordyn Puryear, Microsoft
Joseph Feiman, CA Technologies
Ken Prole, Code Dx, Inc.
Kevin Greene, Mitre Corporation
Larry Hines, Micro Focus
Laurence J. Golding, Individual
Luke Cartey, Semmle
Mel Llaguno, Synopsys
Michael Fanning, Microsoft
Nikolai Mansourov, Object Management Group
Paul Anderson, GrammaTech, Inc.
Paul Brookes, Microsoft
Paul Patrick, FireEye, Inc.
Philip Royer, Splunk Inc.
Pooya Mehregan, Security Compass
Ram Jeyaraman, Microsoft
Ryley Taketa, Microsoft
Scott Louvau, Microsoft
Sean Barnum, FireEye, Inc.
~~Smith Douglas, Kestrel Technology~~
Stefan Hagen, Individual
Sunny Chatterjee, Microsoft
Tim Hudson, Cryptsoft Pty Ltd.
Trey Darley, New Context Services, Inc.
Vamshi Basupalli, SWAMP Project, University of Wisconsin
Yekaterina O'Neil, Micro Focus

  

# Appendix B. (Normative) 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 so many results, some engineering 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 logically 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 **SHOULD** 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 `result.message`, for example: `"The word xxx should not be used in documentation."`

The SARIF format provides the `partialFingerprints` property to allow analysis tools and other components in the SARIF ecosystem 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 a property in the `partialFingerprints` object to the prohibited word. A result management system **SHALLSHOULD** include the information in `partialFingerprints` in its fingerprint computation. See §3.27.17 for more requirements on how a result management system decides which partial fingerprints to use.

An analysis tool **SHALLSHOULD NOT** include in `partialFingerprints` information that a result management system could deduce from other information in the SARIF file, for example, file hashes. Rather, the result management would use such information, along with `partialFingerprints`, in its computation of `fingerprints`.

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.

A result management system **SHOULD NOT** include an absolute line number (or an absolute byte location in a binary fileartifact) in its fingerprint computation.

A result management system **SHALL NOT** includeNOTE: The inclusion of non-deterministic file format elements (Appendix F, §F.2) in its fingerprint computation.

> A result management system **SHALL NOT** includeor non-deterministic absolute URIs (Appendix F, §F.4) in itsthe fingerprint computation will compromise the usefulness of fingerprints for distinguishing logically identical from logically distinct results.

It is difficult to devise an algorithm that constructs a truly stable fingerprint for a result. Fortunately, for practical purposes, the fingerprint does not 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.

  

# Appendix C. (Informative) 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` (§3.24.7) and `sourceLanguage` (§3.24.10~~property~~) properties of the ~~`file`~~`artifact` object (§3.24) provides this information. In the absence of ~~the `mimeType` property~~these properties, 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~~ c".~~extension.~~

  

# Appendix D. (~~Informative~~Normative) 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~~A converter **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~~a converter **MAY** use it to populate the various property bags (§3.8) and tag lists (§3.8.2) defined by the SARIF format, or they ~~may~~**MAY** simply omit it from the output. When populating a property bag with such information, ~~converters should~~a converter **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~~This makes it easier to match these properties with the source data in the native tool format.

When serializing SARIF as JSON, a converter ~~must~~**SHALL** replace any characters in string-valued properties that cannot occur in a JSON string with the appropriate escape sequence as defined by JSON [RFC8259~~.~~].

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

~~If a~~Since each converter ~~were to~~might 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~~ SARIF 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 (notably the rule id; see §3.27.5~~. For that matter, two~~ ), a SARIF consumer **SHOULD NOT** attempt to combine results produced by 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~~for the same tool.

A converter **SHOULD** populate its own semantic version [SEMVER] property `theRun.conversion.tool.driver.semanticVersion` (§3.19.12~~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~~ ). If it does, and if a subsequent version of the converter synthesizes SARIF elements in a sematically incompatible way, it **SHALL** increment the major version component of its semantic version.

Notwithstanding this general guidance ~~is embodied in various sections of the specification. For example~~recommending that a converter synthesize SARIF elements where possible:

- ~~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~~artifact a result was detected in, but not which ~~file~~artifact the analysis tool was originally instructed to scan, ~~should~~**SHOULD** populate `result.locations`

  

(§3.27.12the location.physicalLocation property,), but should notSHOULD NOT attempt to populate result.analysisTarget (see §(§3.27.13).

- A converter should not attempt to guess whetherSHOULD NOT populate the analysis tool's toolComponent.semanticVersion (§3.19.12tool's) unless it knows that the tool component's version string is intended to be interpreted as a semantic version [SEMVERSemantic Version 2.0.0] version string (see §)..

# Appendix E. (Informative) Locating rule and notification metadata

The SARIF format allows rule and notification metadata to be included in a SARIF log file (see §3.19.23 and §3.19.24). A SARIF log file does not 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 metadataMetadata 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 or notification 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, and if external property files (see §3.15.2) are not used, 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).

  

# Appendix F. (~~Normative~~Informative) Producing deterministic SARIF log files

## F.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~~with identical inputs.

~~Certain build systems provide an~~For example~~ of when~~, this is ~~desirable. Consider~~useful in a build system that caches the ~~results of~~output from each build step. If the build is rerun~~,~~ and the inputs to ~~the~~a given 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~~Consider this sequence of build steps~~ where Steps ~~:

1. A~~, B, and C each run an~~ binary analysis tool ~~on a different set of targets, producing log files~~ analyzes A.dll and produces A.sarif~~, B~~.
2. A bug database ingestion tool reads A.sarif~~,~~ and ~~C~~files bugs for any new results.

If A.sarif~~, and then~~ has not changed between this 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~~and the previous one, the build ~~is re-run, only Steps B and D need~~system does not have to ~~be performed~~execute Step 2.

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

## F.2 Non-deterministic file format elements

~~A tool that produces deterministic output **SHALL NOT** emit the following~~Certain optional elements of the SARIF format~~. All of these elements are **OPTIONAL**~~.

~~Not all of these elements~~ are non-deterministic in ~~all cases.~~most situations. A log file that includes these elements will not be deterministic except under special circumstances. For example~~, some~~:

- If a build ~~systems might run all builds~~system always runs on the same machine ~~or~~ under the same account~~. However, avoiding~~, `invocation.machine` and `invocation.account` is deterministic.
- If a binary analysis tool runs in an environment that guarantees the same memory layout from run to run (for example, an environment that allows a binary to be loaded at a fixed address and that does not use address space layout randomization (ASLR)), then `physicalLocation.address` and `run.addresses` are deterministic.

