Static Analysis Results Interchange Format (SARIF) Version 2.0

Committee Specification Draft 01 / Public Review Draft 01

15 June 2018

Specification URIs

This 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:

http://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.docx (Authoritative)
http://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html
http://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 Fanning (mikefan@microsoft.com), Microsoft
Laurence J. Golding (larrygolding@comcast.net), Individual Member

Additional artifacts:

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

- JSON schemas: http://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/schemas/sarif-schema.json

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/.

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. OASIS Committee Specification Draft 01/Public Review Draft 01. http://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/sarif-v2.0-csprd01.html. Latest version: http://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html.

Notices

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

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

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

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

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark for above guidance.

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. 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).

1.1 IPR Policy

This specification is being developed 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 [BCP14] [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

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

analysis target

programming artifact which a static analysis tool is instructed to analyze

artifact

see programming artifact

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 file

file considered as a sequence of bytes

binary region

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

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

1-based index of a character within a line

converter

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

deterministic producer

SARIF producer which, given identical inputs, repeatedly produces an identical SARIF log file

direct producer

analysis tool which acts as a SARIF producer

embedded link

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

embedded resource

resource that is contained within a SARIF log file

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

external resource

resource that is contained within a SARIF resource file

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 programming artifact in which it occurs is modified

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

localization

process of adapting a collection of resources to a language, region, or culture

log file

output file produced by a static 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 programming artifact in which it occurs

logical location

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

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

logical name

string that partially identifies the programmatic construct specified by a logical location, typically by specifying 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

file which is contained within another file

nested logical location

logical location that is nested 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.

parent (file)

file which contains one or more nested files

physical location

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

plain text message

message string which does not contain any formatting information

(programming) artifact

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

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

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 object consisting of a set of properties with arbitrary camelCase names

redaction-aware 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 file

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

resource

item that requires localization, such as a message string or rule metadata

result

condition present in a programming artifact and reported by a static analysis tool

result file

file in which a static 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

rule

specific criterion for correctness verified by a static analysis tool

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

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

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

rule configuration information

rule metadata that a tool can modify at runtime, before executing its scan

rule id

stable value which a static analysis tool associates with a rule

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

Example: `CA2001`

rule metadata

information that describes a rule

Example: id, description, category, author

run

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

2. set of results produced by such an invocation

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 resources for a single language, in the format defined by the SARIF specification

stable value

value which, once established, never changes over time

(static analysis) tool

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

Example: Lint

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

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

text region

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

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

top-level file

file which is not contained within any other file

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++

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.

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.

[ISO639-1] “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, https://www.iso.org/standard/4767.html.

[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, https://www.iso.org/standard/39534.html.

[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.

[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", RFC 7763, DOI 10.17487/RFC7763, March 2016, http://www.rfc-editor.org/info/rfc7763.

[RFC7764] Leonard, S., "Guidance on Markdown: Design Philosophies, Stability Strategies, and Select Registrations", RFC 7764, DOI 10.17487/RFC7764, March 2016, http://www.rfc-editor.org/info/rfc7764.

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

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

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

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 SARIF format. The examples are formatted for clarity, as permitted by [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 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.

2.3 Property notation

A JSON 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:

1. 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 };
```

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

```
? JSON string character ?
```

3 File format

3.1 General

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

A SARIF log file **SHALL** conform to the requirements of the JSON format. The top-level value in the log file **SHALL** conform to the JSON object grammar; that is, it **SHALL** consist of a comma-separated sequence of name/value pairs, enclosed in curly brackets, as described in [RFC8259]. We refer to the object represented by this top-level value as the `sarifLog` object (§3.10).

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

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

3.2 fileContent objects

3.2.1 General

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

3.2.2 text property

If the external file is a text file, a `fileContent` 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 file is a text file 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 [RFC8259] requires to be escaped.

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

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

3.2.3 binary property

If the external file is a binary file, or if the SARIF producer cannot determine whether the external file is a text file or a binary file, a `fileContent` 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.

If the external file is a text file 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 is a UTF-8 text file, 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 fileLocation objects

3.3.1 General

Certain properties in this specification specify the location of a file. SARIF represents a file location with a `fileLocation` object. The most important member of a `fileLocation` object is its `uri` property (§3.3.2). If the `uri` property contains a relative reference (the term used in [RFC 3986] for what is commonly called a "relative URI"), the `uriBaseId` property (§3.3.3) can sometimes be used to resolve the relative reference to an absolute URI.

3.3.2 uri property

3.3.2.1 General

A `fileLocation` object **SHALL** 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` (§3.11.13).

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

> NOTE 2: For example, in the normalized form specified in RFC 3986:
>
> - Percent-encoded characters use upper-case hexadecimal digits.
> - Characters in the ALPHA and DIGIT ranges are not be percent-encoded, nor are hyphen, underscore, or tilde.
> - The "`:`" delimiter is omitted if the port component of the authority is empty.
> - In the host component, registered names and hexadecimal addresses use lower-case.

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 from such a URI, for example, for security reasons. If it does so, then to maximize interoperability with previous version of the URI specification, it **SHOULD** specify the URI with leading `"//"`, as in EXAMPLE 2. See [RFC8089] for more information on this point.

3.3.3 uriBaseId property

If the value of its `uri` property (§3.3.2) is a relative reference, a `fileLocation` 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.

To avoid ambiguity in interpreting the property names (§3.11.13.2) in `run.files` (§3.11.13), the `uriBaseId` property **SHALL NOT** contain the character `"#"`.

If a SARIF consumer requires an absolute URI (for example, to display the specified file to a user), then it needs to have the necessary information to resolve the `uriBaseId` property to an absolute URI, which can then be combined with the relative reference stored in the `uri` property. One possibility is for the SARIF producer and consumers to agree on the meanings of any values 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 on the consumer's command line, or through a user interface prompt.

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"
```

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 `fileLocation` object (§3.3) 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": {
  "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 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 §3.3.4.

3.3.4 Guidance on the use of fileLocation 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` objects represent non-deterministic URIs. **The** `uri` property (§3.3.2) **SHOULD** contain a relative reference that *is* deterministic, for example, the relative path from the root of a source code enlistment to the file. The `uriBaseId` property (§3.3.3) **SHOULD** capture the non-deterministic portion of the URI, for example, the absolute path to the root of the 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 (§3.11).
  "originalUriBaseIds": {                            # See §3.11.12.
    "SRCROOT": "file:///C:/browser/src"
  },

  "results": [                                       # See §3.11.16.
    {                                                # A result object (§3.19).
      "locations": [                                 # See §3.19.10.
        {                                            # A location object (§3.20).
          "physicalLocation": {                        # See §3.20.2.
            "fileLocation": {                        # A fileLocation object.
              "uri": "ui/window.cpp",
              "uriBaseId": "SRCROOT"
            }
          }
        }
      ]
    }
  ]
}
```

3.4 String properties

3.4.1 General

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

Certain string-valued properties in this specification (for example, `invocation.commandLine` (§3.13.2)) 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.” The description of every redaction-aware property will state that it is redaction-aware.

If a SARIF producer or a SARIF post-processor chooses to redact sensitive information in a redaction-aware property, it **SHALL** replace the sensitive information with the string whose value is provided by the `run.redactionToken` property (§3.11.21).

3.4.3 GUID-valued string properties

Certain string-valued properties in this specification provide unique stable identifiers in the form of a GUID [RFC4122].

> EXAMPLE: `"f81d4fae-7dec-11d0-a765-00a0c91e6bf6"`
>
> NOTE: RFC4122 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.

3.4.4 Hierarchical strings

3.4.4.1 General

Certain string-valued properties and certain property names in this specification (for example, the value of the `run.automationLogicalId` property (§3.11.6), and the property names in a property bag (§3.7)) 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 ? - "/";
```

For examples, see §3.7.2 and §3.11.6.

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: 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.

3.4.4.2 Versioned hierarchical strings

Certain hierarchical strings in this specification (for example, the property names in `result.fingerprints` (§3.19.12) and `result.partialFingerprints` (§3.19.13)) 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: 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 (§3.19).
  "partialFingerprints": {         # See §3.19.13.
    "prohibitedWordHash": "4efcc21977b55",
    "prohibitedWordHash/v1": "097886bc876fe"
  }
}
```

> NOTE: 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.

3.5 Object properties

Certain properties in this specification are defined to be JSON objects whose property names satisfy certain conditions. Examples are `run.files` (§3.11.13) and `rule.messageStrings` (§3.36.7). 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.

3.6 Array properties

3.6.1 General

Certain properties in this specification are defined to be JSON arrays. Examples are the `invocation.toolNotifications` property (§3.13.20) and the `file.hashes` property (§3.17.10). 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.

3.6.2 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 [JSCHEMA01], §4.3, "Instance equality". In particular, two strings are considered equal when they consist of the same sequence of Unicode [UNICODE10] code points.

3.7 Property bags

3.7.1 General

Certain properties in this specification are defined to be “property bags”. A property bag is a JSON object (§3.5) containing an arbitrary set of properties.

The property names are hierarchical strings (§3.4.4). 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.

3.7.2 Tags

3.7.2.1 General

If a property bag contains a property named `tags`, the property value **SHALL** be an array of zero or more unique strings (§3.6.2). Two strings **SHALL** be considered the same if they consist of the same sequence of Unicode [UNICODE10] code points.

The strings in the `tags` array are hierarchical (§3.4.4).

EXAMPLE: In this example, the SARIF producer categorizes scan results according to the Common Weakness Enumeration taxonomy [CWE].

```
{                                 # A result object (§3.19).
  "ruleId": "CA2124",
  ...
  "properties": {
    "tags": [
      "CWE/22"
    ]
  }
}
```

3.7.2.2 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 in the same property bag that contains the tag, or in the property bag of any SARIF element which lexically contains the element containing the tag.

EXAMPLE: Suppose a SARIF-producing tool classifies results according to the Common 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 (§3.19)
  "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. It might provide that information within the property bag containing the tag (in this example, the property bag belonging to the `result` object):

```
{                                 # A result object (§3.19)
  "ruleId": "SEC0251",
  "message": {
    "text": "The path 'data/../bin' is not within the 'data' directory"
  },
  "properties": {
    "tags": [
      "security",
      "CWE/22"
    ],
    "CWE/22": {
      "description": "Improper Limitation of a Pathname",
      "url": "https://cwe.mitre.org/data/definitions/22.html"
    }
  }
}
```

However, there might be several results associated with CWE 22. To avoid duplicating the metadata, the tool might choose to place it in the property bag belonging to the `run` object (§3.11) that lexically contains the `result` object:

```
{                                 # A run object (see §3.11)
  "results": [
    {
      "ruleId": "SEC0251",
      "message": {
        "text": "The path 'data/../bin' is not within the 'data' directory"
      },
      "properties": {
        "tags": [
          "security",
          "CWE/22"
        ]
      }
    }
  ],
  "properties": {                 # The run object's property bag.
    "CWE/22": {
      "description": "Improper Limitation of a Pathname",
      "url": "https://cwe.mitre.org/data/definitions/22.html"
    }
  }
}
```

3.8 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 [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 12 *);

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

A SARIF producer **SHOULD** base the number of digits in `fraction` on the precision of the clock on the computer on which it runs.

3.9 message objects

3.9.1 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”).
- Message strings that incorporate formatting information (“rich text messages”).
- Message strings with placeholders for variable information.
- Localized message strings.

3.9.2 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** contain JSON escaped line breaks (`"\r"` or `"\n"`). However, if 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 (§3.9.4) and embedded links (§3.9.5).

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.

3.9.3 Rich text messages

3.9.3.1 General

Rich text messages **MAY** be of arbitrary length and **SHOULD** contain formatting information. The message string **MAY** also contain placeholders (§3.9.4) and embedded links (§3.9.5).

Every rich text message in a given run **SHALL** be expressed in the same markup language, specified by the `run.richMessageMimeType` property (§3.11.20). For maximum interoperability among SARIF log files produced by different tools, direct producers **SHALL** express rich text messages 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 Security implications

If the rich text message format is any variant of Markdown, then 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. 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 messages **SHALL NOT** attempt to render them and **SHALL** instead fall back to the corresponding plain text messages.

3.9.4 Messages with placeholders

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

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

index = non negative integer
```

`index` represents a 0-based index into the array of strings contained in the `arguments` property (§3.9.11).

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 (§3.9.11). Within both plain text and rich text 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 message strings **MAY** contain different numbers of placeholders.
- A given placeholder index **SHALL** have the same meaning across all the 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 (§3.9.7) 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 (§3.11).
  "results": [                                       # See §3.11.16.
    {                                                # A result object (§3.19).
      "ruleId": "CA2101",                            # See §3.19.6.
      "message": {                                   # See §3.19.8.
        "text": "Variable '{0}' is uninitialized.",  # See §3.9.7.
        "arguments": [ "pBuffer" ]                   # See §3.9.11.
      }
    }
  ]
}
```

3.9.5 Messages with embedded links

A message string **MAY** include one or more links to locations within files mentioned in the enclosing `result` object (§3.19). We refer to these links as “embedded links”.

Within a rich text message (§3.9.3), 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: The GFM link syntax is very flexible. Since a SARIF viewer that renders rich text messages will presumably rely on a full-featured GFM processor, there is no need to restrict the embedded link syntax in SARIF rich text messages.

Within a plain text message (§3.9.2), 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;

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.

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 §3.9.7
}
```

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` object (§3.19) **SHALL** contain exactly one `physicalLocation` object (§3.21) whose `id` property (§3.21.2) equals the value of `link destination`.

