ecma INTERNATIONAL

Standard ECMA-428

1st Edition / December 2025

Common Lifecycle Enumeration (CLE) specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

is the registered trademark of Ecma International

COPYRIGHT PROTECTED DOCUMENT

Contents

Introduction

The Common Lifecycle Enumeration (CLE) specification provides a standardized, machine-readable format for communicating lifecycle events of software and hardware components throughout the supply chain. As modern software systems increasingly rely on complex networks of dependencies and third-party components, understanding the lifecycle status of these components becomes critical for maintaining secure, compliant, and reliable systems.

This Standard addresses the challenge of tracking component lifecycles across diverse ecosystems by defining a unified format for expressing events such as releases, end-of-support announcements, end-of-life declarations, and component transitions. By providing a consistent structure for this information, CLE enables automated tooling to assess risks, plan migrations, and maintain supply chain transparency.

The CLE specification is designed to complement existing standards in the software supply chain ecosystem, including the Package-URL (PURL) specification for component identification and the Version Range (VERS) specification (inlined here, pending standardization) for version constraints. It integrates with Software Bill of Materials (SBOM) formats and transparency exchange protocols to provide comprehensive lifecycle visibility.

This document specifies version 1.0.0 of the Common Lifecycle Enumeration standard, developed under the auspices of Ecma International Technical Committee 54, Task Group 3 (TC54-TG3).

About this Standard

The document at https://tc54.org/ecmaXXX/ is the most accurate and up-to-date Common Lifecycle Enumeration specification.

This document is available as a single page and as multiple pages.

Contributing to this Standard

This Standard is developed on GitHub with the help of the OWASP community. There are a number of ways to contribute to the development of this Standard:

GitHub Repository: https://github.com/Ecma-TC54/ECMA-xxx-CLE
Issues: All Issues <https://github.com/Ecma-TC54/ECMA-xxx-CLE/issues>, File a New Issue <https://github.com/Ecma-TC54/ECMA-xxx-CLE/issues/new>
Pull Requests: All Pull Requests <https://github.com/Ecma-TC54/ECMA-xxx-CLE/pulls>, Create a New Pull Request <https://github.com/Ecma-TC54/ECMA-xxx-CLE/pulls/new>
Editors:

- Benji Visser
- Jordan Harband
- Steve Springett

Community:

- Chat: Slack Channel <https://owasp.slack.com/archives/C06GUKY03NC>

Refer to the colophon for more information on how this document is created.

This Ecma Standard was developed by Technical Committee 54 and was adopted by the General Assembly of December 2025.

Common Lifecycle Enumeration (CLE) specification

1 Scope

This Standard defines the Common Lifecycle Enumeration (CLE) specification version 1.0.0. The CLE provides a standardized format for communicating software component lifecycle events in a machine-readable format. This Standard defines the JSON schema and requirements for CLE documents.

2 Conformance

A conforming implementation of C LE must provide and consume objects that conform to the JSON Schema defined in this Standard. The key words "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may", and "optional" in this document are to be interpreted as described in RFC 2119.

3 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ECMA-428, *Package-URL (PURL) Specification*
https://ecma-tc54.github.io/ECMA-xxx-PURL/

ISO 8601, *Date and time — Representations for information interchange*
https://www.iso.org/standard/70907.html

JSON Schema Draft 2020-12
https://json-schema.org/draft/2020-12/json-schema-core

4 Terms and definitions

4.1
component
discrete unit of software or hardware that can be independently identified, versioned, and tracked through its lifecycle

4.2
lifecycle event
A significant occurrence in the existence of a component, such as release, end of support, or renaming.

4.3
PURL
Package-URL - a standardized format for identifying software packages across diverse ecosystems.

5 Types of work

The following types of work are commonly associated with software projects regardless of whether they are open source or not:

- **Marketing**: Promoting and advertising a software project to potential users.
- **Substantial Modifications**: Making substantial changes to a software project that are not considered bug fixes or security fixes, such as adding new features or functionality.
- **Bug Fixes**: Addressing and resolving issues or defects in a software project.
- **Security Fixes**: A distinct type of bug fix focused on security vulnerabilities that is useful to differentiate from other types of bug fixes.
- **Distribution**: The process of making a software project available for use by others.
- **Documentation**: Writing and updating documentation for a software project to help users understand how to use it.

6 Schema definition

6.1 JSON Schema

CLE is formally specified by a Draft 2020-12 JSON Schema.