Authors of analysis tools are encouraged to provide a mechanism (for example, a command line option such as `--known-deterministic-properties:<property name>…`) which allows the tool to emit specified properties even when producing deterministic output.

Avoiding these elements, in conjunction with the techniques described in subsequent sections of this Appendix, ~~guarantees~~makes it more likely that the analysis tool will produce deterministic output~~.~~:

- ~~`invocation.startTime`~~
- Non-deterministic elements in property bag properties.
- Non-deterministic elements in user-facing messages, for example, a timestamp in a result message.

- The trailing component of run.automationDetails.id
- run.automationDetails.guid
- run.baselineGuid
- run.originalUriBaseIds
- run.addresses, because security measures such as address space layout randomization (ASLR) might place the same code at different addresses from run to run.
- invocation.commandLine, because it might specify non-deterministic absolute file paths or other non-deterministic elements.
- invocation.~~endTime~~arguments, for the same reason.
- invocation.processId
- invocation.startTimeUtc
- invocation.endTimeUtc
- invocation.machine
- invocation.account
- invocation.~~fileName (~~workingDirectory, because ~~fileName is specified as being an absolute path, and tools~~the tool might be ~~stored in~~launched from different directories on different machines~~)~~.
- ~~invocation.workingDirectory~~
- invocation.environmentVariables
- ~~The use of absolute file paths in~~ invocation.~~commandLine (~~stdin, stdout, stderr, or stdoutStderr, because ~~builds performed~~the tool's console output might include non-deterministic elements such as timestamps.
- versionControlDetails.revisionId
- versionControlDetails.asOfTimeUtc
- versionControlDetails.mappedTo, because a repository might be downloaded to different directories on different machines~~ might use a different root directory)~~.
- threadFlow.threadId
- threadFlowLocation.executionTimeUtc
- notification.threadId
- notification.~~time~~timeUtc
- result.~~instanceGuid~~guid
- ~~run.instanceGuid~~
- ~~run.automationLogicalId~~
- ~~run.baselineInstanceGuid~~
- ~~run.originalUriBaseIds~~
- 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~~
- physicalLocation.address, for the same reason as run.addresses.

## F.3 Array and dictionary element ordering

~~A tool that produces deterministic output **SHALL** emit~~One obstacle to determinism in SARIF log files is the ordering of array ~~and dictionary~~ elements ~~in a deterministic order~~and object properties.

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

For other arrays, ~~the~~for example properties.tags, 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~~artifacts 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`~~`artifact.hashes` object, a tool might order the property names alphabetically, using a locale-insensitive ordering.

## F.4 Absolute paths

~~The~~ Another obstacle to determinism is the use of ~~non-deterministic absolute file paths (that is,~~ absolute paths which might differ from machine to machine~~) in `fileLocation.uri` properties prevents the production of 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.

~~A tool that produces deterministic output **SHALL NOT** emit non-deterministic~~Tools can 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 ~~`fileLocation`~~`artifactLocation.uri` property with the corresponding ~~`fileLocation`~~`artifactLocation.uriBaseId` property.

## F.5 Inherently non-deterministic tools

The algorithms used by some tools are inherently non-deterministic because, for example, they perform random sampling or random traversals of the graphs that represent the code. Generally, these tools produce mostly the same result set, but there might be small differences between runs.

Such tools can avoid this source of non-determinism by, for example, providing a command-line argument to specify the random number generator seed.

## ~~F.5~~F.6 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 ~~`fileLocation`~~`artifactLocation.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.

## ~~F.6~~F.7 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 it 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.

  

# Appendix G. (Informative) 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 ~~file~~ content. This maximizes the likelihood that an automated tool can safely apply multiple fixes to the same ~~file~~artifact.

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:
   a. Insert a quotation mark before `A3101`.
   b. 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 replacements maximizes the amount of work that can be done by automated fixup tools.

  

# Appendix H. (Informative) Diagnosing results in generated files

Sometimes it is desirable to analyze files generated by the build. These files are usually not under source control, and the build might even overwrite them multiple times. This Appendix offers guidance on how to persist enough information in a SARIF log file to facilitate the diagnosis of results in these files.

In what follows, we will refer to files that are generated only once as “singly generated,” and files that are generated multiple times as “multiply generated~~”.~~.”

It can be difficult to diagnose results in generated files for the following reasons:

- The file might not be available to the engineer who diagnoses the result (for example, the engineer might not have a build environment).
- If the file is multiply generated, then at best only the last version is available, but results might have been found in previous versions.
- It might be difficult to tell which instance of a multiply generated file contained the result.

For both singly and multiply generated files, there are two options (which can be used together):

1. Use the `physicalLocation` object’s (§3.29) `region` (§3.29.4) and `contextRegion` (§3.29.5) properties to store enough of the generated file’s contents to facilitate diagnosis. The `region` object’s (§3.30) `snippet` property (§3.30.13) holds the relevant portion of the file contents.
2. Use the ~~`file`~~`artifact` object’s (§3.24) `contents` (§3.24.8) property to persist the entire contents of the file in `theRun.artifacts` (§3.14.15~~`run.files`.~~).

The first option is more compact; the second allows a SARIF viewer to present results with greater context.

EXAMPLE 1: In this example, the analysis tool populates `region.snippet` and `contextRegion.snippet`, allowing a SARIF viewer to display just enough context (one hopes) to diagnose the result.

```
{                                            # A run object (§3.14).
  "originalUriBaseIds": {                    # See §3.14.14
    "GENERATED": {
      "uri": "file:///C:/code/browser/obj"/"
    }
  },

  "results": [                               # See §3.14.23.
    {                                        # A result object (§3.27).
      "ruleId": "CS6789",                    # See §3.27.5.
      "message": {                           # See §3.27.11.
        "text": "Division by 0!""
      },
      "locations": [                         # See §3.27.12.
        {                                    # A location object (§3.28).
          "physicalLocation": {              # See §3.28.3.
            "fileLocationartifactLocation": {
              "uri": "ui/window.g.cs",       # A generated file (".g").
              "uriBaseId": "GENERATED"
            },
            "region": {
              "startLine": 42,
              "snippet": {
                "text": "    int z = x / y;\r\n"
              }
            },
            "contextRegion": {
              "startLine": 40,
              "endLine": 42,
```

```
                "snippet": {
                  "text":
                   "    int x = 54;\r\n    int y = 0;\r\n    int z = x / y;\r\n"
                }
              }
            }
          }
        ]
      }
  ],

  ...
}
```