> NOTE: `link destination` is required to be an integer, rather than arbitrary string, to avoid confusion with normal Markdown link syntax. 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 is exactly one `physicalLocation` object whose `id` property matches the `link destination`.

```
{
  "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": [
            {
              "physicalLocation": {
                "id": 3,
                "uri": "file:///C:/code/input.c",
                "region": {
                  "startLine": 15,
                  "startColumn": 9
                }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

3.9.6 Message string resources

3.9.6.1 General

A `message` object can directly contain message strings in its `text` (§3.9.7) and `richText` (§3.9.8) properties. It can also indirectly refer to message strings through its `messageId` (§3.9.9) and `richMessageId` (§3.9.10) 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 (§3.11) can optionally contain the message string resources for a single language, namely the language designated by its `tool.language` property (§3.12.8). We refer to these message strings as "embedded resources." Embedded message string resources are stored in the `run.resources.messageStrings` property (§3.35.2).

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." §3.9.6.3 defines the naming convention and file lookup procedure for SARIF resources files. §3.9.6.4 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 language, 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 messages.

If the consumer can render rich text messages, the string lookup procedure is:

1. If `message.richText` is present, use its value.
2. Otherwise, if `message.richMessageId` is present, and `run.resources.messageStrings` is present and contains a property whose name matches `message.richMessageId`, use the value of that property.
3. Otherwise, execute the lookup procedure for plain text messages, below.

If the consumer cannot render rich text messages, the string lookup procedure is:

1. If `message.text` is present, use its value.
2. Otherwise, if `message.messageId` is present, and `run.resources.messageStrings` is present and contains a property whose name matches `message.messageId`, use the value of that property.
3. Otherwise, the string 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` (§3.12.9) 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 [RFC5646] 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 [ISO639-1], [ISO639-2] or, [ISO639-3], 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 §3.9.6.2. 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 §3.9.6.2 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.

3.9.6.4.2 sarifLog object

The root element of a SARIF resource file is a `sarifLog` object (§3.10). 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.

3.9.6.4.3 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.

3.9.6.4.4 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.

3.9.6.4.5 resources object

The `resources` object in a SARIF resource file is identical to the `resources` object in a SARIF log file (§3.35).

3.9.7 text property

A `message` object **MAY** contain a property named `text` whose value is a non-empty string containing a plain text message (§3.9.2).

3.9.8 richText property

A `message` object **MAY** contain a property named `richText` whose value is a non-empty string containing a rich text message (§3.9.3).

If the `richText` property is present, the `text` property (§3.9.7) **SHALL** also be present. This ensures that the message is viewable even in contexts that do not support the rendering of rich text.

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

3.9.9 messageId property

A `message` object **MAY** contain a property named `messageId` whose value is a non-empty string containing the resource identifier (§3.9.6) for the desired plain text message (§3.9.2). See §3.9.6.2 and §3.9.6.3 for details of the resource string lookup procedure.

3.9.10 richMessageId property

A `message` object **MAY** contain a property named `richMessageId` whose value is a non-empty string containing the resource identifier (§3.9.6) for the desired rich text message (§3.9.3).

SARIF consumers that cannot (or choose not to) render rich text **SHALL** ignore the `richMessageId` property and use the `messageId` property instead. See §3.9.6.2 and §3.9.6.3 for details of the resource string lookup procedure.

3.9.11 arguments property

If the message string specified by any of the properties `text` (§3.9.7), `richText` (§3.9.8), `messageId` (§3.9.9), or `richMessageId` (§3.9.10) contains any placeholders (§3.9.4), the `message` object **SHALL** contain a property named `arguments` whose value is an array of strings. §3.9.4 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`, or `richMessageId` contains any placeholders, the arguments property **SHALL** 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` properties.

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

3.10 sarifLog object

3.10.1 General

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

EXAMPLE:

```
{
  "version": "2.0.0", # See §3.10.2.
  "runs": [           # See §3.10.4.
```

```
    {
      ...              # A run object (§3.11)
    },
    ...
    {
      ...              # Another run object
    }
  ]
}
```

3.10.2 version property

A `sarifLog` object **SHALL** contain a property named `version` whose value is a string designating the version of the SARIF format to which this log file conforms. This string **SHALL** have the value `"2.0.0"`.

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.

3.10.3 $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 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 (§3.10.2).

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

3.10.4 runs property

A `sarifLog` object **SHALL** contain a property named `runs` whose value is an array of one or more `run` objects (§3.11).

3.11 run object

3.11.1 General

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

EXAMPLE:

```
{
  "tool": {        # See §3.11.8.
       ...         # A tool object (§3.12).
  },
  "results": [     # See §3.11.16.
    {
      ...          # A result object (§3.19).
    },
    ...
    {
      ...          # Another result object.
    }
  ]
}
```

3.11.2 instanceGuid property

A `run` object **MAY** contain a property named `instanceGuid` whose value is a GUID-valued string (§3.4.3) which provides a unique, stable identifier for the run.

A result management system or other components 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.

3.11.3 logicalId property

A `run` object **MAY** contain a property named `logicalId` 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 (§3.4.4).

EXAMPLE 2:

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

An 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.

3.11.4 description property

An `run` object **MAY** contain a property named `description` whose value is a `message` object (§3.9) that describes this run.

If `logicalId` (§3.11.3) is present, description **SHOULD** describe the type of run defined by `logicalId`.

EXAMPLE:

```
{                                                    # A run object (§3.11).
  "logicalId": "Nightly security scanner run/x86/debug", # See §3.11.3.
  "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"
    ]
  }
}
```

3.11.5 baselineInstanceGuid property

A `run` object **MAY** contain a property named `baselineInstanceGuid` whose value is a GUID-valued string (§3.4.3) which **SHALL** equal the `instanceGuid` property (§3.11.2) of some previous run.

If the `run` object has a `logicalId` property (§3.11.3), then the run identified by `baselineInstanceGuid` **SHALL** have the same value for `logicalId`.

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

If `baselineInstanceGuid` is present, the `result.baselineState` property (§3.19.20) of every `result` object (§3.19) in the containing `run` object **SHALL** be computed with respect to the run specified by `baselineInstanceGuid`.

3.11.6 automationLogicalId property

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

`automationLogicalId` is hierarchical (§3.4.4).

> 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",
  ...
}
```

3.11.7 architecture property

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

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

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

3.11.8 tool property

A `run` object **SHALL** contain a property named `tool` whose value is a `tool` object (§3.12) that describes the analysis tool that was run.

3.11.9 invocations property

A `run` object **MAY** contain a property named `invocations` whose value is an array of unique (§3.6.2) `invocation` objects (§3.13) that describe the invocation of the analysis tool that was run.

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 to analyze and another program to analyze those files.

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.

3.11.10 conversion property

If a `run` object was produced by a converter, it **MAY** contain a property named `conversion` whose value is a `conversion` object (§3.14) 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.

3.11.11 versionControlProvenance property

A `run` object **MAY** contain a property named `versionControlProvenance` whose value is an array of one or more unique (§3.6.2) `versionControlDetails` objects (§3.16). 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 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 (§3.16).
      "uri": "https://github.com/example/browser",   # See §3.16.3.
      "revisionId": "fd3fbae"                        # See §3.16.4.
      "branch": "master"                             # See §3.16.5.
    }
  ]
}
```

3.11.12 originalUriBaseIds property

A `run` object **MAY** contain a property named `originalUriBaseIds` whose value is a JSON object (§3.5) each of whose property names designates a URI base id (§3.3.3). The value of each property is an absolute URI [RFC3986] which is the value of that URI base id on the machine where the SARIF producer ran.

This property allows SARIF consumers to resolve any relative references which appear in any `fileLocation` objects (§3.2) 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.

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.

> EXAMPLE: In this example, the URI base id `"SRCROOT"` on the machine where the SARIF producer ran was `"file:///C:/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:\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
  "originalBaseIds": {
    "SRCROOT": "file:///C:/src"
  },

  "results": [
    {                                              # A result object (§3.19)
      "ruleId": "CA1001",
      "locations": [
        {                                          # A location object (§3.20)
          "physicalLocation": {                    # See §3.20.2.
            "fileLocation": {                      # A fileLocation object (§3.2)
              "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 §3.3.2.2).

3.11.13 files property

3.11.13.1 General

A `run` object **SHOULD** contain a property named `files` whose value is a JSON object (§3.5) each of whose properties represents a file relevant to the run.

The object specified by the `files` property **SHOULD** contain properties representing at least those files in which results were detected, but it **MAY** contain properties representing all files examined by the tool (whether or not results were detected in those files), or any subset of those files. It **MAY** also include other files relevant to the run, such as attachments (§3.13.5, §3.19.21).

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

EXAMPLE:

```
"files": {
  "file:///C:/Code/main.c": {
    "mimeType": "text/x-c",
    "hashes": [
      {
        "value": "b13ce2678a8807ba0765ab94a0ecd394f869bc81",
        "algorithm": "sha-256"
      }
    ]
  }
}
```

3.11.13.2 Property names

The property names in the `files` object are related to the file locations specified in `fileLocation` objects (§3.2) 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 (§3.3.2) 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 §3.3.2.

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 (§3.11).
  "results": [
    {                              # A result object (§3.19).
      "relatedLocations": [
        {                          # A location object (§3.34).
          "physicalLocation": {    # A physicalLocation object (§3.21).
            "fileLocation": {      # A fileLocation object (§3.2).
              "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 §3.3.2.

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 (§3.11).
  "results": [
    {                              # A result object (§3.19).
      "relatedLocations": [
        {                          # A location object (§3.20).
          "physicalLocation": {    # A physicalLocation object (§3.21).
            "fileLocation": {      # A fileLocation object (§3.2).
              "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 (§3.3.3), 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 (§3.11).
  "results": [
    {                               # A result object (§3.19).
      "relatedLocations": [
        {                           # A location object (§3.20).
          "physicalLocation": {     # A physicalLocation object (§3.21).
            "fileLocation": {       # A fileLocation object (§3.2).
              "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 [RFC3986].

> 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 (§3.11)
```

```
  "results": [
    {                              # A result object (§3.19)
      "relatedLocations": [
        {                          # A location object (§3.20)
          "physicalLocation": {    # A physicalLocation object (§3.21)
            "fileLocation": {      # A fileLocation object (§3.3)
              "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 §3.3.2. 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.
    ...
  }
}
```

3.11.13.3 Property values

Each property value in the `files` object **SHALL** be a `file` object (§3.17) which contains information about the file identified by the property name (§3.11.13.2).

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

If the file is a nested file, then the property name **SHALL** 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 representing each level of nesting **MAY** be combined in any way desired, as long as no two of the resulting property names are equivalent as defined in §3.3.2.

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 (§3.17.2), or in the `offset` (§3.17.4) and `length` (§3.17.5) 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 [RFC3986], those character **SHALL** be percent-encoded as specified in [RFC3986].

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 §3.17.

```
"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 §3.17.3
  },
  "file:///C:/Code/app.zip#/docs/intro.docx/Flash1": {
    "offset": 17522,
    "length": 4050,
    "mimeType": "application/x-shockwave-flash",
    "parentKey": "file:///C:/Code/app.zip#/docs/intro.docx"
  }
}
```

3.11.14 logicalLocations property

Depending on the circumstances, a `run` object either **MAY** or **SHOULD** contain a property named `logicalLocations` whose value is a JSON object (§3.5) each of whose properties 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 §3.20.3, each property name in the `logicalLocations` object **SHALL** be the fully qualified name of the logical location. See §3.20.3 for examples. The property names **SHALL** follow the naming rules for fully qualified logical names described in §3.24.2.

Each property value in the `logicalLocations` object **SHALL** be a `logicalLocation` object (§3.24).

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

If a nested logical location appears in the `logicalLocations` object, then the `logicalLocations` object **SHALL** also contain properties describing each of its parents, up to and including the top-level logical location.

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

```
"logicalLocations": {
  "namespaceA::namespaceB::classC": {
    "name": "classC",
    "kind": "type",
    "parentKey": "namespaceA::namespaceB"
  },
"namespaceA::namespaceB": {
    "name": "namespaceB",
    "kind": "namespace"
    "parentKey": "namespaceA"
  },
  "namespaceA": {
    "name": "namespaceA",
    "kind": "namespace"
  }
}
```

NOTE: The detailed information in `logicalLocations` is useful, even though much of it is captured in `location.fullyQualifiedLogicalName` (§3.20.3), 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` string.

3.11.15 graphs property

A `run` object **MAY** contain a property named `graphs` whose value is an array of one or more unique (§3.6.2) `graph` objects (§3.27) each of which 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 (§3.30) defined in the `graphTraversals` property (§3.19.16) of any `result` object (§3.19) in the `run`.

3.11.16 results property

A `run` object **SHALL** contain a property named `results` whose value is an array of zero or more `result` objects (§3.19), each of which represents a single result detected in the course of the run.

> NOTE: The `results` array is not defined to contain unique (§3.6.2) 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 `results` array **SHALL** be empty if the tool invocation that produced the `run` object did not detect any results.

3.11.17 resources property

A `run` object **MAY** contain a property named `resources` whose value is a `resources` object (§3.35). A `resources` object represents items that can be localized, such as resource strings and rule metadata.

3.11.18 defaultFileEncoding

A `run` object **MAY** contain a property named `defaultFileEncoding` whose value is a string that provides a default for the `encoding` property (§3.17.9) of any `file` object (§3.17) in `run.files` (§3.11.13) that refers to a text file. The string **SHALL** be one of the character set names specified in [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` object that does not contain an `encoding` property **SHALL** be taken to be unknown.

For an example, see §3.17.9.

3.11.19 columnKind property

If a SARIF producer processes text files, 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.

`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 files, `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 file contents (for example, by displaying the file in an editor and highlighting a region), the consumer **SHALL** recompute column numbers in its (the consumer's) native measurement unit.

3.11.20 richMessageMimeType property

A `run` object **MAY** contain a property named `richMessageMimeType` whose value is a string that specifies the MIME type [RFC2045] of all rich text message properties (§3.9.3) in the run. If this property is absent, it **SHALL** default to `"text/markdown;variant=GFM"`. [RFC7763] defines the `"text/markdown"` media type, and [RFC7764] registers `"GFM"` as the value of the variant parameter which specifies GitHub-Flavored Markdown [GFM].

For a discussion of the security implications of expressing rich text messages in GFM, see §3.9.3.2.

3.11.21 redactionToken property

If the value of any redaction-aware property (§3.4.2) in the run has been redacted, the `run` object **SHALL** contain a property named `redactionToken` whose value is the string used to replace the redacted text. If no text in the run has been redacted, the `redactionToken` property **SHALL** be absent.

The value of `redactionToken` **SHOULD** be the string `"[REDACTED]"`. If for any reason a different value is used, it **MAY** be any readily identifiable string. 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-aware property in the run.

EXAMPLE 1: In this example, the leading portion of a full path name has been redacted from the redaction-aware property `invocation.commandLine` to avoid revealing information about the machine.

```
{                     # A run object.
  "redactionToken": "[REDACTED]",

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

3.11.22 properties property

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

3.12 tool object

3.12.1 General

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

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

EXAMPLE:

```
{
  "name": "CodeScanner",                                        # see §3.12.2
  "fullName": "CodeScanner 1.1, Developer Preview (en-US)",     # see §3.12.3
  "semanticVersion": "1.1.2-beta.12",                           # see §3.12.4
  "version": "1.1.2b12,                                         # see §3.12.5
  "fileVersion": "1.1.1502.2"                                   # see §3.12.6
}
```

3.12.2 name property

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

EXAMPLE: `"CodeScanner"`

3.12.3 fullName property

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

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

3.12.4 semanticVersion property

In a log file produced by an analysis tool, a tool object **MAY** contain a property named `semanticVersion` whose value is a string containing the tool version in a format that conforms to the syntax and semantics specified by [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.

A converter **SHALL NOT** emit the `semanticVersion` property.

NOTE 2: The rationale is that an analysis tool knows whether its version string is intended to be interpreted according to SemVer. A converter will in general not know this, even if the tool's version string conforms to the pattern specified by SemVer.

3.12.5 version property

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

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

3.12.6 fileVersion property

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

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

3.12.7 downloadUri property

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

3.12.8 language property

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

The `language` property specifies:

1. The language of the message strings contained in the `text` (§3.9.7) and `richText` (§3.9.8) properties of any `message` object (§3.9) in the containing `run` object (§3.11).
2. The language of any embedded resources (§3.9.6) contained in the `resources` property (§3.11.17) of the containing `run` object.

3.12.9 resourceLocation property

If a SARIF producer provides external resources (§3.9.6) for languages other than the tool's declared language (§3.12.8), the `tool` object **SHALL** contain a property named `resourceLocation` whose value is a `fileLocation` object (§3.2) which specifies the location of a directory containing the tool's SARIF resource files.

If a SARIF producer does not provide external resources, the `resourceLocation` property **SHALL** be absent.

If the `fileLocation` object's `uri` property (§3.3.2) specifies a relative reference, then its `uriBaseId` property (§3.3.3) **SHOULD** be present, and the `run` object's `originalUriBaseIds` property (§3.11.12) **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.

```
{                                                    # A run object (§3.11).
  "tool": {
    "name": "SecurityScanner",
    "version": "2.0.1",
    "resourceLocation": {                            # A fileLocation object (§3.2).
      "uri": "resources",
      "uriBaseId": "TOOLINSTALLDIR"
    }
  },

  "originalUriBaseIds": {                            # See §3.11.12.
    "TOOLINSTALLDIR": "file:///C:/Program%20Files/SecurityScanner/2.0.1"
  }
}
```

EXAMPLE 2: In this example, the SARIF resource files are available on the analysis tool's web site.