Each published version of the specification is accompanied by a versioned meta-schema at a stable URI:

`https://cle.example.com/schema/cle-<major>.<minor>.<patch>.schema.json`

Every CLE document must include a top-level **`$schema`** field whose value is the URI of the meta-schema for the version it follows.

The **`$schema`** field serves two purposes:

- **Validation** – It allows tooling to validate that the document is structurally correct and conforms to the CLE specification.
- **Version Signaling** – It identifies the version of the CLE specification the document follows, so consumers can parse and interpret it accordingly.

6.2 Top-level fields

Table 1 — Required Fields

Field	Type	Description
`$schema`	string	URI identifying the JSON Schema document that describes the version of the CLE schema to use.
`identifier`	string or array[string]	Component identifier(s) in the PURL format. When an array is provided, each identifier alias must identify the exact same bits from the same software but distributed differently.
`updatedAt`	string	ISO 8601 timestamp indicating when this CLE document was last updated.
`events`	array	Ordered array of Event objects representing the component's lifecycle events. Must be ordered by ID in descending order (newest events with highest IDs first).

Table 2 — Additional Fields

Field	Type	Description
definitions	object	Container for reusable policy definitions that can be referenced throughout the document.
index	string	URL pointing to the index file that lists all CLE pages for this component. Only allowed to be present when pagination is used.
next	string	URL pointing to the next CLE page containing newer events (higher event IDs).

6.3 Definitions object

The definitions object allows specification of reusable policies and calculations that can be referenced by events.

6.3.1 Support definitions

The support object defines the support policies provided for a specific version or version range of a component. This may include first-party manufacturer support or third-party support options endorsed by the manufacturer.

Support policies are immutable once defined - the semantics and meaning of a support policy must not change over time for a given support policy ID. This ensures consistent interpretation of support commitments across the component's lifecycle.

Table 3 — Support Policy Fields

Field	Type	Required	Description
id	string	Yes	Unique identifier for the support policy.
description	string	Yes	Human readable description of the policy.
url	string	No	URL to detailed documentation about this support policy.

6.4 Event object

The base object that represents a discrete lifecycle event. All events share these common fields.

Events are immutable once created - the content and meaning of a specific event must not change after it has been published. Additionally, the ordering of events across shards must not change.

Table 4 — Event Required Fields

Field	Type	Description
id	integer	A unique, auto-incrementing integer identifier for the event.
type	string	The type of lifecycle event. Must be one of the defined Event Types.
effective	string	The time when the event takes effect, as an ISO 8601 formatted timestamp in UTC.
published	string	The time when the event was first published, as an ISO 8601 formatted timestamp in UTC.

7 Event types

7.1 released

Category: Version Event

Indicates when a component version is released and available for use.

Additional Required Fields:

- **`version`** - The version string of the released component

Additional Optional Fields:

- **`license`** - License identifier that summarizes the license as declared in the component's metadata

7.2 endOfDevelopment

Category: Version Event

The manufacturer or maintainer ceases work on Substantial Modifications for a specific version or version range of a component or service. Security Fixes and Bug Fixes will continue to be provided for this specific version or version range until endOfSupport is declared, but no new features or enhancements will be added.

Additional Required Fields:

- **`versions`** - Array of version specifications, per A
- **`supportId`** - Reference to a support policy defined in the definitions section

7.3 endOfSupport

Category: Version Event

The manufacturer or maintainer ceases providing Security Fixes and Bug Fixes for a specific version or version range of a component or service.

The **`supportId`** field must be included and used to specify which support policy is ending, referencing a support policy defined in the definitions section.

Additional Required Fields:

- **`versions`** - Array of version specifications
- **`supportId`** - Reference to a support policy defined in the definitions section

7.4 endOfLife

Category: Version Event

The manufacturer or maintainer formally ceases all work (including Distribution, Substantial Modifications, Bug Fixes, Security Fixes, Documentation, and Maintenance) for a specific version or version range of a component. No further updates, support, or distribution will be provided for this specific version or version range. The component is considered retired.

Additional Required Fields:

- **`versions`** - Array of version specifications, per A

7.5 endOfDistribution

Category: Version Event

The manufacturer or maintainer ceases distribution of a specific version or version range of a component or service. This should only be used when the manufacturer has control over the distribution of the component or service.