EXAMPLE 2: In this example, the analysis tool populates ~~file~~artifact.contents, allowing a SARIF viewer to present the result in a larger context at the expense of a larger log file.

```
{
  "originalUriBaseIds": {
    "GENERATED": "file:///dev-1.example.com/code/browser/obj"{
      "uri": "file:///dev-1.example.com/code/browser/obj/"
    }
  },

  "results": [
    {
      "ruleId": "CS6789",
      "message": {
        "text": "Division by 0!""
      },
      "locations": [
        {
          "physicalLocation": {
            "fileLocationartifactLocation": {
              "uri": "ui/window.g.cs",
              "uriBaseId": "GENERATED"",
              "index": 0
            },
            "region": {
              "startLine": 42
            },
            "contextRegion": {
              "startLine": 40,
              "endLine": 42
            }
          }
        }
      ]
    }
  ],

  "files": {    artifacts": [                          # See §3.14.15.
    "ui/window.g.cs":                      # Property name matches uri property above.
      {                                        # A file      {
# An artifact object (§3.24).
      "contents":"location": {                       # See §3.24.2.
        "uri": "ui/window.g.cs",
        "uriBaseId": "GENERATED"
      },
      "contents": {                                # See §3.24.8.
        {                                # A fileContent object (§).
          "text": "..."                          # See §3.3.2.
```

```
      }
    }
  }]
}
}
```

Multiply generated files are treated similarly, but they present an additional problem: if more than one version of a given multiply generated file appears in `run.files``theRun.artifacts` – either because the analysis tool wishes to persist the file contents, or for any other reason – then there must be a way to give each instance a different property namedistinguish them.

In EXAMPLE 2 above, if `"ui/window.g.cs"` is multiply generated, there can't be two properties in `run.files` with that property name. Prepending the property name with the URI base id (for example, `"#GENERATED#ui/window.g.cs"`), as described in §, doesn't help, because each version of the generated file has the same URI base id.

The recommended solution is for the analysis tool to create a new URI base identry in `theRun.artifacts` for each version of the generated files. For example, the tool might append an incremented integer to the URI base id for each version of the file. The result might look like the following example.

EXAMPLE 3: In this example, `"ui/window.g.cs"` is multiply generated. The analysis tool creates URI base ids `"GENERATED-1"` and `"GENERATED-2"`distinct entries in `theRun.artifacts` to distinguish the two versions.

```
{
  "originalUriBaseIds": {
    "GENERATED-1": {
      "uri": "file:///dev-1.example.com/code/browser/obj",/"
    "GENERATED-2": "file:///dev-1.example.com/code/browser/obj"
    }
  },

  "results": [
    {
      "ruleId": "CS6789",
      "message": {
        "text": "Division by 0!""
      },
      "locations": [
        {
          "physicalLocation": {
            "fileLocationartifactLocation": {
              "uri": "ui/window.g.cs",
              "uriBaseId": "GENERATED-1"",
            },  "index": 0                    # Points to the appropriate
instance
            },                              #  of the generated file.
            "region": {
              "startLine": 42
            },
            "contextRegion": {
              "startLine": 40,
              "endLine": 42
            }
          }
        }
      ]
    }
  ],
```

  

```
  "files": {artifacts": [
    "#GENERATED-1#{
      "location": {
        "uri": "ui/window.g.cs": {          # Unique property name.",
      ...
    },

    "#  "uriBaseId": "GENERATED-2#",
      },
      "lastModifiedTimeUtc": "2019-04-13T11:45:23.477",
      "contents": {
        "text": "..."
      }
    },

    {
      "location": {
        "uri": ui/window.g.cs",
        "uriBaseId": "GENERATED",
      },
      "lastModifiedTimeUtc": "2019-04-13T11:46:27.013",
      "contents": {
        "text": "..."
      }
    }
  ]
}
```

  

# Appendix I. (Informative) Detecting incomplete result sets

This specification describes three conditions that inform the SARIF consumer that the tool has failed to produce a comprehensive set of results. For convenience, this Appendix gathers those conditions together in one place:

- If any `invocation` object (§3.20) in `theRun.invocations` (§3.14.11) has a value of `false` for its `executionSuccessful` property (§3.20.14), the tool either failed to start, terminated with an exit code that denotes failure, or terminated with an unhandled exception or signal.

- If any `notification` object (§3.58) in `invocation.toolExecutionNotifications` (§3.20.21) or `toolConfigurationNotifications` (§3.20.22": {        # Unique) has a value of `"error"` for its `level` property (§3.58.6name), it is possible that the tool was unable to execute every analysis rule on every analysis target. Therefore, the results cannot be assumed to be complete.

- If `theRun.results` (§3.14.23) is `null`, the tool either failed to start or failed to begin its analysis.

These conditions apply separately to each run in the log file.

  

# Appendix J. (Informative) Sample sourceLanguage values

```
This Appendix contains a list of sample values for the artifact.sourceLanguage
property (§3.24.10 ...
    }
  }
}
```

) for some common programming languages. The purpose of this Appendix is to promote interoperability by encouraging SARIF producers to use the same identifiers for these languages.

The names of some of the languages in this list are the trademarks of their respective owners.

- abap
- actionscript
- ada
- apex
- c
- clojure
- cobol
- coldfusion
- cplusplus
- csharp
- css
- d
- erlang
- fsharp
- fortran
- go
- groovy
- haskell
- java
- javascript
- json
- jsp
- julia
- lisp
- lua
- markdown (variants: markdown/gfm, markdown/cmark)
- objectivec
- objectpascal
- ocaml
- perl
- php
- prolog
- python
- r
- razor
- ruby
- rust
- scala
- scheme
- sql (variants: sql/tsql, sql/psql).
- swift
- typescript
- visualbasic
- visualbasicdotnet
- yaml
- Markup languages:
    - html
    - sgml
    - xml
- Typesetting languages:
    - latex
    - nroff
    - roff
    - tex
    - troff
- UNIX® shell languages:
    - bash
    - csh
    - ksh
    - sh
    - tcsh
- Windows® shell languages:
    - cmd
    - powershell

  

# ~~Appendix I.~~Appendix K. (Informative) Examples

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

## ~~I.1~~K.1 Minimal valid SARIF log file

This is a minimal valid SARIF log file. It contains only those elements required by the specification (elements which the specification states **SHALL** be present).