```
{                                                    # A run object (§3.11).
  "tool": {
    "name": "SecurityScanner",
    "version": "2.0.1",
    "resourceLocation": {                            # A fileLocation object (§3.2).
      "uri": ".",
      "uriBaseId": "RESOURCES"
    }
  },

  "originalUriBaseIds": {                            # See §3.11.12.
    "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.

3.12.10 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.

3.12.11 properties property

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

3.13 invocation object

3.13.1 General

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

3.13.2 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.

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

The `commandLine` property is redaction-aware (§3.4.2) 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 /"
}
```

3.13.3 arguments property

An `invocation` object **MAY** contain a property named `arguments` whose value is an array of strings, containing in order the command line arguments passed to the tool from the operating system.

EXAMPLE: If the tool is implemented as a C# or Java program, `arguments` would contain the contents of the `args` array passed to entry point method.

NOTE: Although the `commandLine` property (§3.13.2) contains the same information, parsing it is error prone even if one understands the quoting and escaping conventions. SARIF consumers might find the pre-parsed `arguments` property easier to use.

3.13.4 responseFiles property

An `invocation` object **MAY** contain a property named `responseFiles` whose value is an array of `fileLocation` objects (§3.3), each of which represents a response file specified on the tool's command line.

A SARIF producer **MAY** embed the contents of a response file in the SARIF log file by mentioning the response file in `run.files` (§3.11.13) and providing a value for `file.contents` (§3.17.8).

EXAMPLE:

```
{                       # An invocation object.
    "commandLine": "/quiet @analyzer.rsp @strict.rsp" @options.rsp,

    "responseFiles": [
      {                 # A fileLocation object (§3.3)..
        "uri": "analyzer.rsp",
        "uriBaseId": "RESPONSEFILEDIR"
      },
      {
        "uri": "strict.rsp",
        "uriBaseId": "RESPONSEFILEDIR"
      },
      {
        "uri": "options.rsp",
        "uriBaseId": "RESPONSEFILEDIR"
      }
    }
    ...
}
```

3.13.5 attachments property

An `invocation` object **MAY** contain a property named `attachments` whose value is an array of one or more unique (§3.6.2) `attachment` objects (§3.14). Each `attachment` object **SHALL** describe a file relevant to the invocation of the tool. Typically, these would be files specified on the tool's command line, and therefore mentioned in the `commandLine` property (§3.13.2) or the `arguments` property (§3.13.3), if present. They might also be files implicitly consumed by the tool, such as a configuration file.

For an example, see EXAMPLE 1 in §3.14.1.

3.13.6 startTime property

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

3.13.7 endTime property

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

3.13.8 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 §3.13.9.

3.13.9 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"
}
```

3.13.10 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 §3.13.11.

3.13.11 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"
}
```

3.13.12 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."
}
```

3.13.13 toolExecutionSuccessful property

An `invocation` object **SHOULD** contain a property named `toolExecutionSuccessful` 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. 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 (§3.13.8) is present and has a non-zero value; otherwise it **SHALL** default to `true`.

EXAMPLE:

```
{
  "exitCode": 1,
  "exitCodeDescription": "Scan successful; warnings detected.",
  "toolExecutionSuccessful": true
}
```

3.13.14 machine property

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

3.13.15 account property

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

3.13.16 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.

3.13.17 executableLocation property

An `invocation` object **MAY** contain a property named `executableLocation` whose value is a `fileLocation` object (§3.3) specifying the absolute URI of the tool's 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 §3.3.4 for non-deterministic absolute URIs.

> NOTE 1: This property is defined in the `invocation` object rather than in the `tool` object (§3.12) because the identical tool might be invoked from different paths on different machines.
>
> NOTE 2: This property might duplicate information in the `commandLine` property (§3.13.2). 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.

3.13.18 workingDirectory property

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

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

3.13.19 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 variable names and values are likely to reveal highly sensitive information. For example, on a Windows machine, environment variables reveal the directories on the execution path, user account name, machine name, logon domain controller, *etc.*
>
> NOTE 2: The result of setting an environment variable to an empty string is operating system-dependent. On Windows, it removes the variable from the environment. In Unix, an environment variable can have an empty value.

3.13.20 toolNotifications property

A `configuration` object **MAY** contain a property named `toolNotifications` whose value is an array of zero or more `notification` objects (§3.41). Each element of the array represents a runtime condition detected by the invoked process. The presence within this array of any `notification` object whose level property (§3.41.6) is `"error"` **SHALL** mean that the run failed.

The information in `toolNotifications` 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 `toolNotifications`, disable the rule, and continue to execute the remaining rules.
>
> If the error occurs outside of the evaluation of a rule, the tool might report the error in `toolNotifications` and then halt. If the tool exits abnormally, it might not have the opportunity to report the error.

3.13.21 configurationNotifications property

A `configuration` object **MAY** contain a property named `configurationNotifications` whose value is an array of zero or more `notification` objects (§3.41). Each element of the array represents a condition relevant to the tool's configuration. The presence within this array of any `notification` object whose `level` property (§3.41.6) is `"error"` **SHALL** mean that the run failed.

The information in `configurationNotifications` is primarily intended for the engineers who configure the analysis tool, to aid them in diagnosing errors in the configuration. This 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 report the problem in `configurationNotifications`. The tool might continue to run, reporting results for the rules that are correctly configured.

```
"configurationNotifications": [
  {
    "id": "UnknownRule",
    "ruleId": "ABC0001",
    "level": "warning",
    "message": {
      "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 report the problem in `configurationNotifications`. The tool might report the problem as a warning and continue to run, or it might report the problem as an error and terminate.

```
"configurationNotifications": [
  {
    "id": "UnknownCommandLineArgument",
    "level": "error",
    "message": {
      "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 the problem as an error in `configurationNotifications` and then terminate.

```
"configurationNotifications": [
  {
    "id": "CannotFindRulePlugin",
    "level": "error",
    "message": {
      "text": "Cannot find rule plugin \"C:\\AnalysisTool\\CustomChecks.dll."
    }
  }
]
```

3.13.22 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` objects (§3.3) 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` (§3.11.13) and providing a value for `file.contents` (§3.17.8).

3.13.23 properties property

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

3.14 attachment object

3.14.1 General

An `attachment` object describes a file relevant to the invocation of a tool (see §3.13.5) or to the detection of a result (see §3.19.21).

A SARIF producer **MAY** embed the contents of an attachment in the log file by mentioning the attachment file in `run.files` (§3.11.13) and providing a value for `file.contents` (§3.17.8).

EXAMPLE 1: In this example, `.scanrc` is the configuration file for the tool being run:

```
{                                                  # A run object (§3.11).
  "invocations": [                                 # See §3.11.9.
    {                                              # An invocation object (§3.13).
      ...
      "attachments": [                             # See §3.13.5.
        {                                          # An attachment object.
          "description": {                         # See §3.14.2.
            "text": "Configuration file"
          },
          "fileLocation": {                        # See §3.14.3.
            "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 (§3.19).
  ...
  "attachments": [                                 # See §3.19.21.
    {                                              # An attachment object.
      "description": {                             # See §3.14.2.
        "text": "Screen shot"
      },
      "fileLocation": {                            # See §3.14.3.
        "uri": "file:///C:/ScanOutput/image001.png"
      }
    }
  ]
}
```

3.14.2 description property

An `attachment` object **SHOULD** contain a property named `description` whose value is a `message` object (§3.9) describing the role played by the attachment.

3.14.3 fileLocation property

An `attachment` object **SHALL** contain a property named `fileLocation` whose value is a `fileLocation` object (§3.2) that specifies the location of the attachment file.

3.14.4 regions property

An `attachment` object **MAY** contain a property named `regions` whose value is an array of one or more unique (§3.6.2) `region` objects (§3.22), each of which specifies a region of interest within the attachment. These `region` objects **SHOULD** contain a `message` property (§3.22.14) so a user can understand their relevance.

3.14.5 rectangles property

If the attachment is an image file (for example `.png` or `.svg`), an `attachment` object **MAY** contain a property named `rectangles` whose value is an array of one or more unique (§3.6.2) `rectangle` objects (§3.23), each of which specifies an area of interest within the image. These `rectangle` objects **SHOULD** contain a `message` property (§3.23.3) so a user can understand their relevance.

If the attachment is not an image file, `rectangles` **SHALL** be absent.

3.15 conversion object

3.15.1 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": {
        "name": "AndroidStudio"
      },
      "conversion": {
        "tool": {                                          # see §3.15.2
          "name": "SARIF SDK Multitool",
        },
                                                           # see §3.15.3
        "invocation": "Sarif.Multitool.exe convert -t AndroidStudio northwind.log"

        "analysisToolLogFileLocation": {                   # see §3.15.4
          "uri": "northwind.log",
          "uriBaseId": "$LOG_DIR$"
        }
      },
      "results": [
        ...
      ]
    }
  ]
}
```

3.15.2 tool property

A `conversion` object **SHALL** contain a property named `tool` whose value is a `tool` object (§3.12) that describes the converter.

3.15.3 invocation property

A `conversion` object **MAY** contain a property named `invocation` whose value is an `invocation` object (§3.13) that describes the invocation of the converter.

3.15.4 analysisToolLogFiles 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 analysis tool whose output was converted to SARIF produced any per-run files, the `conversion` object **MAY** contain a property named `analysisToolLogFiles` whose value is an array of one or more unique (§3.6.2) `fileLocation` objects (§3.2) that specify the locations of those files.

If the analysis tool did not produce any per-run files, `analysisToolLogFiles` **SHALL** be absent.

Per-result files are handled by the `result.conversionProvenance` property (§3.19.23).

3.16 versionControlDetails object

3.16.1 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` (§3.11).

For an example, see §3.11.11.

3.16.2 Constraints

A `versionControlDetails` object **SHALL** 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 might not suffice because a Git branch is a pointer to the latest commit along a line of development; however, `branch` together with `timestamp` might suffice (although that is not an idiomatic use of Git). Similarly, `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` might suffice.

3.16.3 uri property

A `versionControlDetails` object **SHALL** contain a property named `uri` whose value is a string containing an absolute URI [RFC3986] that specifies the location of the repository containing the scanned files.

3.16.4 revisionId property

A `versionControlDetails` object **SHOULD** contain a property named `revisionId` whose value is a string that uniquely and permanently identifies the appropriate revision of the scanned files.

3.16.5 branch property

A `versionControlDetails` object **MAY** contain a property named `branch` whose value is a string containing the name of a branch containing the correct revision of the scanned files.

3.16.6 tag property

A `versionControlDetails` object **MAY** contain a property named `tag` whose value is a 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` property is not an array. The purpose of `tag` is to aid in identifying a revision so that a scan can be reproduced, not to exhaustively describe the revision.

3.16.7 timestamp property

A `versionControlDetails` object **MAY** contain a property named `timestamp` 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 §3.8.

3.16.8 properties property

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

3.17 file object

3.17.1 General

A `file` object represents a single file.

3.17.2 fileLocation property

Depending on the circumstances, a `file` object either **SHALL**, **MAY**, or **SHALL NOT** contain a property named `fileLocation` whose value is a `fileLocation` object (§3.2).

If the `file` object represents a top-level file, then `fileLocation` **MAY** be present. If it is present, the value of its `uri` property (§3.3.2) **SHALL** equal the name of the property within `run.files` (§3.11.13) 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 `file` object represents a nested file whose location relative to its parent can be expressed only by means of a path, then the `fileLocation` property **SHALL** be present, and the value of its `uri` property **SHALL** be a relative reference [RFC3986] expressing that path.

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

If the `file` object represents a nested file whose location within its parent can be expressed either by means of a path or by means of a byte offset from the start of the parent, then either the `fileLocation` property or the `offset` property (§3.17.4) 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 §3.17.3
  }
}
```

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"
  }
}
```

3.17.3 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` (§3.11.13).

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.

3.17.4 offset property

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

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

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

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

If the `file` object represents a nested file whose location within its parent can be expressed either by means of a path or by means of a byte offset from the start of the parent, then either the `fileLocation` property (§3.17.2) or the `offset` property or both **SHALL** be present; they **SHALL NOT** both be absent. If the `offset` property is present, its value **SHALL** be that byte offset.

3.17.5 length property

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

3.17.6 roles property

A `file` object **MAY** have a property named `roles` whose value is an array of one or more distinct strings, each of which specifies a role that this file 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.
- `"attachment"`: The file is an attachment mentioned in `invocation.attachments` (§3.13.5) or `result.attachments` (§3.19.21).
- `"responseFile"`: The file contains command line arguments to a program, as specified in `invocation.responseFiles` (§3.13.4).
- `"resultFile"`: A result was detected in this file.
- `"standardStream"`: The file contains the contents of one of the standard input or output streams, as specified in `invocation.stdin`, `invocation.stdout`, `invocation.stderr`, or `invocation.stdoutStderr` (§3.13.22).
- `"traceFile"`: The analysis tool traced through this file while executing or simulating the execution of the code under test.

The following role values denote files that have changed since the baseline run. If `baselineInstanceGuid` (§3.11.5) is present on the containing `run` object (§3.11), 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 determines the baseline run.

- `"unmodifiedFile"`: The file has not been modified since the baseline run.
- `"modifiedFile"`: The file was modified after the baseline run.
- `"addedFile"`: The file was added after the baseline run.
- `"deletedFile"`: The file was deleted after the baseline run.
- `"renamedFile"`: The file was renamed after the baseline run. In this case, the `file` object specifies the new name.
- `"uncontrolledFile"`: The file is not under version control.

NOTE: 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.

3.17.7 mimeType property

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

3.17.8 contents property

A `file` object **MAY** contain a property named contents whose value is a `fileContent` object (§3.2) representing the entire contents of the file.

3.17.9 encoding property

If a `file` object represents a text file, it **MAY** contain a property named `encoding` whose value is a string that specifies the file's text encoding. The string **SHALL** be one of the character set names specified in [IANA-ENC]. The property value **SHALL** be case-insensitive.

If the `file` object represents a text file and this property is absent, it **SHALL** default to the value of the `defaultFileEncoding` property (§3.11.18) of the containing `run` object (§3.11), if that property is present; otherwise, the file's encoding **SHALL** be taken to be unknown.

If the `file` object represents a binary file, the `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 (§3.11)
  "defaultFileEncoding": "UTF-16BE",    # See §3.11.18.

  "files": {                            # See §3.11.13.
    "output.txt": {
      # encoding property omitted
    },

    "data.txt": {
      "encoding": "UTF-16LE"
    }
  }
}
```

3.17.10 hashes property

A `file` object **MAY** contain a property named `hashes` whose value is an array of unique (§3.6.2) hash objects (§3.18), each of which specifies a hashed value for the file specified by the `file` object, along with the name of the hash function used to compute the hash.

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

The array **SHOULD** contain an entry whose `algorithm` property is `"sha-256"`. 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 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 [IANA-HASH].

The array **MAY** contain entries whose `algorithm` property does not appear in [IANA-HASH], 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 [IANA-HASH].

> NOTE: A hash value for an analysis target can be useful when a log file is processed by a result management system. The value can be used as a key when persisting results in a database. This allows a build system to use cached results, rather than repeating the analysis, when a target has not changed. A file hash can also be useful for validating results in a policy compliance system, allowing an auditor to validate that rerunning analysis against a target that hashes to a specific value reproduces the provided results.
>
> The `file` object defines an array of hash values, rather than a single hash value, to allow a log file to be consumed by multiple tool chains that might expect hash values produced by differing 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.