Additional Required Fields:

- **`versions`** - Array of version specifications, per A

7.6 endOfMarketing

Category: Version Event

The manufacturer or maintainer ceases marketing and promotion of a specific version or version range of a component or service. The component or service may still be available, and existing support policies may remain in effect, but the manufacturer will no longer seek new customers or promote its use.

Additional Required Fields:

- **`versions`** - Array of version specifications, per A

7.7 supersededBy

Category: Version Event

Indicates when a version of a component is superseded by another version of a component. This should only exist for components in which version progression is not implicit.

Additional Required Fields:

- **`supersededByVersion`** - Plain version string that supersedes it

Additional Optional Fields:

- **`versions`** - Array of version specifications, per A

7.8 componentRenamed

Category: Component Event

Indicates when a component is renamed.

Additional Required Fields:

- **`identifiers`** - Array of identifier objects specifying the new identifiers for the component

Additional Optional Fields:

- **`description`** - Human-readable description of the event
- **`references`** - List of URLs to supporting documentation

7.9 withdrawn

Category: Meta Event

Indicates that a previously published event is being withdrawn or revoked. This is used in a prepend-only event model where events cannot be modified, only new events can be added. When an event is withdrawn, it should be ignored during processing as if it never existed.

Additional Required Fields:

- **`eventId`** - The ID of the event being withdrawn

Additional Optional Fields:

- **`references`** - List of URLs to supporting documentation
- **`reason`** - Human-readable explanation for why the event is being withdrawn

8 Event categories

The CLE specification supports three distinct categories of events:

8.1 Version events

Events that affect specific versions or ranges of versions of a component. These events use either the **`version`** or **`range`** field to specify which versions are affected. Version Events include:

- **`released`**
- **`endOfDevelopment`**
- **`endOfSupport`**
- **`endOfLife`**
- **`endOfDistribution`**
- **`endOfMarketing`**
- **`supersededBy`**

8.2 Component events

Events that affect the component itself and may impact how the component is identified or referenced. These events often affect all versions of a component from the effective date forward. Component Events include:

- **`componentRenamed`** - Changes how the component is identified

8.3 Meta events

Events that affect other events in the history. These are used in the prepend-only event model to manage the event stream. Meta Events include:

- **`withdrawn`** - Revokes a previously published event

9 Event processing rules

CLE uses a prepend-only event model with the following rules:

1. **Immutability**: Once an event is published, it cannot be modified. New events must be added to correct or update information.
2. **Ordering**: Events must be ordered by ID in descending order (newest events with highest IDs first).

3. **ID Assignment**: Event IDs must be assigned as auto-incrementing integers in the order events are added (not by effective date).
4. **Processing Order**: When processing events, consumers should process them in reverse order (oldest to newest by ID) to build the correct state.
5. **Withdrawn Events**: When a **`withdrawn`** event is encountered:
   a. The event referenced by **`eventId`** should be ignored as if it never existed.
   b. The withdrawn event itself remains in the history for audit purposes.
   c. Any dependent events or calculations based on the withdrawn event should be recalculated.

10 Pagination

CLE supports pagination to handle components with extensive event histories. When a CLE file reaches the maximum limit of `100,000` events, it must be split into multiple pages.

10.1 Pagination rules

1. **Page Size Limit**: A single CLE page must not exceed `100,000` events.
2. **Event ID Continuity**: Event IDs must be globally unique and incrementing across all pages.
3. **Event Ordering**: Within each page, events must be ordered by ID in descending order.
4. **Page Chaining**: Pages are linked using the **`next`** field.
5. **Index File**: When pagination is used, an index file SHOULD be provided via the **`index`** field.

10.2 CLE index schema

The CLE index file provides a directory of all pages for a component.

Table 5 — Index Fields

Field	Type	Required	Description
`$schema`	string	Yes	URI identifying the CLE index schema version.
`pages`	array[object]	Yes	Array of page descriptor objects, ordered by event ID ranges.

Table 6 — Page Descriptor Fields

Field	Type	Required	Description
`url`	string	Yes	URL of the CLE page file.
`firstEventId`	integer	Yes	The lowest event ID in this page.
`lastEventId`	integer	Yes	The highest event ID in this page.

10.3 Processing paginated CLE files

When processing paginated CLE files:

1. Start with any page (typically discovered via the index file or a known entry point).
2. Process events within the current page according to standard processing rules.
3. To process all events, use the index file to discover all pages, or follow **`next`** links to traverse pages.
4. Event IDs are globally unique, so **`withdrawn`** events can reference events on any page.