The file contains a single `run` object (§3.14) with an empty `results` array (§3.14.23), as would happen if the tool detected no issues in any of the ~~files~~artifacts it scanned.

```
{
  "version": "2.1.0.0",
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "CodeScanner"
        }
      },
      "results": [
      ]
    }
  ]
}
```

## ~~I.2~~K.2 Minimal recommended SARIF log file with source information

This is a minimal recommended SARIF log file for the case where an analysis tool produced results and source location information is available.

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

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

The file contains a single `run` object (§3.14) with a `results` array (§3.14.23). The results array contains a single `result` object (§3.27) so the recommended elements of the `result` object can be shown.

Its `run.~~files~~artifacts` property (§3.14.15) specifies only those ~~files~~artifacts in which the tool detected a result.

It does not contain a `run.logicalLocations` property (§3.14.17), because when physical location information is available, that property is optional (it "**MAY**" be present).

This example also includes a `~~run~~toolComponent.rules` property (§3.19.23) 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": "2.1.0.0",
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "CodeScanner"",
      },     "rules": [
          "files": {
            "id": "C2001",
            "fullDescription": {
              "text": "A variable was used without being initialized. This can
result"file:///build.example.com/work/
                          in runtime errors such as null reference exceptions."
```

  

```
              },
              "messageStrings": {
                "default": {
                  "text": "Variable \"{0}\" was used without being initialized."
                }
              }
            }
          ]
        }
      },
      "artifacts": [
        {
          "location": {
            "uri": "src/collections/list.cpp": {",
          "mimeType": "text/x-  "uriBaseId": "SRCROOT"
          },
          "sourceLanguage": "c"
        }
      },],
      "results": [
        {
          "ruleId": "C2001",
          "ruleIndex": 0,
          "message": {
            "text": "Variable \"id": "default",
            "arguments": [
              "count\" was used without being initialized.","
            "richText": "Variable `count` was used without being initialized."]
          },
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "file://build.example.com/work/src/collections/list.cpp",
                  "uriBaseId": "SRCROOT",
                  "index": 0
                },
                "region": {
                  "startLine": 15
                }
              },
              "fullyQualifiedLogicalNamelogicalLocations": [
                {
                  "fullyQualifiedName": "collections::list:::add"
                }
              ]
            }
      },    ]
      "rules": {  }
        "C2001": {]
          "id": "C2001",
          "fullDescription": {
            "text": "A variable was used without being initialized. This can result in
                     runtime errors such as null reference exceptions."
          }
        }
      }
    }
  ]
}
```

## I.3K.3 Minimal recommended SARIF log file without source information

This is a minimal recommended SARIF file for the case where an analysis tool produced results and source location information is not available.

1. The analysis tool was run with the intent of scanning files and producing results (see §), but
2. The analysis tool does not have source location information available.

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

The file contains a single `run` object (§3.14) with a `results` array (§3.14.23). The results array contains a single `result` object (§3.27) so the recommended elements of the `result` object can be shown.

Its `run.filesartifacts` property (§3.14.15) specifies only those filesartifacts in which the tool detected a result.

It contains a `run.logicalLocations` property (§3.14.17), because when physical location information is not available, that property is recommended.


```
{
  "version": "2.1.0.0",
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "BinaryScanner"
      },,  }
      "files": {},
       "file://build.example.com/work/"artifact": [
        {
          "location": {
            "uri": "bin/example": {",
          "mimeType": "application/vnd.microsoft.portable-executable  "uriBaseId":
"BINROOT"
          }
      },  }
      ],
      "logicalLocations": {[
        "Example": {
          "name": "Example",
          "kind": "namespace"
        },
        "Example.Worker": {
          "name": "Worker",
          "kind": "type",
          "parentKeyfullyQualifiedName": "Example"
        },
        "Example.Worker.DoWork": {",
          "kind": "type",
          "parentIndex": 0
        },
        {
          "name": "DoWork",
          "fullyQualifiedName": "Example.Worker.DoWork",
          "kind": "function",
          "parentKey": "Example.Worker"parentIndex": 1
        }
      },],
      "results": [
        {
          "ruleId": "B6412",
          "message": {
            "text": "The insecure method \"Crypto.Sha1.Encrypt\" should not be used.",."
            "richText": "The insecure method `Crypto.Sha1.Encrypt` should not be used."
          },
          "level": "warning",
          "locations": [
            {
              "fullyQualifiedLogicalNamelogicalLocations": [
                {
                  "fullyQualifiedName": "Example.Worker.DoWork"",
            }
          ]
        }
      ]
    }
  ]
}
```

## ~~I.4 SARIF resource file with rule metadata~~

~~This sample demonstrates the use of SARIF for exporting a tool's rule metadata. The file conforms to the SARIF resource file format (§) and contains rule metadata for the language specified by `tool.language` (§).~~

```
{
  "version": "      "index": 2.0.0",
  "runs": [              }
    {           ]
      "tool": {      }
        "name": "BinaryAnalyzer",  ]
        "language": "en-US"}
      },]
      "resources": {
        "rules": {
          "BA2006": {
            "id": "BA2006",
            "name": {
              "text": "BuildWithSecureTools"
            },
            "shortDescription": {
              "text": "Application code should be compiled with
                       the most up-to-date tool sets."
            },
            "fullDescription": {
              "text": "Application code should be compiled with
                       the most up-to-date tool sets. The latest
                       version is 2.2."
            },
            "messageStrings": {
              "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"
          }
        }
      }
    }
  ]
}
```

## ~~I.5~~K.4 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~~possible, 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`~~`suppressed` (to demonstrate the `result.`~~`suppressionStates`~~`suppressions` property), which is unrealistic.