3.17.11 lastModifiedTime property

A `file` object **MAY** contain a property named `lastModifiedTime` whose value is a string specifying the date and time at which the file was most recently modified. The string **SHALL** be in the format specified in §3.8.

> 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` (§3.17.10), as the most lightweight mechanism to determine whether the analysis needs to be repeated.

3.17.12 properties property

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

3.18 hash object

3.18.1 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:

```
{
    "value":"b13ce2678a8807ba0765ab94a0ecd394f869bc81",    # see §3.18.2
    "algorithm":"sha-256"                                  # see §3.18.3
}
```

3.18.2 value property

A `hash` object **SHALL** contain a property named `value` whose value is a string representation of the hash digest of some file or collection of files, computed by the hash function named in the `algorithm` property (§3.18.3). For hash functions that compute a numeric value, `value` **SHALL** 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.

3.18.3 algorithm property

A `hash` object **SHALL** contain a property named `algorithm` whose value is a string specifying the name of the hash function used to compute the hash `value` (§3.18.2). If the hash function is one whose name appears in the IANA registry of hash function textual names [IANA-HASH], `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 registry.

SARIF consumers **SHALL** treat `algorithm` as case-insensitive (even when comparing to hash function names in the IANA registry).

3.19 result object

3.19.1 General

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

3.19.2 Constraints

At least one of the `message` property (§3.19.8) and the `ruleMessageId` property (§3.19.9) **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` (§3.9.7) directly contains the message string `"Variable 'pBuffer' is uninitialized."`. `result.ruleId` (§3.19.6) and `result.ruleMessageId` (§3.19.9) together designate the rule message string `"Variable '{0}' is uninitialized."` which, along with the contents of `result.message.arguments` (§3.9.11), produces the identical string.

```
{                                                  # A run object (§3.11).
  "results": [                                     # See §3.11.16.
    {                                              # A result object (§3.19).
      "ruleId": "CA2101",                          # See §3.19.6.
      "message": {                                 # See §3.19.8.
        "text": "Variable 'pBuffer' is uninitialized.",  # See §3.9.7.
        "arguments": [ "pBuffer" ]                 # See §3.9.11.
      },
      "ruleMessageId": "default"                   # See §3.19.9.
    }
  ],
```

```
  "resources": {                                  # See §3.11.17.
    "rules": {                                    # See §3.35.3
      "CA2010": {                                 # A rule object (§3.36).
        "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 (§3.9.6) designated by `ruleId` and `ruleMessageId` might not produce the same string as `message.text`.

```
{                                                 # A run object (§3.11).
  "results": [                                    # See §3.11.16.
    {                                             # A result object (§3.19).
      "ruleId": "CA2101",                         # See §3.19.6.
      "message": {                                # See §3.19.8.
        "text": "Variable 'pBuffer' is uninitialized.",  # See §3.9.7.
        "arguments": [ "pBuffer" ]                # See §3.9.11.
      },
      "ruleMessageId": "default"                  # See §3.19.9.
    }
  ]
}
```

3.19.3 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 the 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 (§3.19.12). Result management systems that group results into equivalence classes based on an arbitrary unique identifier **SHOULD** populate the `correlationGuid` property (§3.19.5), regardless of whether they also compute a fingerprint.

3.19.4 instanceGuid property

A `result` object **MAY** contain a property named `instanceGuid` whose value is a GUID-valued string (§3.4.3) defining a unique, stable identifier for the result.

Direct SARIF producers and SARIF converters **SHOULD NOT** 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 §3.19.12 and Appendix B), but the `instanceGuid` properties for those results **SHALL** be distinct.

3.19.5 correlationGuid property

A `result` object **MAY** contain a property named `correlationGuid` whose value is a GUID-valued string (§3.4.3) 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` (§3.19.12) provide two different ways for result management systems to associate results that are logically identical. See §3.19.3 for more information.

3.19.6 ruleId property

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

> EXAMPLE 1:

```
"results": [
  {
     "ruleId": "CA2101"
      ...
  }
]
```

Direct producers **SHALL** emit `ruleId`.

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 synthesize it 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 (§3.11) contains a `resources.rules` property (§3.11.17, §3.35.3), 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 (§3.36.3) of the specified rule object (§3.36) **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 (§3.11).
  "results": [                # See §3.11.16.
    {                         # A result object (§3.19).
      "ruleId": "CA1711-1",   # Specifies a property name within "rules".
      ...
    }
  ],
  "resources": {              # See §3.11.17.
    "rules": {                # See §3.35.3.
```

```
      "CA1711-1": {          # A rule object (§3.36).
        "id": "CA1711",      # See §3.36.3.
        ...
      },
      "CA1711-2": {          # Another rule object with the same rule id.
        "id": "CA1711",
        ...
      }
    }
  }
}
```

3.19.7 level property

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

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

- `"pass"`: The rule specified by the `ruleId` property (§3.19.4) 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` property was evaluated, and the tool concluded that there was insufficient information to decide whether a problem exists.
- `"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:

```
"results": [
  {
    "ruleId": "ABC0001",
    "level": "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",
    "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",
    "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 (§3.37.3) of the `ruleConfiguration` object (§3.37) contained in the `configuration` property (§3.36.11) of the `rule` object (§3.36) specified by this `result` object's `ruleId` property (§3.19.4).

In that case, if the `run` object (§3.11) containing this result does not include a `resources.rules` property (§3.11.17, §3.35.3) (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"`.

3.19.8 message property

A `result` object **MAY** contain a property named `message` whose value is a `message` object (§3.9) that describes the result. If the `message` property is absent, the `ruleMessageId` property (§3.19.9) **SHALL** be present. Both `message` and `ruleMessageId` **MAY** be present. See §3.19.2 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: 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."
    }
  }
]
```

3.19.9 ruleMessageId 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` (§3.19.8) **SHALL** be present. Both `message` and `ruleMessageId` **MAY** be present. See §3.19.2 for more information.

If `resources.rules` (§3.11.17, §3.35.3) is present on the containing `run` object (§3.11), then `ruleMessageId` **SHALL** equal one of the property names in the `messageStrings` property (§3.36.7) of the `rule` object (§3.36) whose property name within `resources.rules` equals the `ruleId` property (§3.19.6) of this `result` object. `ruleMessageId` **MAY** also equal one of the property names in the `richMessageStrings` property (§3.36.8) of that `rule` object.

EXAMPLE 1: In this example, the `result` object's `ruleId` and `ruleMessageId` properties together specify the string identified by `"default"` within the rule metadata for the rule whose property name within `resources.rules` is `"CA2101"`.

```
{                                  # A run object (§3.11).
  "results": [
    {                              # A result object (§3.19).
      "ruleId": "CA2101",
      "ruleMessageId": "default",
      ...
    }
  ],

  "resources": {                   # A resources object (§3.35).
    "rules": {
      "CA2101": {
        "id": "CA2101",
        "messageStrings": {
          "default": "This is the default message for this rule.",
          "special": "This is another message for this rule, used in special cases."
        },
        "richMessageStrings": {
          "default": "This is _the_ default message for **this** rule.",
          "special": "This is another message for this rule, used in _special_ cases."
        }
      }
    }
  }
}
```

If the message string identified by `ruleId` and `ruleMessageId` includes placeholders (§3.9.4), then `result.message.arguments` (§3.19.8, §3.9.11) **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 (§3.11).
  "results": [
    {                             # A result object (§3.19).
      "ruleId": "CA2101",
      "ruleMessageId": "default",
      "message": {
        "arguments": [
          "counter"
        ]
      }
    }
  ],

  "resources": {
    "rules": {
      "CA2101": {                 # A rule object
        "messageStrings": {
          "default": "Variable \"{0}\" is uninitialized."
        }
      }
    }
  }
}
```

3.19.10 locations property

A `result` object **SHOULD** contain a property named `locations` whose value is an array of one or more unique (§3.6.2) `location` objects (§3.20), each of which specifies a location where the result occurred.

NOTE: In rare circumstances, it might not be possible to specify a location for a result. However, 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.

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

EXAMPLE 2: In C#, which support “partial” classes, portions of the declaration of a single class can occur at multiple locations in the source code. If an analysis tool reports that the name of such a class does not conform to a specified convention, then the resulting log file 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.

In contrast, consider a tool which locates misspelled words in variable names. If the tool detects a misspelled variable name, it must produce a single `result` object whose `locations` array contains the location of every reference to the variable, since fixing some but not all of the references would cause a compilation error.

3.19.11 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` object (§3.3) 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 the scan, the tool detected a result in the included file mouse.h.

```
{                                  # A result object (§3.19).
  "analysisTarget": {              # A fileLocation object (§3.3
    "uri": "input/mouse.c",
    "uriBaseId": "SRCROOT"
  },

  "locations": [                   # See §3.19.10.
    {                              # A location object (§3.20).
      "physicalLocation": {        # See §3.20.2.
        "fileLocation": {          # A fileLocation object.
          "uri": "input/mouse.h",
          "uriBaseId": "SRCROOT"
        },

        "region": {
          "startLine": 42
        }
      }
    }
  ]
}
```

3.19.12 fingerprints property

A `result` object **MAY** contain a property named `fingerprints` whose value is a JSON object (§3.5).

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 (§3.4.4.2). A result management system **MAY** use the property names to identify the method used to calculate the fingerprint.

EXAMPLE: In this example, the producer has calculated a fingerprint using version 2 of a fingerprinting method it refers to as `"contextRegionHash"`:

```
{
    "fingerprints": {
      "contextRegionHash/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 (§3.11).
  "results": [                     # See §3.11.16.
```

```
    {                                # A result object (§3.19).
      "fingerprints": {
        "contextRegionHash/v1": "1234567900abc"
        "contextRegionHash/v2": "234567900abcd"
    },
    {
      "fingerprints": {
        "contextRegionHash/v2": "234567900abcd"
        "contextRegionHash/v3": "34567900abcde"
    }
  ]
}
```

This property is an array, rather than a single string, to allow a result management system to select among a variety of methods for deciding whether two results are logically identical or logically distinct.

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` (§3.19.5) provide two different ways for result management systems to associate results that are logically identical. See §3.19.3 for more information.

3.19.13 partialFingerprints property

A `result` object **MAY** contain a property named `partialFingerprints` whose value is a JSON object (§3.5).

Each property value in this object **SHALL** be a string that contributes to the stable, unique identity, or "fingerprint," of the result (see §3.19.12). Appendix B explains how a result management system can compute these fingerprints.

Each property name in this object **SHALL** be a versioned hierarchical string (§3.4.4.2). 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 (§3.19).
  "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 (§3.11).
  "results": [                   # See §3.11.16.
    {                            # A result object (§3.19).
      "partialFingerprints": {
        "prohibitedWordHash/v1": "1234567900abc"
        "prohibitedWordHash/v2": "234567900abcd"
```

```
    },
    {
      "partialFingerprints": {
        "prohibitedWordHash/v2": "234567900abcd"
        "prohibitedWordHash/v3": "34567900abcde"
    }
  ]
}
```

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 the partial fingerprints in its fingerprint computation.

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: In this example, a SARIF-producing document checker has computed two partial fingerprints, one being a hash of a word that should not appear in a document, and the other being a hash of the document's language.

```
{                           # A result object
  ...
  "partialFingerprints": {
    "wordHash": " 2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae",
    "langHash": "5c49f88dafe66e0ecdca8f682ae0b38c38ccd3ad464e3358e899beca88c18560"
  }
}
```

EXAMPLE 2. 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"
  }
}
```

3.19.14 codeFlows property

A `result` object **MAY** contain a property named `codeFlows` whose value is an array of one or more unique (§3.6.2) `codeFlow` objects (§3.26). 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 code flow might be relevant to a single result.

3.19.15 graphs property

A `result` object **MAY** contain a property named `graphs` whose value is an array of one or more unique (§3.6.2) `graph` objects (§3.27) each of which 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 (§3.30) defined in the `graphTraversals` property (§3.19.16) of the `result` object in which it is defined. This contrasts with `graph` objects defined at the `run` level (§3.11.15), which **MAY** be referenced by `graphTraversal` objects defined in the `graphTraversals` property of any `result` object in the containing `run`.

3.19.16 graphTraversals property

If a `result` object contains a `graphs` property (§3.19.15), or if its containing `run` object (§3.11) contains a `graphs` property (§3.11.15), then the `result` object **MAY** contain a property named `graphTraversals` whose value is an array of one or more unique (§3.6.2) `graphTraversal` objects (§3.30). If neither the `result` object nor its containing `run` object 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.

3.19.17 stacks property

A `result` object **MAY** contain a property named `stacks` whose value is an array of one or more unique (§3.6.2) stack objects (§3.32). The `stacks` property is intended for use by analysis tools that collects call stack information in the process of producing results.

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

3.19.18 relatedLocations property

A `result` object **MAY** contain a property named `relatedLocations` whose value is an array of one or more unique (§3.6.2) `location` objects (§3.20), 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'."
    },
    "locations": [
      {
        "physicalLocation": {
          "uri": "file:///C:/Code/a.js",
          "region": {
            "startLine": "6",
            "startColumn": "10"
          }
        }
      }
    ],
    "relatedLocations": [   # An array of location objects
```

```
                            # (§3.20)
      {                     # A location object.
        "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.
```

3.19.19 suppressionStates property

3.19.19.1 General

A `result` object **MAY** contain a property named `suppressionStates` whose value is an array of unique (§3.6.2) strings. This property **SHALL** be present if and only if the analysis tool that produced the log file wishes to convey the information that the condition described by the result object should be "suppressed".

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

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

- The developer has suppressed the result in the source code (see §3.19.19.2).
- The result is marked as suppressed in an external store such as a database (see §3.19.19.3).

3.19.19.2 suppressedInSource value

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

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

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

3.19.19.3 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"`.

3.19.20 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` (§3.11.5) is present on the containing `run` object (§3.11), 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"`: This result was detected both in the current run and in the baseline run.
- `"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: 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` (§3.19.12) and `partialFingerprints` (§3.19.13) 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.

3.19.21 attachments property

A `result` object **MAY** contain a property named `attachments` whose value is an array of one or more unique (§3.6.2) `attachment` objects (§3.14). Each `attachment` object **SHALL** describe a file relevant to the detection of the result.

For an example, see EXAMPLE 2 in §3.14.1.

3.19.22 workItemUris property

A `result` object **MAY** contain a property named `workItemUris` whose value is an array of one or more unique (§3.6.2) strings, each containing the absolute URI [RFC3986] of a work item associated with this result.

> NOTE: 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.

3.19.23 conversionProvenance 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 (§3.11) 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 `result` object **MAY** contain a property named `conversionProvenance` whose value is an array of one or more unique (§3.6.2) `physicalLocation` objects (§3.21) which specify the relevant portions of those files.

Direct producers **SHALL NOT** emit the `conversionProvenance` property.

Per-run files are handled by the `conversion.analysisToolLogFiles` property (§3.15.4).