11 Use cases

The CLE specification aims to address several common use cases in software component lifecycle management.

11.1 v1.0.0 Supported use cases

- **General Availability**: Track when new versions of a component are released and available for use.
- **End of Support/End of Life**: Communicate when versions will no longer receive updates or support.
- **Component Renaming**: Handle cases where a component's identifiers change.

11.2 Future use cases

These use cases will be addressed in future versions of the specification based on community feedback and requirements:

- **Complex License Changes**: Handling license changes for previously released versions.
- **Component Bundling/Unbundling**: Track when components are bundled into or extracted from larger packages.
- **Component Acquisition**: Handle cases where components change ownership.
- **Extended Support**: Support for third-party extended support offerings.
- **Third Party Claims**: Handling CLE from a third party perspective.
- **Component Forking**: Track when components are forked into new projects.
- **Export Restrictions**: Handle cases where components become restricted in certain regions.
- **Security Status Changes**: Track when components are marked as compromised or unsafe.

Annex A
(normative)

Version Range (VERS) specification (inlined)

This specification is a new syntax for dependency and vulnerable version ranges.

A.1 Version range specifier

A version range specifier (aka. "vers") is a URI string using the **`vers`** URI-scheme with this syntax:

```
vers:<versioning-scheme>/<version-constraint>|<version-constraint>|...
```

For example, to define a set of versions that contains either version **`1.2.3`**, or any versions greater than or equal to **`2.0.0`** but less than **`5.0.0`** using the **`node-semver`** versioning scheme used with the **`npm`** Package-URL type, the version range specifier will be:

```
vers:npm@1.2.3|≥=2.0.0|<5.0.0
```

`vers` is the URI-scheme and is an acronym for "VErsion Range Specifier". It has been selected because it is short, obviously about version and available for a future formal URI-scheme registration at IANA. The pipe "|" is used as a simple separator between `<version-constraint>`. Each `<version-constraint>` in this pipe-separated list contains a comparator and a version:

```
<comparator:version>
```

This list of **`<version-constraint>`** are signposts in the version timeline of a package that specify version intervals.

A **`<version>`** satisfies a version range specifier if it is contained within any of the intervals defined by these **`<version-constraint>`**.

A.2 Using version range specifiers

`vers` primary usage is to test if a version is within a range.

A version is within a version range if falls in any of the intervals defined by a range. Otherwise, the version is outside of the version range.

Some important usages derived from this include:

- **Resolving a version range specifier to a list of concrete versions.** In this case, the input is one or more known versions of a package. Each version is then tested to check if it lies within or outside the range. For example, given a vulnerability and the **`vers`** describing the vulnerable versions of a package, this process is used to determine if an existing package version is vulnerable.
- **Selecting one of several versions that are within a range.** In this case, given several versions that are within a range and several packages that express package dependencies qualified by a version range, a package management tool will determine and select the set of package versions that satisfy all the version ranges constraints of all dependencies. This usually requires deploying heuristics and algorithms (possibly complex such as sat solvers) that are ecosystem- and tool-specific and outside of the scope for this specification; yet **`vers`** could be used in tandem with **`purl`** to provide an input to this dependencies resolution process.

A.2.1 Examples

A single version in an npm package dependency:

- originally seen as a dependency on version "1.2.3" in a package.json manifest
- the version range spec is: **`vers:npm/1.2.3`**

A list of versions, enumerated:

- **`vers:pypi/0.0.0|0.0.1|0.0.2|0.0.3|1.0|2.0pre1`**

A complex statement about a vulnerability in a "maven" package that affects multiple branches each with their own fixed versions at https://repo1.maven.org/maven2/org/apache/tomee/apache-tomee/ Note how the constraints are sorted:

- "affects Apache TomEE 8.0.0-M1 - 8.0.1, Apache TomEE 7.1.0 - 7.1.2, Apache TomEE 7.0.0-M1 - 7.0.7, Apache TomEE 1.0.0-beta1 - 1.7.5."
- a normalized version range spec is:
  `vers:maven/>=1.0.0-beta1|<=1.7.5|>=7.0.0-M1|<=7.0.7|>=7.1.0|<=7.1.2|>=8.0.0-M1|<=`
- alternatively, four **`vers`** express the same range, using one **`vers`** for each vulnerable "branches":
  - **`vers:tomee/>=1.0.0-beta1|<=1.7.5`**
  - **`vers:tomee/>=7.0.0-M1|<=7.0.7`**
  - **`vers:tomee/>=7.1.0|<=7.1.2`**
  - **`vers:tomee/>=8.0.0-M1|<=8.0.1`**