```
{
  "version": "2.1.0.0",
  "$schema": "http://json.schemastore.orghttps://raw.githubusercontent.com/oasis-tcs/sarif-2.0.0spec/master/Schemata/sarif-schema-2.1.0.json",
  "runs": [
    {
      "instanceGuidautomationId": {
        "guid": "BC650830-A9FE-44CB-8818-AD6C387279A0",
      "logicalId  "id": "Nightly code scan",/2018-10-08"
```

```
      "baselineInstanceGuid},
      "baselineGuid": "0A106451-C9B1-4309-A7EE-06988B95F723",
      "automationLogicalIdrunAggregates": [
        {
          "id": "Build-/14.0.1.2-/Release-/20160716-13:22:18",
      "architecture": "x86",
          "correlationGuid": "26F138B6-6014-4D3D-B174-6E1ACE9439F3"
        }
      ],
      "tool": {
        "driver": {
          "name": "CodeScanner",
          "fullName": "CodeScanner 1.1 for UnixMicrosoft Windows (R) (en-US)",
          "version": "2.1",
          "semanticVersion": "2.1.0",
        "fileVersion  "dottedQuadFileVersion": "2.1.0.0",
        "language": "en-US  "releaseDateUtc": "2019-03-17",
        "sarifLoggerVersion": "1.25.0  "organization": "Example Corporation",
          "product": "Code Scanner",
          "productSuite": "Code Quality Tools",
          "shortDescription": {
            "text": "A scanner for code."
          },
          "fullDescription": {
            "text": "A really great scanner for all your code."
          },
          "properties": {
            "copyright": "Copyright (c) 2017 by Example Corporation.."
                         All rights reserved."},
        }  "globalMessageStrings": {
      }, "variableDeclared": {
              "text": "Variable \"{0}\" was declared here.",
              "markdown": " Variable `{0}` was declared here."
            }
          },
          "rules": [
            {
              "id": "C2001",
              "deprecatedIds": [
                "CA2000"
              ],
              "defaultConfiguration": {
                "level": "error",
                "rank": 95
              },
              "shortDescription": {
                "text": "A variable was used without being initialized."
              },
              "fullDescription": {
                "text": "A variable was used without being initialized. This can result
                         in runtime errors such as null reference exceptions."
              },
              "messageStrings": {
                "default": {
                  "text": "Variable \"{0}\" was used without being initialized.
                           It was declared [here]({1}).",
                  "markdown": "Variable `{0}` was used without being initialized.
                           It was declared [here]({1})."
                }
              }
            }
          ],
          "notifications": [
            {
              "id": "start",
              "shortDescription": {
                "text": "The run started."
              },
              "messageStrings": {
                "default": {
                  "text": "Run started."
                }
```

  

```
              }
            },
            {
              "id": "end",
              "shortDescription": {
                "text": "The run ended".
              },
              "messageStrings": {
                "default": {
                  "text": "Run ended."
                }
              }
            }
          ],
          "language": "en-US"
        },
        "extensions": [
          {
            "name": "CodeScanner Security Rules",
            "version": "3.1",
            "rules": [
              {
                "id": "S0001",
                "defaultConfiguration": {
                  "level": "error"
                },
                "shortDescription": {
                  "text": "Do not use weak cryptographic algorithms."
                },
                "messageStrings": {
                  "default": {
                    "text": "The cryptographic algorithm '{0}' should not be used."
                  }
                }
              }
            ]
          }
        ]
      },
      "language": "en-US",
      "versionControlProvenance": [
        {
          "repositoryUri": "https://github.com/example-corp/browser",
          "revisionId": "5da53fbb2a0aaa12d648b73984acc9aac2e11c2a",
          "mappedTo": {
            "uriBaseId": "PROJECTROOT"
          }
        }
      ],
      "originalUriBaseIds": {
        "PROJECTROOT": {
          "uri": "file://build.example.com/work/"
        },
        "SRCROOT": "file://build.example.com/work/{
          "uri": " src/",
          "uriBaseId": "PROJECTROOT"
        },
        "BINROOT": "file://build.example.com/work/{
          "uri": " bin/"/",
          "uriBaseId": "PROJECTROOT"
        }
      },
      "invocations": [
        {
          "commandLine": "CodeScanner @build/collections.rsp",
          "responseFiles": [
            {
              "uri": "build/collections.rsp",
              "uriBaseId": "SRCROOT"",
            }  "index": 0
          ],  }
          "startTime],
```

  

```
          "startTimeUtc": "2016-07-16T14:18:25Z",
          "endTimeendTimeUtc": "2016-07-16T14:19:01Z",
          "machine": "BLD01",
          "account": "buildAgent",
          "processId": 1218,
          "fileName": "/bin/tools/CodeScanner",
          "workingDirectory": "/home/buildAgent/src",{
            "uri": "file:///home/buildAgent/src"
          },
          "environmentVariables": {
            "PATH": "/usr/local/bin:/bin:/bin/tools:/home/buildAgent/bin",
            "HOME": "/home/buildAgent",
            "TZ": "EST"
          },
          "configurationNotificationstoolConfigurationNotifications": [
            {
              "descriptor": {
                "id": "UnknownRule","
              },
              "associatedRule": {
                "ruleId": "ABC0001","
              },
              "level": "warning",
              "message": {
                "text": "Could not disable rule \"ABC0001\" because
                         there is no rule with that id."
              }
            }
          ],
          "toolNotificationstoolExecutionNotifications": [
            {
              "descriptor": {
                "id": "CTN0001","
              },
              "level": "note",
              "message": {
                "text": "Run started."
              }
            },
            {
              "descriptor": {
                "id": "CTN9999","
              "ruleId": "C2152",},
              "associatedRule": {
                "id": "C2001",
                "index": 0,
              },
              "level": "error",
              "message": {
                "text": "Exception evaluating rule \"C2152C2001\". Rule disabled;
                         run continues."
              },
              "locations": [
                {
                  "physicalLocation": {
                "fileLocation    "artifactLocation": {
                      "uri": "crypto/hash.cpp",
                      "uriBaseId": "SRCROOT"",
                }       "index": 4
              },      }
                  }
                }
              ],
              "threadId": 52,
              "timetimeUtc": "2016-07-16T14:18:43.119Z",
              "exception": {
                "kind": "ExecutionEngine.RuleFailureException",
                "message": {
                  "text": "Unhandled exception during rule evaluation."
                },
                "stack": {
                  "frames": [
```

  