> 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 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 §3.14)
        ...
      },
      "results": [
        {
          "ruleId": "Assertions",
          "message": {
            "text": "Assertions are unreliable. ..."
          },
          ...
          "conversionProvenance": [     # An array of physicalLocation objects (§3.21).
            {
              "fileLocation": {                # See §3.21.3.
                "uri": "AndroidStudio.log",
                "uriBaseId": "$LOGSROOT"
              },
              "region": {                      # See §3.21.4.
                "startLine": 3,
                "startColumn": 3,
                "endLine": 12,
                "endColumn": 13
                "snippet": {
                  "text": "<problem>\n ... \n  </problem>",
                }
              }
            }
          ],
          ...
```

```
          }
        ]
      }
    ]
}
```

3.19.24 fixes property

A `result` object **MAY** contain a property names `fixes` whose value is an array of one or more unique (§3.6.2) `fix` objects (§3.37).

3.19.25 properties property

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

3.20 location object

3.20.1 General

A `location` object describes a location. Depending on the circumstances, a `location` object is described by physical location (§3.21), a logical location (§3.20.3), 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 reasons to include logical locations in the SARIF format in addition to physical locations:
>
> 1. In the absence of symbol information, binary analysis tools might not have source code locations available, so information about line and column numbers might not be present in the log file. In this case, code editors, other programs, or end users can use logical location to navigate from a result to the correct source code location.
>
> 2. Logical location information is an important contributor to fingerprinting scenarios, because it is typically more resilient to changes in source code than are line locations. See Appendix B for more information about fingerprinting. The `fullyQualifiedLogicalName` property (§3.20.3) is particularly convenient for fingerprinting.

In rare circumstances, their might be neither physical nor logical location information available for a `location` object. See §3.34.3 for an example. In that case, the location object **SHOULD** contain a message property (§3.20.4) explaining the significance of this “location.”

3.20.2 physicalLocation property

If physical location information is available, a `location` object **SHALL** contain a property named `physicalLocation` whose value is a `physicalLocation` object (§3.21) that identifies the file within which the location lies. If physical location information is not available, `physicalLocation` **SHALL** be absent.

3.20.3 fullyQualifiedLogicalName property

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

The format of `fullyQualifiedLogicalName` **SHALL** follow the naming rules for fully qualified logical locations described in §3.24.2.

EXAMPLE 1: C: `create_process`

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

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

If the `logicalLocations` property (§3.11.14) of the containing `run` object (§3.11) 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 (§3.11.14), 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 (§3.24.4) of the `logicalLocation` object (§3.24) **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 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 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 §3.20.1, `fullyQualifiedLogicalName` is also particularly convenient for fingerprinting, although the more detailed information in `run.logicalLocations` could be used instead.
- It relieves viewers from having to format the logical location from the more detailed information in `run.logicalLocations`.
- It is useful for producing readable in-source suppressions (for example, "suppress all instance of rule `CA2101` in the class `NamespaceA.NamespaceB.ClassC`").

3.20.4 message property

A `location` object **MAY** contain a property named `message` whose value is a `message` object (§3.9) relevant to the location.

3.20.5 annotations property

A `location` object **MAY** contain a property named `annotations` whose value is an array of one or more unique (§3.6.2) `region` objects (§3.22), each of which describes a region within the file specified by the `location` object that is relevant to the location. . Each of these `region` objects **SHOULD** contain a `message` property (§3.22.14) 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 (§3.22).
    "startLine": 12,
    "startColumn": 13,
    "endColumn": 19,
    "message": {
      "text": "(y + z) = 42"
    }
  }
]
```

3.20.6 properties property

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

3.21 physicalLocation object

3.21.1 General

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

3.21.2 id property

A `physicalLocation` object **MAY** contain a property named `id` whose value is a non-negative integer that **SHALL** be unique among all `physicalLocation` objects belonging to the containing `result` object (§3.19). The value does not need to be unique across all `result` objects in the run.

> EXAMPLE: Within a `result` 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 (§3.9.4) within a `message` object (§3.9) 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.

3.21.3 fileLocation property

A `physicalLocation` object **SHALL** contain a property named `fileLocation` whose value is a `fileLocation` object (§3.2) that represents the location of the file.

If `run.files` (§3.11.13) 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 (§3.11).
  "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
            }
          }
        }
      ]
    }
```

```
  ]
}
```

3.21.4 region property

A `physicalLocation` object **MAY** contain a property named `region` whose value is a `region` object (§3.22) that represents a relevant portion of the file. In particular, if the `physicalLocation` object occurs within the `locations` property (§3.19.10) of a `result` object (§3.19), the region property **SHALL** specify the region within the file 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 (§3.19).
  "locations": [                        # See §3.19.10.
    {                                   # A location object (§3.20).
      "physicalLocation": {             # See §3.20.2.
        "fileLocation": {               # A physicalLocation 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 file, then the `region` property **SHALL** specify the location with respect to the innermost nested file.

> 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 file.

3.21.5 contextRegion property

If a `physicalLocation` object contains a `region` property (§3.21.4), it **MAY** also contain a property named `contextRegion` whose value is a `region` object (§3.22) 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 of `contextRegion` is to enable a viewer to provide visual context when displaying a portion of a file.

> EXAMPLE In this example, an analysis tool detected a result on line 42. The tool provides additional context SARIF viewers by specifying a range of content surrounding the result line.

```
{                                       # A result object (§3.19).
  "locations": [                        # See §3.19.10.
    {                                   # A location object (§3.20).
      "physicalLocation": {             # See §3.20.2.
        "fileLocation": {               # A physicalLocation object.
          "uri": "ui/window.c",
          "uriBaseId": "SRCROOT"
        },

        "region": {                     # See §3.21.4.
```

```
          "startLine": 42,
          "snippet": {
            "text": "int n = m + 1;"
          }
        },

        "contextRegion": {
          "startLine": 41,
          "endLine": 43,
          "snippet": {
            "text": "int m;\nint n = m + 1\n\n"
          }
        }
      }
    }
  ]
}
```

3.22 region object

3.22.1 General

A `region` object represents a region, that is, a contiguous portion of a file.

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 both text properties and binary properties are present, they **SHALL** specify the identical range of bytes in the file, as determined by the file's character encoding.

For regions in binary files, a region object **SHALL** contain binary properties and **SHALL NOT** contain text properties.

If any text properties are present, enough text properties **SHALL** be present to fully specify a text region (see §3.22.2). If any binary properties are present, then enough binary properties **SHALL** be present to fully specify a binary region (see §3.22.3).

3.22.2 Text regions

NOTE 1: The examples in this section assume a text file with the following contents:

```
abcd\r\nefg\r\nhijk\r\nlmn
```

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 **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 **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.

Column numbers are expressed in the measurement unit specified by the `columnKind` property (§3.11.19) of the containing `run` object (§3.11).

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: 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 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 to have the encoding specified by `file.encoding` (§3.17.9), if present, or else by `run.defaultFileEncoding` (§3.11.18), 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: If a consumer incorrectly determines a file's encoding, it might not display the file 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 ways:

- By means of the "line/column" properties `startLine` (§3.22.5), `startColumn` (§3.22.6), `endLine` (§3.22.7), and `endColumn` (§3.22.8).
- By means of the "offset/length" properties `charOffset` (§3.22.9) and `charLength` (§3.22.10).
- 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.

The end of a text region **MAY** be specified by a combination of `endLine` and `endColumn`, or by `charLength`, or both.

A text region does not include the character specified by `endColumn`.

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 these 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 the `"g"` on line 2.

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.

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`.

```
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 }
{ "startLine": 1, "charLength": 14, "endLine": 3 }
{ "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 the `"b"` on line 1 through the `"d"` at the end of line 1. `endColumn` defaults to 5 (because there are 4 characters on the line, excluding the newline sequence). `charLength` defaults to 3.

It is identical, to these regions (among others):

```
{ "startLine": 1, "startColumn": 2, "endLine": 1 }
{ "startLine": 1, "startColumn": 2, "endLine": 1, "endColumn": 5 }
{ "startLine": 1, "startColumn": 2, "endLine": 1, "charLength": 3 }
{ "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 }
```

If a region spans more than one line, it **SHALL** include the newline sequences of all but the last line in the region.

EXAMPLE 7: The region


```
{ "startLine": 2, "endLine": 3 }
```

includes the characters `"efg\r\nhijk"`.

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: This is consistent with the rule that a region does not include the character in column `endColumn`.

EXAMPLE 8: These regions (among others) specify an insertion point before the `"b"` on line 1.

```
{ "startLine": 1, "startColumn": 2, "endColumn": 2 }
{ "startLine": 1, "startColumn": 2, "charLength": 0 }
```

EXAMPLE 9: 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": 0 }
{ "startLine": 1, "charLength": 0 }
```

To specify an insertion point after the last character in a file, set `endLine` to the number of the last line in the file, and set `endColumn` to a value one greater than the number of characters on the line, *including* any trailing newline sequence..

EXAMPLE 10: 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": 6, "charLength": 0 }
```

3.22.3 Binary regions

The byte offset of the first byte in a file **SHALL** be 0.

To specify a byte region, at least `byteOffset` (§3.22.11) **SHALL** be present. `byteLength` (§3.22.12) **MAY** also be present. `byteOffset` specifies the start of the region. `byteLength` specifies the end of the region. A `byteLength` value of 0 represents an insertion point before the byte specified by `byteOffset`.

3.22.4 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 in NOTE 1 of §3.22.2. 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": 6,    // 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.

3.22.5 startLine property

When a `region` object represents a text region, it **MAY** 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 §3.22.2.

3.22.6 startColumn property

When a `region` object represents a text region, 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 **SHALL** be inferred as specified in §3.22.2.

3.22.7 endLine property

When a `region` object represents a text region, 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 §3.22.2.

3.22.8 endColumn property

When a `region` object represents a text region, 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 value **SHALL** be inferred as specified in §3.22.2.

3.22.9 charOffset property

When a `region` object represents a text region, it **MAY** 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.

If `charOffset` is absent, it **SHALL** be inferred as specified in §3.22.2.

3.22.10 charLength property

When a `region` object represents a text region, 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 absent, or it **SHALL** have the value 0.

The sum of `charOffset` (§3.22.9) and `charLength` **SHALL** be greater than or equal to 0 and less than or equal to the number of characters in the file.

A region whose `charOffset` is equal to the number of characters in the file and whose `charLength` is 0 is permitted and **SHALL** represent an insertion point at the end of the file.

3.22.11 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.

3.22.12 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 defaults to 0.

The sum of `byteOffset` (§3.22.11) and `byteLength` **SHALL** be greater than or equal to 0 and less than or equal to the number of bytes in the file.

A `region` object whose `byteOffset` equals the number of bytes in the file and whose `byteLength` is 0 is permitted, and **SHALL** represent an insertion point at the end of the file.

3.22.13 snippet property

A `region` object **MAY** contain a property named `snippet` whose value is a `fileContent` object (§3.2) representing the portion of the file specified by the `region` object.

> NOTE: The purpose of the `snippet` property is to allow a SARIF viewer to present the contents of the region even if the file 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.

3.22.14 message property

A `region` object **MAY** contain a property named `message` whose value is a `message` object (§3.9) containing a message relevant to the region.

A SARIF viewer **SHOULD** 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.

3.23 rectangle object

3.23.1 General

A `rectangle` object specifies a rectangular area within an image. When a SARIF viewer displays an image, it **SHOULD** indicate the presence of these areas, for example, by highlighting them or surrounding them with a border.

3.23.2 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 [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`).

3.23.3 message property

A `rectangle` object **SHOULD** contain a property named `message` whose value is a `message` object (§3.9) containing a message relevant to this area of the image.

A SARIF viewer **SHOULD** 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.

3.24 logicalLocation object

3.24.1 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 as property values within the `run.logicalLocations` object (§3.11.14).

3.24.2 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` (§3.24.3): a logical name.
- `logicalLocation.fullyQualifiedName` (§3.24.4): a fully qualified logical name.
- `location.fullyQualifiedLogicalName` (§3.20.3): a fully qualified logical name, with one rare exception (see §3.20.3).
- The property names in the object specified by `run.logicalLocations` (§3.11.14): fully qualified logical names, with one rare exception (see §3.20.3).

Whenever possible, logical names and fully qualified logical names **SHALL** 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` (§3.24.6), 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"`.

3.24.3 name property

With one exception described below, a `logicalLocation` object **SHALL** 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 §3.24.2.

EXAMPLE 1: 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 (§3.24.6).

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 §3.11.14.
  "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"
    }
}
```

3.24.4 fullyQualifiedName property

A `logicalLocation` object either **SHALL** 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 §3.24.2.

If the fully qualified name does not equal the property name for this `logicalLocation` object in the `run.logicalLocations` object (§3.11.14), then `fullyQualifiedName` **SHALL** be present. This is an extremely rare corner case. See §3.20.3 for an explanation of the corner case and for an example. Otherwise, `fullyQualifiedName` **MAY** be present.

3.24.5 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` (§3.20.3) also be present.

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"
}
```

3.24.6 kind property

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

- `"function"`
- `"member"`
- `"module"`
- `"namespace"`
- `"package"`
- `"resource"`
- `"type"`
- `"returnType"`
- `"parameter"`
- `"variable"`

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

3.24.7 parentKey property

If the logical location represented by the `logicalLocation` object is a nested logical location, then the `logicalLocation` object **SHALL** contain a property named `parentKey` whose value is a string that

matches the property name of the parent `logicalLocation` object within `run.logicalLocations` (§3.11.14).

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

3.25 codeFlow object

3.25.1 General

A `codeFlow` object describes the progress of one more programs through one or more thread flows, which together result in the detection of a result 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 (§3.19).
  "codeFlows": [                        # See §3.19.14.
    {                                   # A codeFlow object (§3.25).
      "message": {                      # See §3.25.2.
        "text": "..."
      },

      "threadFlows": [                  # See §3.25.3.
        {                               # A threadFlow object (§3.26).
          "id": "thread-123",           # See §3.26.2.
          "message": {                  # See §3.26.3.
            "text": "..."
          },

          "locations": [                # See §3.26.4.
            {                           # A threadFlowLocation object (§3.34).
              "location": {             # See §3.34.3.
                "physicalLocation": {     # See §3.20.2.
                  "fileLocation": {
                    "uri": "ui/window.c",
                    "uriBaseId": "SRCROOT"
                  },

                  "region": {
                    "startLine": 42
                  }
                }
              },

              "state": {                # See §3.34.7.
                "x": "42",
                "y": "54",
                "x + y": "96"
              },

              "nestingLevel": 0,        # See §3.34.8.
              "executionOrder": 2       # See §3.34.9.
            }
          ]
        }
      ]
    }
  ]
}
```

3.25.2 message property

A `codeFlow` object **MAY** contain a property named `message` whose value is a `message` object (§3.9) relevant to the code flow.

3.25.3 threadFlows property

A `codeFlow` object **SHALL** contain a property named `threadFlows` whose value is an array of one or more unique (§3.6.2) `threadFlow` objects (§3.26), each of which describes the progress of a program through a single thread of execution such as an operating system thread or a fiber.

3.25.4 properties property

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

3.26 threadFlow object

3.26.1 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 §3.25.1.

3.26.2 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 (§3.25).

> 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.

3.26.3 message property

A `threadFlow` object **MAY** contain a property named `message` whose value is a `message` object (§3.9) relevant to the thread flow.

3.26.4 locations property

A `threadFlow` object **SHALL** contain a property named `locations` whose value is an array of one or more `codeFlowCodeLocation` objects (§3.34). Each element of the array **SHALL** represent a single location visited by the tool in the course of producing the result. This array does not need to include every location visited by the tool, but the elements that are present **SHALL** occur in the order that the tool visited them. The elements do need to be unique within the array.

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

3.26.5 properties property

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

3.27 graph object

3.27.1 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` (§3.11.15) and at the result level in `result.graphs` (§3.19.15).

A path through a graph, called a "graph traversal," is represented by a `graphTraversal` object (§3.30).

3.27.2 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 (§3.11.15) or `result.graphs` property (§3.19.15). The `id` property does not have to be unique across all `graph` objects in all `result.graphs` properties in the `run`.

3.27.3 description property

A `graph` object **MAY** contain a property named `description` whose value is a `message` object (§3.9) that describes the graph.

3.27.4 nodes property

A `graph` object **SHALL** contain a property named `nodes` whose value is an array of unique (§3.6.2) `node` objects (§3.28) which represent the nodes of the graph.

3.27.5 edges property

A `graph` object **SHALL** contain a property named `edges` whose value is an array of unique (§3.6.2) `edge` objects (§3.29) which represent the edges of the graph.

3.27.6 properties property

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

3.28 node object

3.28.1 General

A `node` object represents a node in the graph represented by the containing `graph` object (§3.27).

3.28.2 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 (§3.27). `id` **SHALL** be unique among all nodes in the graph, regardless of nesting (see §3.28.5).

EXAMPLE: This graph is invalid because two nodes have the same `id`, even though the nodes are within unrelated nested graphs.

```
{                               # A graph object (§3.27).
  "nodes": [                    # See §3.27.4.
    {                           # A node object.
      "id": "n1",
      "children": [             # See §3.28.5.
        {
```

```
          "id": "n3"
        }
      ]
    },
    {
      "id": "n2",
      "children": [
        {
          "id": "n3"          # INVALID: duplicate id.
        }
      ]
    }
  ],
  ...
}
```

3.28.3 label property

A `node` object **MAY** contain a property named `label` whose value is a `message` object (§3.9) that provides a short description of the node.

3.28.4 location property

A `node` object **SHOULD** have a property named `location` whose value is a `location` object (§3.20) that specifies the location associated with the node.

3.28.5 children property

A `node` object **MAY** contain a property named `children` whose value is an array of unique (§3.6.2) `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.

3.28.6 properties property

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

3.29 edge object

3.29.1 General

An `edge` object represents a directed edge in the graph represented by the containing `graph` object (§3.27).

3.29.2 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 (§3.27).

3.29.3 label property

An `edge` object **MAY** contain a property named `label` whose value is a `message` object (§3.9) that provides a short description of the edge.

3.29.4 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 (§3.28.2) of one of the `node` objects (§3.28) in the containing `graph` object (§3.27). It **MAY** equal the id of any node within the graph, regardless of nesting (see §3.28.5).

EXAMPLE: In this example, an edge connects two nodes defined in unrelated nested graphs.

```
{                                 # A graph object (§3.27).
  "nodes": [                      # See §3.27.4.
    {                             # A node object.
      "id": "n1",
      "children": [               # See §3.28.5.
        {
          "id": "n3"
        }
      ]
    },
    {
      "id": "n2",
      "children": [
        {
          "id": "n4"
        }
      ]
    }
  ],
  "edges": [                      # See §3.27.5.
    {
      "sourceNodeId": "n3",       # Source node and target node are in separate
      "targetNodeId": "n4"        # nested graphs: ok.
    }
  ],
  ...
}
```

3.29.5 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 (§3.28.2) of one of the `node` objects (§3.28) in the containing `graph` object (§3.27). It **MAY** equal `sourceNodeId` (§3.29.4).

3.29.6 properties property

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

3.30 graphTraversal object

3.30.1 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 (§3.31). For an example, see §3.30.5.

3.30.2 graphId property

A `graphTraversal` object **SHALL** contain a property named `graphId` whose value is a string that equals the `id` property (§3.27.2) of the `graph` object (§3.27) being traversed.

The value of `graphId` **SHALL** equal the `id` of a `graph` object that occurs in the `graphs` property (§3.19.15) of the containing `result` object (§3.19), or the `id` of a `graph` object that occurs in the `graphs` property (§3.11.15) of the containing `run` object (§3.11), or both (in which case the `graph` object in `result.graphs` takes precedence).

3.30.3 description property

A `graphTraversal` object **MAY** contain a property named `description` whose value is a `message` object (§3.9) that describes the graph traversal.

3.30.4 initialState property

A `graphTraversal` object **MAY** contain a property named `initialState` whose value is a JSON object (§3.5) 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` (§3.31.4), 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 §3.34.7.

3.30.5 edgeTraversals property

A `graphTraversal` object **SHALL** contain a property named `edgeTraversals` whose value is an array of `edgeTraversal` objects (§3.31) 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` (§3.30.4) and `edgeTraversal.finalState` (§3.31.4).

```
{                                          # A result object (§3.19).
  "graphs": [                              # See §3.19.15.
    {                                      # A graph object (§3.27).
      "id": "g1",                          # See §3.27.2.

      "nodes": [                           # See §3.27.4.
        { "id": "n1" },                    # A node object (§3.28).
        { "id": "n2" },
        { "id": "n3" },
        { "id": "n4" }
      ],

      "edges": [                           # See §3.27.5.
        {                                  # An edge object (§3.29).
          "id": "e1",                      # See §3.29.2.
          "sourceNodeId": "n1",            # See §3.29.4.
          "targetNodeId": "n2"             # See §3.29.5.
        },
        {
          "id": "e2",
          "sourceNodeId": "n2",
          "targetNodeId": "n3"
```

```
        },
        {
          "id": "e3",
          "sourceNodeId": "n2",
          "targetNodeId": "n4"
        }
      ]
    }
  ],

  "graphTraversals": [                      # See §3.19.16.
    {                                       # A graphTraversal object (§3.30).
      "graphId": "g1",                      # See §3.30.2.

      "initialState": {                     # See §3.30.4.
        "x": "1",
        "y": "2",
        "x + y": "3"
      },

      "edgeTraversals": [                   # See §3.30.5.
        {                                   # An edgeTraversal object (§3.31).
          "edgeId": "e1",                   # See §3.31.2.

          "finalState": {                   # See §3.31.4.
            "x": "4",
            "y": "2",
            "x + y": "6"
          }
        },
        {
          "edgeId": "e3",

          "finalState": {
            "x": "4",
            "y": "7",
            "x + y": "11"
          }
        }
      ]
    }
  ]
}
```

3.30.6 properties property

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

3.31 edgeTraversal object

3.31.1 General

An `edgeTraversal` object represents the traversal of a single edge during a graph traversal.

3.31.2 edgeId property

An `edgeTraversal` object **SHALL** contain a property named `edgeId` whose value is a string which equals the `id` property (§3.29.2) of one of the `edge` objects (§3.29) in the graph identified by the `graphId` property (§3.30.2) of the containing `graphTraversal` object (§3.30).

3.31.3 message property

An `edgeTraversal` object **MAY** contain a property named `message` whose value is a `message` object (§3.9) that contains a message to display to the user as the edge is traversed.

3.31.4 finalState property

An `edgeTraversal` object **MAY** contain a property named `finalState` whose value is a JSON object (§3.5) each of whose properties represents the value of a relevant expression after the edge has been traversed. This property, together with `graphTraversal.initialState` (§3.30.4), 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 §3.34.7.

3.31.5 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 (§3.28.5). 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 **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 (§3.19).
  "graphs": [                                        # See §3.19.15.
    {                                                # A graph object (§3.27).
      "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 §3.19.16.
    {                                           # A graphTraversal object (§3.30).
      "graphId": "code",                        # The graph being traversed.
      "edgeTraversals": [
        { "edgeId": "ea1" },
        {
          "edgeId": "eab",
          "stepOverEdgeCount": 4
        },
        { "edgeId": "eb1" },
        { "edgeId": "eb2" },
        { "edgeId": "eba" },
        { "edgeId": "ea3" }
      ]
    }
  ]
}
```

3.31.6 properties property

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

3.32 stack object

3.32.1 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.

3.32.2 message property

A `stack` object **MAY** contain a property named `message` whose value is `message` object (§3.9) relevant to this call stack.

3.32.3 frames property

A stack object **SHALL** contain a property named `frames` whose value is an array of one or more `stackFrame` objects (§3.33). 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.

3.32.4 properties property

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

3.33 stackFrame object

3.33.1 General

A `stackFrame` object describes a single stack frame within a call stack (§3.32).

3.33.2 location property

A `stackFrame` object **MAY** contain a property named `location` whose value is a `location` object (§3.20) specifying the location to which this stack frame refers.

3.33.3 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.

3.33.4 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.

3.33.5 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.

3.33.6 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 (§3.22.11), if any, specified by the `physicalLocation` property (§3.33.2). `physicalLocation.region.byteOffset` specifies an offset from the start of a file, not from the start of a method.

3.33.7 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.

3.33.8 properties property

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

3.34 threadFlowLocation object

3.34.1 General

A `threadFlowLocation` object represents a location visited by an analysis tool in the course of simulating or monitoring the execution of a program.

3.34.2 step property

A `threadFlowLocation` object **MAY** contain a property named `step` whose value is an integer specifying the 1-based sequence number of the location within the thread flow: 1 for the first location, 2 for the second, and so on.

NOTE: This property has two primary purposes:

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

3.34.3 location property

If location information is available, a `threadFlowLocation` object **SHALL** contain a property named `location` whose value is a `location` object (§3.20) 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 (§3.20.4). 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` (§3.34.9) to ensure that the location in the external program executes before the location in the program being analyzed.

```
{                                          # A codeFlow object (§3.25).
  "threadFlows": [                         # See §3.25.3.
    {                                      # A threadFlow object (§3.26).
      "message": {                         # See §3.26.3.
        "text": "An external program."
      },
      "locations": [                       # See §3.26.4.
        {                                  # 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,
          "message": {
            "text": "Attempt to write to the file."
          },
          "location": {
            "physicalLocation": {
              "fileLocation": {
                "uri": "io/logger.c",
                "uriBaseId": "SRCROOT"
              },
              "region": {
                "startLine": 42,
                "snippet": {
                  "text": "    fprintf(fd, "test\\n");\n"
                }
              }
            }
          }
        }
      ]
    }
  ]
}
```

3.34.4 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.

3.34.5 stack property

A `threadFlowLocation` object **MAY** contain a property named `stack` whose value is a `stack` object (§3.32) that represents the call stack leading to this location.

3.34.6 kind property

A `threadFlowLocation` object **MAY** contain a property named `kind` whose value is a string that describes the meaning of this location. The interpretation of `kind` depends on the tool that produced the log file. A SARIF consumer that wishes to take action based on `kind` **SHALL** examine `run.tool` (§3.11.8, §3.12) to determine if it (the consumer) knows how to interpret the `kind` values produced by that tool.

`kind` **SHOULD** be a human-readable string (as opposed to, for example, a GUID or a hash value).

A SARIF producer **MAY** provide additional `kind`-dependent information by populating `threadFlowLocation.properties` with properties whose names and values depend on `kind`. A SARIF consumer that knows how to interpret `kind` for this tool **MAY** use this additional information.

EXAMPLE:

```
"kind": "taintedDataSource"
```

3.34.7 state property

A `threadFlowLocation` object **MAY** contain a property named `state` whose value is a JSON object (§3.5) each of whose properties represents the value of an expression relevant to the location in the context of the code flow. This property enables a SARIF viewer to present a debugger-like "watch window" experience as the user navigates through a code flow.

EXAMPLE 1: In this example, the `state` property captures the values of the expressions `"x"`, `"y"`, and `"x + y"`.

```
{                                 # An threadFlowLocation object.
  "state": {
    "x": "42",
    "y": "54",
    "x + y": "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

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).
- A Boolean such as `"true"`.

3.34.8 nestingLevel property

A `threadFlowLocation` object **MAY** contain a property named `nestingLevel` whose value is an 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.

3.34.9 executionOrder property

A `threadFlowLocation` object **MAY** contain a property named `executionOrder` whose value is a positive 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` (§3.25). `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 is absent, it **SHALL** default to 0, which is not otherwise a valid value for `executionOrder`.

3.34.10 timestamp property

A `threadFlowLocation` object **MAY** contain a property named `timestamp` whose value is a string specifying the date and time at which the code at this location was executed. The string **SHALL** be in the format specified in §3.8.

3.34.11 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.

3.34.12 properties property

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

3.35 resources object

3.35.1 General

A `resources` object represents items that can be localized, such as message strings and rule metadata.

3.35.2 messageStrings property

A `resources` object **MAY** contain a property named `messageStrings` whose value is a JSON object (§3.5) each of whose properties represents a single localized string. The property names correspond to resource identifiers (§3.9.6) within `message` objects (§3.9). If the property name is used as the value of the `messageId` property (§3.9.9) of any `message` object in the containing `run` object (§3.11), the

property value **SHALL** be a plain text string (§3.9.2). If the property name is used as the value of the `richMessageId` property (§3.9.10) of any `message` object in the containing `run` object, the property value **SHALL** be a rich text string (§3.9.3). 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` object.

EXAMPLE:

```
"resources": {
  "messageStrings": {
    "call": "Function call",
    "return": "Function return"
  }
}
```

3.35.3 rules property

A `resources` object **MAY** contain a property named `rules` whose value is a JSON object (§3.5), each of whose properties represents a `rule` object (§3.36).

If there is only one `rule` object with a particular `id` (§3.36.3), then the property name for that `rule` object **SHALL** be the rule id.

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

```
"resources": {
  "rules": {
    "CA1001": {
      "id": "CA1001",
      "shortDescription": {
        "text": "Types that own disposable fields should be
                 disposable."
      }
    },
    "CA1002": {
      "id": "CA1002",
      "shortDescription": {
        "text": "Do not expose generic lists."
      }
    }
  }
}
```

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

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

```
"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 (§3.19) by means of the result's `ruleId` property (§3.19.6).

3.36 rule object

3.36.1 General

A `rule` object contains information that describes a rule. We refer to this information as “rule metadata.”

3.36.2 Constraints

Either the `shortDescription` property (§3.36.5) or the `fullDescription` property (§3.36.6) or both **SHALL** be present.

3.36.3 id property

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

EXAMPLE: `"CA2101"`

NOTE: Rule identifiers must be stable for two reasons:

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

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

3.36.4 name property

A `rule` object **MAY** contain a property named `name` whose value is a `message` object (§3.9) containing a rule identifier that is understandable to an end user. If `name` contains implementation details that change over time, a tool author might alter a rule's name (while leaving the stable `id` property unchanged).