```
                    {
                      "messagelocation": {
                        "message": {
                          "text": "Exception thrown"
                        },
                        "logicalLocations": [
                          {
                            "fullyQualifiedName":
                              "Rules.SecureHashAlgorithmRule.Evaluate"
                          }
                        ],
                        "physicalLocation": {
                          "address": {
                            "offset": 4244988
                          }
                        }
                      },
                      "module": "RuleLibrary",
                      "threadId": 52,
                      "fullyQualifiedLogicalName":},
                        "Rules.SecureHashAlgorithmRule.Evaluate",{
                      "location": {
                        "logicalLocations": [
                          {
                            "fullyQualifiedName":
                              "ExecutionEngine.Engine.EvaluateRule"
                          }
                        ],
                        "physicalLocation": {
                          "address": 10092852{
                    },,         "offset": 4245514
                    {         }
                        }
                      },
                      "module": "ExecutionEngine",
                      "threadId": 52,
                      "fullyQualifiedLogicalName":
                          "ExecutionEngine.Engine.EvaluateRule",
                      "address": 10073356
                    }
                  ]
                },
                "innerExceptions": [
                  {
                    "kind": "System.ArgumentException",
                    "message": "length is < 0"
                  }
                ]
              }
            },
            {
              "id": "CTN0002",descriptor": {
                "id": "CTN0002"
              },
              "level": "note",
              "message": {
                "text": "Run ended."
              }
            }
          }],
        }  "exitCode": 0,
      },     "executionSuccessful": true
      "files": {  }
      ],
      "artifacts": [
        {
          "location": {
            "uri": "build/collections.rsp": {",
            "uriBaseId": "SRCROOT"
          },
          "mimeType": "text/plain",
          "length": 81,
```

```
          "contents": {
            "text": "-input src/collections/*.cpp -log out/collections.sarif -rules all
-disable C9999"
          }
        }},
        {
          "location": {
            "uri": "application/main.cpp",
            "uriBaseId": "SRCROOT"
          },
          "sourceLanguage": "cplusplus",
          "length": 1742,
          "hashes": {
            "sha-256": "cc8e6a99f3eff00adc649fee132ba80fe333ea5a"
          }
        },
        {
          "location": {
            "uri": "collections/list.cpp": {",
          "mimeType": "text/x-c  "uriBaseId": "SRCROOT"
          },
          "sourceLanguage": "cplusplus",
          "length": 980,
          "hashes": [{
           {
              "algorithm": "sha-256",
              "value": "b13ce2678a8807ba0765ab94a0ecd394f869bc81"
          }
        ]},
        },{
          "location": {
            "uri": "collections/list.h",
            "uriBaseId": "SRCROOT"
          },
          "sourceLanguage": "cplusplus",
          "length": 24656,
          "hashes": {
            "sha-256": "849be119aaba4e9f88921a99e3036fb6c2a8144a"
          }
        },
        {
          "location": {
            "uri": "crypto/hash.cpp",
            "uriBaseId": "SRCROOT"
          },
          "sourceLanguage": "cplusplus",
          "length": 1424,
          "hashes": {
            "sha-256": "3ffe2b77dz255cdf95f97d986d7a6ad8f287eaed"
          }
        },
        {
          "location": {
            "uri": "app.zip": {",
            "uriBaseId": "BINROOT"
          },
          "mimeType": "application/zip"",
        },  "length": 310450,
        "app.zip#/  "hashes": {
            "sha-256": "df18a5e74b6b46ddaa23ad7271ee2b7c5731cbe1"
          }
        },
        {
          "location": {
            "uri": "/docs/intro.docx": {"
           "uri": "/docs/intro.docx",},
          "mimeType":
             "application/vnd.openxmlformats-officedocument.wordprocessingml.document",
           "parentKey": "app.zip","parentIndex": 5,
          "offset": 17522,
          "length": 4050
        }
```

  

```
      },],
      "logicalLocations": {[
        "collections::list::add": {
          "name": "add",
          "fullyQualifiedName": "collections::list::add",
          "decoratedName": "?add@list@collections@@QAEXH@Z",
          "kind": "function",
          "parentKey": "collections::list"parentIndex": 1
        },
        "collections::list": {
          "name": "list",
          "fullyQualifiedName": "collections::list",
          "kind": "type",
          "parentKey": "collections"parentIndex": 2
        },
        "collections": {
          "name": "collections",
          "kind": "namespace"
        }},
      },  {
          "name": "add_core",
          "fullyQualfiedName": "collections::list::add_core",
          "decoratedName": "?add_core@list@collections@@QAEXH@Z",
          "kind": "function",
          "parentIndex": 1
        },
        {
          "fullyQualifiedName": "main",
          "kind": "function"
        }
      ],
      "results": [
        {
          "ruleId": "C2001",
          "ruleMessageId": "defaultruleIndex": 0,
          "kind": "fail",
          "level": "error",
          "message": {
            "id": "default",
            "arguments": [
              "ptr"",
            }  "0"
          },  ]
          "suppressionStates": [ "suppressedExternally"}],
          "suppressions": [
            {
              "kind": "external",
              "state": "accepted"
            }
          ],
          "baselineState": "existing",
          "level": "error",rank": 95,
          "analysisTarget": {
            "uri": "collections/list.cpp",
            "uriBaseId": "SRCROOT",
            "index": 2
          },
"
          },
          "locations": [
            {
              "physicalLocation": {
                "fileLocationartifactLocation": {
                  "uri": "collections/list.h",
                  "uriBaseId": "SRCROOT"",
                  "index": 3
                },
                "region": {
                  "startLine": 15,
                  "startColumn": 9,
                  "endLine": 15,
                  "endColumn": 10,
```

```
                  "charLength": 1,
                  "charOffset": 254,
                  "snippet": {
                    "text": "add_core(ptr, offset, val);\n    return;"
                  }
                }
              },
              "fullyQualifiedLogicalNamelogicalLocations": [
                {
                  "fullyQualifiedName": "collections::list:::add"",
                  "index": 0
                }
              ]
            }
          ],
          "relatedLocations": [
            {
              "id": 0,
              "message": {
                "text": "Variable \"id": "variableDeclared",
                "arguments": [
                  "ptr\" was declared here.",""
                "richText": "Variable `ptr` was declared here."]
              },
              "physicalLocation": {
                "fileLocationartifactLocation": {
                  "uri": "collections/list.h",
                  "uriBaseId": "SRCROOT"",
                  "index": 3
                },
                "region": {
                  "startLine": 8,
                  "startColumn": 5
                }
              },
              "fullyQualifiedLogicalNamelogicalLocations": [
                {
                  "fullyQualifiedName": "collections::list:::add"",
                  "index": 0
                }
              ]
            }
          ],
          "codeFlows": [
            {
              "message": {
                "text": "Path from declaration to usage"
              },