NOTE 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"
  }
}
```

3.36.5 shortDescription property

A `rule` object **MAY** contain a property named `shortDescription` whose value is a `message` object (§3.9) 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.

EXAMPLE:

```
{                          # A rule object
  "shortDescription": {
    "text": "Specify marshaling for P/Invoke string arguments"
  }
}
```

3.36.6 fullDescription property

A `rule` object **SHOULD** contain a property named `fullDescription` whose value is a `message` object (§3.9) that describes the rule.

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

The first sentence of `fullDescription` **SHOULD** provide a concise description of the rule, suitable for display in cases where available space is limited. Tools that construct `fullDescription` in this way do not need to provide a value for `shortDescription` (§3.36.5). Tools that do not construct `fullDescription` in this way **SHOULD** provide a value for `shortDescription`, because otherwise, the initial portion of `fullDescription` that a viewer displays where available space is limited might not be understandable.

3.36.7 messageStrings property

A `rule` object **MAY** contain a property named `messageStrings` whose value is a JSON object (§3.5) consisting of a set of properties with arbitrary names.

The value of each property **SHALL** be a plain text message string (§3.9.2). As with any message string, it **MAY** contain placeholders (§3.9.4) and embedded links (§3.9.5).

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` properties (§3.19.9) in the run. The `messageStrings` property **MAY** contain additional properties whose names do not appear as the value of the `result.ruleMessageId` property for any result 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 rule, rather than restricting it to those messages that happen to appear in the log file.

EXAMPLE:

```
{                        # A rule object
  "messageStrings": {
    "objectCreation":  "{0} creates a new instance of {1} which is never used.
                        Pass the instance as an argument to another method,
                        assign the instance to a variable,
                        or remove the object creation if it is unnecessary.",
    "stringReturnValue": "{0} calls {1} but does not use the new string
                          instance that the method returns.
                          Pass the instance as an argument to another method,
                          assign the instance to a variable,
                          or remove the call if it is unnecessary."
  }
}
```

3.36.8 richMessageStrings property

If a `rule` object contains a `messageStrings` property (§3.36.7), it **MAY** also contain a property named `richMessageStrings` whose value is a JSON object (§3.5) consisting of a set of properties with arbitrary names.

The value of each property **SHALL** be a rich text message string (§3.9.3). As with any message string, it **MAY** contain placeholders (§3.9.4) and embedded links (§3.9.5).

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 §3.9.3.2.

3.36.9 helpUri property

A `rule` object **MAY** contain a property named `helpUri` whose value is a string containing the absolute URI [RFC3986] of the primary documentation for the rule.

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

3.36.10 help property

A `rule` object **MAY** contain a property named `help` whose value is a `message` object (§3.9) which provides the primary documentation for the rule.

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

3.36.11 configuration property

A `rule` object **MAY** contain a property named `configuration` whose value is a `ruleConfiguration` object (§3.37).

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 §3.37.

3.36.12 properties property

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

This property **SHALL NOT** be used to hold rule configuration information. Use the `ruleConfiguration.parameters` property (§3.37.4) for that.

3.37 ruleConfiguration object

3.37.1 General

A `ruleConfiguration` object contains rule configuration information, that is, information about the rule that a SARIF producer can modify at runtime, before executing its scan. For example, if the rule specifies a maximum source file line length, its configuration information might specify the maximum permitted line length.

For an example, see §3.37.4.

3.37.2 enabled property

A `ruleConfiguration` object **MAY** contain a property named `enabled` whose value is a Boolean that specifies whether the rule will be evaluated 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:

```
SecurityScanner --disable "SEC4002,SEC4003" --enable SEC6012
```

3.37.3 defaultLevel property

A `ruleConfiguration` object **MAY** contain a property named `defaultLevel` whose value is one of the strings `"warning"`, `"error"`, `"note"`, or `"open"`, with the same meanings as when those strings appear as the value of `result.level` (§3.19.7).

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

The value of this property **SHALL** provide the value for the `level` property of any `result` object (§3.19) whose `ruleId` property (§3.19.4) refers to this rule configuration and which does not itself specify a `level` property.

EXAMPLE: In this example, a tool allows the user to override a rule's default level:

```
WebScanner --level "WEB1002:error,WEB1005:warning"
```

3.37.4 parameters property

A `ruleConfiguration` object **MAY** contain a property named `parameters` whose value is a property bag (§3.7). This allows a rule to define configuration information that is specific to that rule.

EXAMPLE: In this example, a rule that specifies the maximum permitted source line length is parameterized by the maximum length.

```
{                                   # A rule object (§3.36.)
  "id": "SA2707",
  "name": {
    "text": "LimitSourceLineLength"
  },
  "shortDescription": {
    "text": "Limit source line length for readability."
  },
  "configuration": {
    "enabled": true,
    "defaultLevel": "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"
```

3.38 fix object

3.38.1 General

A `fix` object represents a proposed fix for the problem indicated by the containing `result` object (§3.19). It specifies a set of files to modify. For each file, it specifies regions to remove, and provides new file content to insert.

EXAMPLE:

```
{                                   # A result object (§3.19).
  "fix": {
    "description": {                # See §3.38.2.
      "text": "Private member names begin with '_'"
    },
    "fileChanges": [                # See §3.38.3.
      {                             # A fileChange object (§3.39).
        ...
      }
    ]
  }
}
```

3.38.2 description property

A `fix` object **SHOULD** contain a property named `description` whose value is a `message` object (§3.9) 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'."
  },
  ...
}
```

3.38.3 fileChanges property

A `fix` object **SHALL** contain a property named `fileChanges` whose value is an array of one or more `fileChange` objects (§3.39).

3.39 fileChange object

3.39.1 General

A `fileChange` object represents a change to a single file.

EXAMPLE:

```
{                             # A fix object (§3.37).
  "fileChanges": [            # See §3.38.3.
    {
      "fileLocation": {       # See §3.39.2.
        "uri": "a.h"
      },
      "replacements": [       # See §3.39.3.
        {                     # A replacement object (§3.40).
          ...
        },
        {                     # Another replacement object.
          ...
        }
      ]
    }
  ]
}
```

3.39.2 fileLocation property

A `fileChange` object **SHALL** contain a property named `fileLocation` whose value is a `fileLocation` object (§3.3) that represents the location of the file.

3.39.3 replacements property

A `fileChange` object **SHALL** contain a property named `replacements` whose value is an array of one or more `replacement` objects (§3.40), each of which represents the replacement of a single region of the file specified by the `fileLocation` property (§3.39.2).

3.40 replacement object

3.40.1 General

A `replacement` object represents the replacement of a single region of a file. 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 (§3.40.3) and the insertion of new file content by means of the `insertedContent` property (§3.40.4), then the effect of the replacement **SHALL** be as if the removal were performed before the insertion.

If a single `fileChange` object (§3.39) 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 (§3.39.3). The `deletedRegion` property of each `replacement` object **SHALL** specify the location of the replacement in the unmodified file.

EXAMPLE 1: Suppose a `fileChange` object contains a `replacements` property whose value is the following array of `replacement` objects:

```
"fileChanges": [
  {
    "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. 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.

A `replacement` object can represent either a textual replacement or a binary replacement, depending on whether the `deletedRegion` property (§3.40.3) specifies a text region (§3.22.2) or a binary region (§3.22.3).

EXAMPLE 2: In this example, the `replacements` property specifies a replacement in a text file.

```
"replacements": [
  {
    "deletedRegion": {         # The region object represents a text region (§3.22.2).
      "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, 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 starts with a byte order mark (BOM).

3.40.2 Constraints

If the `deletedRegion` property (§3.40.3) specifies a text region (§3.22.2) and the `insertedContent` property (§3.40.4) is present, then the `insertedContent` property **SHOULD** contain a `text` property (§3.2.2).

If the `deletedRegion` property specifies a binary region (§3.22.3) and the `insertedContent` property is present, then the `insertedContent` property **SHALL** contain a `binary` property (§3.2.3).

3.40.3 deletedRegion property

A `replacement` object **SHALL** contain a property named `deletedRegion` whose value is a `region` object (§3.22) 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.

3.40.4 insertedContent property

A `replacement` object **MAY** contain a property named `insertedContent` whose value is a `fileContent` object (§3.2) 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 `insertedContent` is absent or its properties specify content whose length is zero, the SARIF consumer performing the replacement **SHALL NOT** insert any content.

3.41 notification object

3.41.1 General

A `notification` object describes a condition encountered in the course of running an analysis tool which is relevant to the operation of the tool itself, as opposed to being relevant to a file being analyzed by the tool. Conditions relevant to files being analyzed by a tool are represented by `result` objects (§3.19).

3.41.2 id property

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

> NOTE: In contrast to rule identifiers (see `rule.id`, §3.36.3), which must be stable and opaque, notification identifiers do need to be either stable or opaque, because the reasoning that leads to those requirements for rule ids does not apply to tool notifications. A tool notification with level `"error"` should always be treated as a failure, and tools should not allow them to be disabled. And tool authors are free to change the notification ids at any time, so there is no reason for them to be opaque; to the contrary, they are more useful if they convey information to the user.

3.41.3 ruleId property

If the condition described by the `notification` object is relevant to a particular analysis rule, the `notification` object **SHOULD** contain a property named `ruleId` whose value is a string containing the stable, unique identifier of the rule (§3.36.3).

If there is more than one rule with the desired id, and if the containing `run` object (§3.11) contains a `resources.rules` property (§3.11.17, §3.35.3), 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 (§3.36.3) of the specified rule object (§3.36) **SHALL** contains the actual rule id.

> 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"].id`.

```
{                                       # A run object (§3.11).
  "invocations": [                      # See §3.11.9.
    {                                   # An invocation object (§3.13).
      "configurationNotifications": [   # See §3.13.21.
        {                               # A notification object.
          "id": "CFG0001",
          "message": {
            "text": "Rule configuration is missing."
          },
          "ruleId": "CA1711-1"          # Specifies a property name within "rules".
        }
      ],
    }
  ],

  "resources": {                        # See §3.11.17.
    "rules": {                          # See §3.35.3.
      "CA1711-1": {                     # A rule object (§3.36).
        "id": "CA1711",
        ...
      },
      "CA1711-2": {                     # Another rule object with the same id.
        "id": "CA1711",
        ...
```

```
        }
      }
    }
  }
```

3.41.4 physicalLocation property

If the condition described by the `notification` object is relevant to a particular file location, the `notification` object **SHOULD** contain a property named `physicalLocation` whose value is a `physicalLocation` object (§3.21) that identifies the relevant location.

3.41.5 message property

A `notification` object **SHALL** contain a property named `message` whose value is a `message` object (§3.9) that describes the condition that was encountered.

> NOTE: The `message` object in the `notification.message` property will typically not contain a `richText` (§3.9.8) or `richMessageId` (§3.9.10) property because tool notifications typically appear on the console, where rich text is not supported.

3.41.6 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.

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

3.41.7 threadId property

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

3.41.8 time property

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

3.41.9 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 (§3.42).

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

3.41.10 properties property

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

3.42 exception object

3.42.1 General

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

3.42.2 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.

3.42.3 message property

An `exception` object **SHOULD** contain a property named `message` whose value is a string containing a plain text message string (§3.9.2) 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 populate `message` from the string returned from the `what()` method of any object derived from `std::exception`.

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

> NOTE: The `exception.message` property is not a message object (§3.9) because exception messages, appearing as they do in typical languages and operating systems, are inherently plain text, and require no arguments (§3.9.4).

3.42.4 stack property

An `exception` object **MAY** contain a property named `stack` whose value is a `stack` object (§3.32) that describes the sequence of function calls leading to the exception.

3.42.5 innerExceptions property

An `exception` object **MAY** contain a property named `innerExceptions` whose value is an array of one or more `exception` objects, each of which is considered 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 Conformance

4.1 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**
- **SARIF resource file**: A SARIF file that contains only those elements related to resources.
- **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 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 Conformance Clause 2: SARIF resource file

A text file satisfies the "SARIF resource file" conformance profile if:

- Its name conforms to the convention defined in §3.9.6.4, "SARIF resource file format".
- It contains only those elements defined in §3.9.6.4.
- Those elements that it does contain conform to the syntax and semantics defined in §3, except as modified in §3.9.6.4.

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.

4.5 Conformance Clause 4: 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.

4.6 Conformance Clause 5: Deterministic producer

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.

4.8 Conformance Clause 7: SARIF post-processor

A SARIF post-processor satisfies the “SARIF post-processor” conformance profile if:

- It satisfies the “SARIF producer” conformance profile.
- It additionally satisfies those normative requirements in §3 that are designated as applying to post-processors.

4.9 Conformance Clause 8: 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.

4.10 Conformance Clause 9: 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.

4.11 Conformance Clause 10: 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 Conformance Clause 11: 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 Wysopal, CA Technologies
David Keaton, Individual
Douglas Smith, Kestrel Technology
Duncan Sparrell, sFractal Consulting LLC
Everett Maus, Microsoft
Hendrik Buchwald, RIPS Technologies
Henny Sipma, Kestrel Technology
Jim Kupsch, SWAMP
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
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
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 **SHALL** include the information in `partialFingerprints` in its fingerprint computation. See §3.19.13 for more requirements on how a result management system decides which partial fingerprints to use.

An analysis tool **SHALL 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 file) in its fingerprint computation.

A result management system **SHALL NOT** include non-deterministic file format elements (Appendix F, §F.2) in its fingerprint computation.

A result management system **SHALL NOT** include non-deterministic absolute URIs (Appendix F, §F.4) in its fingerprint computation.

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

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` property of the `file` object provides this information. In the absence of the `mimeType` property, a viewer can fall back to examining the filename extension, for example ".zip". It is recommended that the analysis tool provide the `mimeType` property (which it must know, because it was able to interpret the file in which it detected the result), rather than forcing the viewer to rely on a file name extension.

Appendix D. (Informative) Production of SARIF by converters

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

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

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

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

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

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

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

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

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

- A converter should not attempt to synthesize a `ruleId` for a result if the tool does not provide one.
- A converter that knows which file a result was detected in, but not which file the analysis tool was originally instructed to scan, should populate the `location.physicalLocation` property, but should not attempt to populate `result.analysisTarget` (see §3.19.11).
- A converter should not attempt to guess whether the analysis tool's version string is intended to be interpreted as a Semantic Version 2.0.0 version string (see §3.12.4).

Appendix E. (Informative) Locating rule metadata

The SARIF format allows rule metadata to be included in a SARIF log file (see §3.11.17 and §3.35). 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 metadata should be included in a log file in the following circumstances:

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

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

Appendix F. (Normative) 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 identical inputs.

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

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

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

There are several issues to consider when producing deterministic output:

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

F.2 Non-deterministic file format elements

A tool that produces deterministic output **SHALL NOT** emit the following elements of the SARIF format. All of these elements are **OPTIONAL**.

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

- `invocation.startTime`
- `invocation.endTime`
- `invocation.processId`
- `invocation.machine`
- `invocation.account`
- `invocation.fileName` (because `fileName` is specified as being an absolute path, and tools might be stored in different directories on different machines)
- `invocation.workingDirectory`
- `invocation.environmentVariables`
- The use of absolute file paths in `invocation.commandLine` (because builds performed on different machines might use a different root directory)
- `threadFlow.threadId`
- `notification.threadId`
- `notification.time`
- `result.instanceGuid`
- `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

F.3 Array and dictionary element ordering

A tool that produces deterministic output **SHALL** emit array and dictionary elements in a deterministic order.

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

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

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

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

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

F.4 Absolute paths

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 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.uri` property with the corresponding `fileLocation.uriBaseId` property.

F.5 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.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 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.

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 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 `region` and `contextRegion` properties to store enough of the generated file’s contents to facilitate diagnosis. The `region` object’s `snippet` property holds the relevant portion of the file contents.
2. Use the `file` object’s `contents` property to persist the entire contents of the file in `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 (§3.11).
  "originalUriBaseIds": {                  # See §3.11.12
    "GENERATED": "file:///C:/code/browser/obj"
  },

  "results": [                             # See §3.11.16.
    {                                      # A result object (§3.19).
      "ruleId": "CS6789",                  # See §3.19.6.
      "message": {                         # See §3.19.8.
        "text": "Division by 0!"
      },
      "locations": [                       # See §3.19.10.
        {                                  # A location object (§3.20).
          "physicalLocation": {            # See §3.20.2.
            "fileLocation": {
              "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.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"
  },

  "results": [
    {
      "ruleId": "CS6789",
      "message": {
        "text": "Division by 0!"
      },
      "locations": [
        {
          "physicalLocation": {
            "fileLocation": {
              "uri": "ui/window.g.cs",
              "uriBaseId": "GENERATED"
            },
            "region": {
              "startLine": 42
            },
            "contextRegion": {
              "startLine": 40,
              "endLine": 42
            }
          }
        }
      ]
    }
  ],

  "files": {                                  # See §3.11.13.
    "ui/window.g.cs":                         # Property name matches uri property above.
      {                                       # A file object (§3.17).
        "contents":                           # See §3.17.8.
          {                                   # A fileContent object (§3.2).
            "text": "..."                     # See §3.2.2.
          }
      }
    }
  }
}
```

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` – 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 name.

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 §3.11.13.2, 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 id 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" to distinguish the two versions.