              "threadFlows": [
                {
                  "id": "thread-52",
                  "locations": [
                    {
                      "step": 1,
                      "importance": "essential",
                      "location": {
                        "message": {
                          "text": "Variable \"ptr\" declared.",
                        "richText  "markdown": "Variable `ptr` declared."
                        },
                      "location": {
                        "physicalLocation": {
                          "fileLocationartifactLocation": {
                            "uri":"collections/list.h",
                            "uriBaseId": "SRCROOT"",
                            "index": 3
                          },
                          "region": {
                            "startLine": 15,
                            "snippet": {
                              "text": "int *ptr;"
```

```
                        },}
                      }
                    },
                    "fullyQualifiedLogicalNamelogicalLocations": [
                      {
                        "fullyQualifiedName": "collections::list:::add"",
                        "index": 0
                      }
                    ]
                  },
                  "module": "platform"
                },
                {
                  "step": 2,state": {
                  "state": {
                    "y": {
                      "text": "2",""
                    },
                    "z": {
                      "text": "4",""
                    },
                    "y + z": {
                      "text": "6",""
                    },
                    "q": {
                      "text": "7"
                    }
                  },
                  "importance": "unimportant",
                  "location": {
                    "physicalLocation": {
                      "fileLocationartifactLocation": {
                        "uri":"collections/list.h",
                        "uriBaseId": "SRCROOT"",
                        "index": 3
                      },
                      "region": {
                        "startLine": 15,
                        "snippet": {
                         "text": "offset = (y + z) * q + 1;"
                        }
                      }
                    },
                    "logicalLocations": [
                      {
                        "fullyQualifiedName": "collections::list::add",
                        "index": 0
                      }
                    ],
                    "annotations": [
                      {
                        "startLine": 15,
                        "startColumn": 13,
                        "endColumn": 19,
                        "message": {
                          "text": "(y + z) = 42",
                          "richTextmarkdown": "`(y + z) = 42`"
                        }
                      }
                    ],
                    "fullyQualifiedLogicalName": "collections::list:add"
                  },
                  "module": "platform"
                },
                {
                  "step": 3,
                  "importance": "essential",
                  "location": {
                    "message": {
                      "text": "Uninitialized variable \"ptr\" passed to
                               method \"add_core\".",
                    "richText  "markdown": "Uninitialized variable `ptr` passed to
```

  

```
                                        method `add_core`."
                              },
                            "location": {
                              "physicalLocation": {
                                "fileLocationartifactLocation": {
                                  "uri":"collections/list.h",
                                  "uriBaseId": "SROOT"",
                                  "index": 3
                                },
                                "region": {
                                  "startLine": 25,
                                  "snippet": {
                                    "text": "add_core(ptr, offset, val)"
                                  }
                                }
                              },
                              "fullyQualifiedLogicalNamelogicalLocations": [
                                {
                                  "fullyQualifiedName": "collections::list:::add"",
                                  "index": 0
                                }
                              ]
                            },
                            "module": "platform"
                          }
                        ]
                      }
                    ]
                  }
                ],
                "stacks": [
                  {
                    "message": {
                      "text": "Call stack resulting from usage of uninitialized variable."
                    },
                    "frames": [
                      {
                        "messagelocation": {
                          "message": {
                            "text": "Exception thrown."
                          },
                        "location": {
                          "physicalLocation": {
                            "fileLocationartifactLocation": {
                              "uri": "collections/list.h",
                              "uriBaseId": "SRCROOT"",
                              "index": 3
                            },
                            "region": {
                              "startLine": 110,
                              "startColumn": 15
                            }},
                          },  "address": {
                          "fullyQualifiedLogicalName    "offset": 4229178
                            }
                          },
                          "logicalLocations": [
                            {
                              "fullyQualifiedName": "collections::list:::add_core"",
                              "index": 0
                            }
                          ],
                        },
                        "module": "platform",
                        "threadId": 52,
                        "address": 10092852,
                        "offset": 16,
                        "parameters": [ "null", "0", "14" ]
                      },
                      {
                        "location": {
                          "physicalLocation": {
```

  

```
                    "fileLocationartifactLocation": {
                      "uri": "collections/list.h",
                      "uriBaseId": "SRCROOT"",
                      "index": 3
                    },
                    "region": {
                      "startLine": 43,
                      "startColumn": 15
                    }},
                  },  "address": {
                  "fullyQualifiedLogicalName    "offset": 4229268
                    }
                  },
                  "logicalLocations": [
                    {
                      "fullyQualifiedName": "collections::list:::add"",
                      "index": 0
                    }
                  ]
                },
                "module": "platform",
                "threadId": 52,
                "address": 10092176,
                "offset": 84,
                "parameters": [ "14" ]
              },
              {
                "location": {
                  "physicalLocation": {
                    "fileLocationartifactLocation": {
                      "uri": "application/main.cpp",
                      "uriBaseId": "SRCROOT"",
                      "index": 1
                    },
                    "region": {
                      "startLine": 28,
                      "startColumn": 9
                    }},
                  },  "address": {
                  "fullyQualifiedLogicalName    "offset": 4229836
                    }
                  },
                  "logicalLocations": [
                    {
                      "fullyQualifiedName": "main"",
                      "index": 4
                    }
                  ]
                },
                "module": "application",
                "threadId": 52,
              "address": 10091200,}
            ]
          }
        ],
        "addresses": [
          {
            "baseAddress": 4194304,
            "fullyQualifiedName": "collections.dll",
            "kind": "module",
            "section": ".text"
          },
          {
            "offset": 156100,
            }"fullyQualifiedName": "collections.dll!collections::list::add",
            }"kind": "function",
          }  "parentIndex": 0
          {
            "offset": 22,
            "fullyQualifiedName": "collections.dll!collections::list::add+0x16",
            "parentIndex": 1
          }
```

  

```
          ],
          "fixes": [
            {
              "description": {
                "text": "Initialize the variable to null"
              },
              "fileChangesartifactChanges": [
                {
                  "fileLocationartifactLocation": {
                    "uri": "collections/list.h",
                    "uriBaseId": "SRCROOT"",
                    "index": 3
                  },
                  "replacements": [
                    {
                      "deletedRegion": {
                        "startLine": 42
                      },
                      "insertedContent": {
                        "text": "A different line\n"
                      }
                    }
                  ]
                }
              ]
            }
          ],
          "hostedViewerUri":
            "https://www.example.com/viewer/3918d370-c636-40d8-bf23-8c176043a2df",
          "workItemUris": [
            "https://github.com/example/project/issues/42",
            "https://github.com/example/project/issues/54"
          }],
        }  "provenance": {
      ],
      "resources": {
        "rules": {
          "C2001": {
            "id": "C2001",
            "shortDescription": {
              "text": "A variable was used without being initialized."
            },
            "fullDescription": {
              "text": "A variable was used without being initialized. This can result
                       in runtime errors such as null reference exceptions."
            },
            "messageStrings": {
              "default": "Variable \"{0}\" was used without being initialized."
            },
            "richMessageStrings": {
              "default": "Variable `{0}` was used without being initialized."
            }
          }
        }
      }
            "firstDetectionTimeUtc": "2016-07-15T14:20:42Z",
            "firstDetectionRunGuid": "8F62D8A0-C14F-4516-9959-1A663BA6FB99",
            "lastDetectionTimeUtc": "2016-07-16T14:20:42Z",
            "lastDetectionRunGuid": "BC650830-A9FE-44CB-8818-AD6C387279A0",
            "invocationIndex": 0
          }
        }
      ]
    }
  ]
}
```

  

# ~~Appendix J.~~Appendix L. (Informative) Revision History

| Revision | Date | Editor | Changes Made |
|---|---|---|---|
| 01 | 2017/09/22 | Laurence J. Golding | Initial version, transcribed from contribution with minor corrections. |
| 02 | 2017/11/29 | Laurence J. Golding | Incorporated changes for GitHub issues #25, #27, and #56. |
| 03 | 2018/01/10 | Laurence J. Golding | Incorporated changes for GitHub issues #33, #61, #69, and #72. Made several minor editorial changes and a few changes to correct inaccuracies. |
| 04 | 2018/01/11 | Laurence J. Golding | Incorporated changes for GitHub issue #73. |
| 05 | 2018/01/15 | Laurence J. Golding | Incorporated changes for GitHub issue #79. |
| 06 | 2018/01/16 | Laurence J. Golding | Two minor editorial changes. |
| 07 | 2018/01/17 | Laurence J. Golding | Incorporated changes for GitHub issue #65. |
| 08 | 2018/02/19 | Laurence J. Golding | Incorporated changes for GitHub issues #66, #74, #81, #88. |
| 09 | 2018/02/28 | Laurence J. Golding | Incorporate changes for GitHub issues #82, #83, #89, #90, #91, #92, #94, and #104. |
| 10 | 2018/03/16 | Laurence J. Golding | Incorporate changes for GitHub issues #10, #15, #23, #29, #63, #64, #84, #102, #110. |
| 11 | 2018/03/28 | Laurence J Golding | Incorporate changes for GitHub issues #75, #80, #86, #95, #96, and #133. |
| 12 | 2018/04/18 | Laurence J. Golding | Incorporate changes for GitHub issues #46, #98, #99, #107, #108, #113, #119, #120, #125, and #130. |
| 13 | 2018/05/03 | Laurence J. Golding | Incorporate changes for GitHub issues #122, #126, #134, #136, #137, #139, #145, #147, #154, and #155.<br>Editorial change in `result.ruleMessageId`. |
| 14 | 2018/05/08 | Laurence J. Golding | Address GitHub issue #156: editorial |
| 15 | 2018/05/17 | Laurence J. Golding | Incorporate changes for GitHub issues #103, #138, #141, #143, #153, #157, #159, #160, #161, #162, #163, #165, #166, #167, and #170.<br>Editorial change for “occurs” *vs.* “contains”. |
| 16 | 2018/05/30 | Laurence J. Golding | Incorporate changes for GitHub issues #93, #149, #160 (revised), #171, #176, #181, and #187 (editorial).<br>Editorial change: Remove “semanticVersion” from all but “Comprehensive” example in Appendix I. |

  

| | | | |
|---|---|---|---|
| | | | Editorial change: Improve language for default values. |
| 17 | 2018/06/06 | Laurence J. Golding | Incorporate changes for GitHub issues #158, #164, #172, #175, #178, and #186. |
| 18 | 2018/06/08 | Laurence J. Golding | Incorporate changes for GitHub issues #189 and #191. |
| 19 | 2018/11/14 | Laurence J. Golding | Incorporate changes for GitHub issues #169, #256, #269, #272, and #275. |
| 20 | 2018/11/29 | Laurence J. Golding | Incorporate changes for GitHub issues #186, #188, #274, #279, #280, #284, #285, and #288. |
| 21 | 2018/12/13 | Laurence J. Golding | Incorporate changes for GitHub issues #248, #270, #287, #292, #293, and #297. |
| 22 | 2019/01/10 | Laurence J. Golding | Incorporate changes for GitHub issues #286, #291, #303, and #304. |
| 23 | 2019/02/20 | Laurence J. Golding | Incorporate changes for GitHub issues #146, #312, #317, and #322. |
| 24 | 2019/03/15 | Laurence J. Golding | Incorporate changes for GitHub issues #168, #291, #309, #320, #321, #326, #330, #335, #340, and #341. |
| 25 | 2019/03/16 | Laurence J. Golding | Incorporate changes for GitHub issues #179, #319, and #337. |
| 26 | 2019/03/28 | Laurence J. Golding | Incorporate changes for GitHub issues #202, #302, #311, #314, #315, #318, #324, #325, #327, #338, #344, #346, #347, #348, and #350. |
| 27 | 2019/04/01 | Laurence J. Golding | Incorporate editorial changes for GitHub issues #106, #117, #301, and #342. |
| 28 | 2019/04/17 | Laurence J. Golding | Incorporate changes for GitHub issues #266, #323, #349, #353, #354, #355, #356, #357, #358, #359, #361, #362, #363, #364, #365, #366, #367, #368, #369, #370, #371, #372, #373, #374, #376, and #379. Address issues from Henny Sipma and Paul Anderson. |
| 29 | 2019/04/29 | Laurence J. Golding | Incorporate changes from GitHub issue #375, #376 (tail), #378, #380, #381, #382, #383, #387, #389, #390, #391, #392, #393, #396, #397, #399, #401, #402, #403, and #404. |
| 30 | 2019/05/10 | Laurence J. Golding | Incorporate changes from GitHub issue #405 (post-CSD.2 ballot, non-substantive editorial changes). |
| 31 | 2019/05/15 | Laurence J. Golding | Incorporate changes for GitHub issues #398, #406, #407, #408, #410, #411, #414, #415, #416, #417, and #418. |

  