```
{
  "originalUriBaseIds": {
    "GENERATED-1": "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": {
            "fileLocation": {
              "uri": "ui/window.g.cs",
              "uriBaseId": "GENERATED-1"
            },
            "region": {
              "startLine": 42
            },
            "contextRegion": {
              "startLine": 40,
              "endLine": 42
            }
          }
        }
      ]
    }
  ],

  "files": {
    "#GENERATED-1#ui/window.g.cs": {         # Unique property name.
      ...
    },

    "#GENERATED-2#ui/window.g.cs": {         # Unique property name
      ...
    }
  }
}
```


Appendix I. (Informative) Examples

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

I.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 (§3.11) with an empty `results` array (§3.11.16), as would happen if the tool detected no issues in any of the files it scanned.

```
{
  "version": "2.0.0",
  "runs": [
    {
      "tool": {
        "name": "CodeScanner"
      },
      "results": [
      ]
    }
  ]
}
```

I.2 Minimal recommended SARIF log file with source information

This is a minimal recommended SARIF log file for the case where

1. The analysis tool was run with the intent of scanning files and producing results (see §3.11.16), 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 (§3.11) with a `results` array (§3.11.16). The results array contains a single `result` object (§3.19) so the recommended elements of the `result` object can be shown.

Its `run.files` property (§3.11.13) specifies only those files in which the tool detected a result.

It does not contain a `run.logicalLocations` property (§3.11.14), because when physical location information is available, that property is optional (it "**MAY**" be present).

This example also includes a `run.rules` property (§3.11.17) 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": "2.0.0",
  "runs": [
    {
      "tool": {
        "name": "CodeScanner"
      },
      "files": {
        "file:///build.example.com/work/src/collections/list.cpp": {
          "mimeType": "text/x-c"
        }
      },
      "results": [
        {
          "ruleId": "C2001",
          "message": {
```

```
            "text": "Variable \"count\" was used without being initialized.",
            "richText": "Variable `count` was used without being initialized."
          },
          "locations": [
            {
              "physicalLocation": {
                "uri": "file://build.example.com/work/src/collections/list.cpp",
                "region": {
                  "startLine": 15
                }
              },
              "fullyQualifiedLogicalName": "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.3 Minimal recommended SARIF log file without source information

This is a minimal recommended SARIF file for the case where

1. The analysis tool was run with the intent of scanning files and producing results (see §3.11.16), 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 (§3.11) with a `results` array (§3.11.16). The results array contains a single `result` object (§3.19) so the recommended elements of the `result` object can be shown.

Its `run.files` property (§3.11.13) specifies only those files in which the tool detected a result.

It contains a `run.logicalLocations` property (§3.11.14), because when physical location information is not available, that property is recommended.

```
{
  "version": "2.0.0",
  "runs": [
    {
      "tool": {
        "name": "BinaryScanner"
      },
      "files": {
        "file://build.example.com/work/bin/example": {
          "mimeType": "application/vnd.microsoft.portable-executable"
        }
      },
      "logicalLocations": {
        "Example": {
          "name": "Example",
          "kind": "namespace"
        },
        "Example.Worker": {
          "name": "Worker",
          "kind": "type",
          "parentKey": "Example"
        },
        "Example.Worker.DoWork": {
```

```
          "name": "DoWork",
          "kind": "function",
          "parentKey": "Example.Worker"
        }
      },
      "results": [
        {
          "ruleId": "B6412",
          "message": {
            "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": [
            {
              "fullyQualifiedLogicalName": "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 (§3.9.6.4) and contains rule metadata for the language specified by `tool.language` (§3.12.8).

```
{
  "version": "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 Comprehensive SARIF file

The purpose of this example is to demonstrate the usage of as many SARIF elements as possible. Not all elements are shown, because some are mutually exclusive.

Because the purpose is to present as many elements as possibly, the file as a whole does not represent best practices for SARIF usage, nor does it represent the output of a single, coherent analysis. For example, the result presented in the file involves a runtime exception, but at the same time it is marked as `suppressedExternally` (to demonstrate the `result.suppressionStates` property), which is unrealistic.

```
{
  "version": "2.0.0",
  "$schema": "http://json.schemastore.org/sarif-2.0.0",
  "runs": [
    {
      "instanceGuid": "BC650830-A9FE-44CB-8818-AD6C387279A0",
      "logicalId": "Nightly code scan",
      "baselineInstanceGuid": "0A106451-C9B1-4309-A7EE-06988B95F723",
      "automationLogicalId": "Build-14.0.1.2-Release-20160716-13:22:18",
      "architecture": "x86",
      "tool": {
        "name": "CodeScanner",
        "fullName": "CodeScanner 1.1 for Unix (en-US)",
        "version": "2.1",
        "semanticVersion": "2.1.0",
        "fileVersion": "2.1.0.0",
        "language": "en-US",
        "sarifLoggerVersion": "1.25.0",
        "properties": {
          "copyright": "Copyright (c) 2017 by Example Corporation.
                        All rights reserved."
        }
      },
      "originalUriBaseIds": {
        "SRCROOT": "file://build.example.com/work/src/",
        "BINROOT": "file://build.example.com/work/bin/"
      },
      "invocations": [
        {
          "commandLine": "CodeScanner @build/collections.rsp",
          "responseFiles": [
            {
              "uri": "build/collections.rsp",
              "uriBaseId": "SRCROOT"
            }
          ],
          "startTime": "2016-07-16T14:18:25Z",
          "endTime": "2016-07-16T14:19:01Z",
          "machine": "BLD01",
          "account": "buildAgent",
          "processId": 1218,
          "fileName": "/bin/tools/CodeScanner",
          "workingDirectory": "/home/buildAgent/src",
          "environmentVariables": {
            "PATH": "/usr/local/bin:/bin:/bin/tools:/home/buildAgent/bin",
            "HOME": "/home/buildAgent",
            "TZ": "EST"
          },
          "configurationNotifications": [
            {
              "id": "UnknownRule",
              "ruleId": "ABC0001",
              "level": "warning",
              "message": {
                "text": "Could not disable rule \"ABC0001\" because
                         there is no rule with that id."
              }
```



```
          }
        ],
        "toolNotifications": [
          {
            "id": "CTN0001",
            "level": "note",
            "message": {
              "text": "Run started."
            }
          },
          {
            "id": "CTN9999",
            "ruleId": "C2152",
            "level": "error",
            "message": {
              "text": "Exception evaluating rule \"C2152\". Rule disabled;
                       run continues."
            },
            "physicalLocation": {
              "fileLocation": {
                "uri": "crypto/hash.cpp",
                "uriBaseId": "SRCROOT"
              }
            },
            "threadId": 52,
            "time": "2016-07-16T14:18:43.119Z",
            "exception": {
              "kind": "ExecutionEngine.RuleFailureException",
              "message": {
                "text": "Unhandled exception during rule evaluation."
              },
              "stack": {
                "frames": [
                  {
                    "message": {
                      "text": "Exception thrown"
                    },
                    "module": "RuleLibrary",
                    "threadId": 52,
                    "fullyQualifiedLogicalName":
                      "Rules.SecureHashAlgorithmRule.Evaluate",
                    "address": 10092852
                  },
                  {
                    "module": "ExecutionEngine",
                    "threadId": 52,
                    "fullyQualifiedLogicalName":
                        "ExecutionEngine.Engine.EvaluateRule",
                    "address": 10073356
                  }
                ]
              },
              "innerExceptions": [
                {
                  "kind": "System.ArgumentException",
                  "message": "length is < 0"
                }
              ]
            }
          },
          {
            "id": "CTN0002",
            "level": "note",
            "message": {
              "text": "Run ended."
            }
          }
        ]
      }
    ],
    "files": {
      "build/collections.rsp": {
        "mimeType": "text/plain",
```



```
          "contents": {
            "text": "-input src/collections/*.cpp -log out/collections.sarif -rules all
-disable C9999"
          }
        }
        "collections/list.cpp": {
          "mimeType": "text/x-c",
          "length": 980,
          "hashes": [
            {
              "algorithm": "sha-256",
              "value": "b13ce2678a8807ba0765ab94a0ecd394f869bc81"
            }
          ]
        },
        "app.zip": {
           "mimeType": "application/zip"
        },
        "app.zip#/docs/intro.docx": {
           "uri": "/docs/intro.docx",
           "mimeType":
              "application/vnd.openxmlformats-officedocument.wordprocessingml.document",
           "parentKey": "app.zip",
           "offset": 17522,
           "length": 4050
        }
      },
      "logicalLocations": {
        "collections::list::add": {
          "name": "add",
          "decoratedName": "?add@list@collections@@QAEXH@Z",
          "kind": "function",
          "parentKey": "collections::list"
        },
        "collections::list": {
          "name": "list",
          "kind": "type",
          "parentKey": "collections"
        },
        "collections": {
          "name": "collections",
          "kind": "namespace"
        }
      },
      "results": [
        {
          "ruleId": "C2001",
          "ruleMessageId": "default",
          "message": {
            "arguments": [
              "ptr"
            ]
          },
          "suppressionStates": [ "suppressedExternally" ],
          "baselineState": "existing",
          "level": "error",
          "analysisTarget": {
            "uri": "collections/list.cpp",
            "uriBaseId": "SRCROOT"
          },
          "locations": [
            {
              "physicalLocation": {
                "fileLocation": {
                  "uri": "collections/list.h",
                  "uriBaseId": "SRCROOT"
                },
                "region": {
                  "startLine": 15,
                  "startColumn": 9,
                  "endLine": 15,
                  "endColumn": 10,
                  "charLength": 1,
```



```
                "charOffset": 254,
                "snippet": {
                  "text": "add_core(ptr, offset, val);\n    return;"
                }
              }
            },
            "fullyQualifiedLogicalName": "collections::list:add"
          }
        ],
        "relatedLocations": [
          {
            "message": {
              "text": "Variable \"ptr\" was declared here.",
              "richText": "Variable `ptr` was declared here."
            },
            "physicalLocation": {
              "fileLocation": {
                "uri": "collections/list.h",
                "uriBaseId": "SRCROOT"
              },
              "region": {
                "startLine": 8,
                "startColumn": 5
              }
            },
            "fullyQualifiedLogicalName": "collections::list:add"
          }
        ],
        "codeFlows": [
          {
            "message": {
              "text": "Path from declaration to usage"
            },

            "threadFlows": [
              {
                "id": "thread-52",
                "locations": [
                  {
                    "step": 1,
                    "importance": "essential",
                    "message": {
                      "text": "Variable \"ptr\" declared.",
                      "richText": "Variable `ptr` declared."
                    },
                    "location": {
                      "physicalLocation": {
                        "fileLocation": {
                          "uri":"collections/list.h",
                          "uriBaseId": "SRCROOT"
                        },
                        "region": {
                          "startLine": 15,
                          "snippet": {
                            "text": "int *ptr;"
                          },
                        }
                      },
                      "fullyQualifiedLogicalName": "collections::list:add"
                    },
                    "module": "platform"
                  },
                  {
                    "step": 2,
                    "state": {
                      "y": "2",
                      "z": "4",
                      "y + z": "6",
                      "q": "7"
                    },
                    "importance": "unimportant",
                    "location": {
                      "physicalLocation": {
```



```
                        "fileLocation": {
                          "uri":"collections/list.h",
                          "uriBaseId": "SRCROOT"
                        },
                        "region": {
                          "startLine": 15
                          "snippet": {
                           "text": "offset = (y + z) * q + 1;"
                          }
                        }
                      },
                      "annotations": [
                        {
                          "startLine": 15,
                          "startColumn": 13,
                          "endColumn": 19
                          "message": {
                            "text": "(y + z) = 42",
                            "richText": "`(y + z) = 42`"
                          }
                        }
                      ],
                      "fullyQualifiedLogicalName": "collections::list:add"
                    },
                    "module": "platform"
                  },
                  {
                    "step": 3,
                    "importance": "essential",
                    "message": {
                      "text": "Uninitialized variable \"ptr\" passed to
                               method \"add_core\".",
                      "richText": "Uninitialized variable `ptr` passed to
                                   method `add_core`."
                    },
                    "location": {
                      "physicalLocation": {
                        "fileLocation": {
                          "uri":"collections/list.h",
                          "uriBaseId": "SRCROOT"
                        },
                        "region": {
                          "startLine": 25,
                          "snippet": {
                            "text": "add_core(ptr, offset, val)"
                          }
                        }
                      },
                      "fullyQualifiedLogicalName": "collections::list:add"
                    },
                    "module": "platform"
                  }
                ]
              }
            ]
          }
        ],
        "stacks": [
          {
            "message": {
              "text": "Call stack resulting from usage of uninitialized variable."
            },
            "frames": [
              {
                "message": {
                  "text": "Exception thrown."
                },
                "location": {
                  "physicalLocation": {
                    "fileLocation": {
                      "uri": "collections/list.h",
                      "uriBaseId": "SRCROOT"
                    },
```



```
                    "region": {
                      "startLine": 110,
                      "startColumn": 15
                    }
                  },
                  "fullyQualifiedLogicalName": "collections::list:add_core"
                },
                "module": "platform",
                "threadId": 52,
                "address": 10092852,
                "offset": 16,
                "parameters": [ "null", "0", "14" ]
              },
              {
                "location": {
                  "physicalLocation": {
                    "fileLocation": {
                      "uri": "collections/list.h",
                      "uriBaseId": "SRCROOT"
                    },
                    "region": {
                      "startLine": 43,
                      "startColumn": 15
                    }
                  },
                  "fullyQualifiedLogicalName": "collections::list:add"
                },
                "module": "platform",
                "threadId": 52,
                "address": 10092176,
                "offset": 84,
                "parameters": [ "14" ]
              },
              {
                "location": {
                  "physicalLocation": {
                    "fileLocation": {
                      "uri": "application/main.cpp",
                      "uriBaseId": "SRCROOT"
                    },
                    "region": {
                      "startLine": 28,
                      "startColumn": 9
                    }
                  },
                  "fullyQualifiedLogicalName": "main"
                 },
                "module": "application",
                "threadId": 52,
                "address": 10091200,
                "offset": 156
              }
            ]
          }
        ],
        "fixes": [
          {
            "description": {
              "text": "Initialize the variable to null"
            },
            "fileChanges": [
              {
                "fileLocation": {
                  "uri": "collections/list.h",
                  "uriBaseId": "SRCROOT"
                },
                "replacements": [
                  {
                    "deletedRegion": {
                      "startLine": 42
                    },
                    "insertedContent": {
                      "text": "A different line\n"
```



```
                    }
                  }
                ]
              }
            ]
          }
        ],
        "workItemUris": [
          "https://github.com/example/project/issues/42",
          "https://github.com/example/project/issues/54"
        ]
      }
    ],
    "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."
          }
        }
      }
    }
  ]
}
```


Appendix J. (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. 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. 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). 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.