Converting RubyGems custom syntax for dependency on gem. Note how the pessimistic version constraint is expanded:

- **`'library', '~> 2.2.0', '!= 2.2.1'`**
- the version range spec is: **`vers:gem/>=2.2.0|!= 2.2.1|<2.3.0`**

A.2.2 URI scheme

The **`vers`** URI scheme is an acronym for "VErsion Range Specifier". It has been selected because it is short, obviously about version and available for a future formal registration for this URI-scheme at the IANA registry.

The URI scheme is followed by a colon ":".

`<versioning-scheme>`

The **`<versioning-scheme>`** (such as **`npm`**, **`deb`**, etc.) determines:

- the specific notation and conventions used for a version string encoded in this scheme. Versioning schemes often specify a version segments separator and the meaning of each version segment, such as [major.minor.patch] in semver.
- how two versions are compared as greater or lesser to determine if a version is within or outside a range.
- how a versioning scheme-specific range notation can be transformed in the **`vers`** simplified notation defined here.

By convention the versioning scheme **should** be the same as the **`Package-URL`** package type for a given package ecosystem. It is OK to have other schemes beyond the purl type. A scheme could be specific to a single package name.

The **`<versioning-scheme>`** is followed by a slash "/".

A.2.3 `<version-constraint>`

After the **`<versioning-scheme>`** and "/" there are one or more **`<version-constraint>`** separated by a pipe "|". The pipe "|" has no special meaning beside being a separator.

Each **<version-constraint>** of this list is either a single **<version>** as in **1.2.3** for example or the combination of a **<comparator>** and a **<version>** as in **>=2.0.0** using this syntax:

```
<comparator><version>
```

A single version means that a version equal to this version satisfies the range spec. Equality is based on the equality of two normalized version strings according to their versioning scheme. For most schemes, this is a simple string equality. But schemes can specify normalization and rules for equality such as **pypi** with PEP440.

The special star "*" comparator matches any version. It must be used **alone** exclusive of any other constraint and must not be followed by a version. For example "vers:deb/*" represents all the versions of a Debian package. This includes past, current and possible future versions.

Otherwise, the **<comparator>** is one of these comparison operators:

- "!=": Version exclusion or inequality comparator. This means a version must not be equal to the provided version that must be excluded from the range. For example: "!=1.2.3" means that version "1.2.3" is excluded.
- "<", "<=": Lesser than or lesser-or-equal version comparators point to all versions less than or equal to the provided version. For example "<=1.2.3" means less than or equal to "1.2.3".
- ">", ">=": Greater than or greater-or-equal version comparators point to all versions greater than or equal to the provided version. For example ">=1.2.3" means greater than or equal to "1.2.3".

The **<versioning-scheme>** defines:

- how to compare two version strings using these comparators, and
- the structure of a version string such as "1.2.3" if any. For instance, the **semver** specification for version numbers defines a version as composed primarily of three dot-separated numeric segments named major, minor and patch.

A.3 Normalized, canonical representation and validation

The construction and validation rules are designed such that a **vers** is easier to read and understand by humans and straightforward to process by tools, attempting to avoid the creation of empty or impossible version ranges.

- Spaces are not significant and removed in a canonical form. For example "<1.2.3|>=2.0" and " < 1.2. 3 | > = 2 . 0" are equivalent.
- A version range specifier contains only printable ASCII letters, digits and punctuation.
- The URI scheme and versioning scheme are always lowercase as in **vers:npm**.
- The versions are case-sensitive, and a versioning scheme may specify its own case sensitivity.
- If a **version** in a **<version-constraint>** contains separator or comparator characters (i.e. **><=!*|**), it must be quoted using the URL quoting rules. This should be rare in practice.

The list of **<version-constraint>s** of a range are signposts in the version timeline of a package. With these few and simple validation rules, we can avoid the creation of most empty or impossible version ranges:

- **Constraints are sorted by version**. The canonical ordering is the versions order. The ordering of **<version-constraint>** is not significant otherwise but this sort order is needed when checking if a version is contained in a range.
- **Versions are unique**. Each **version** must be unique in a range and can occur only once in any **<version-constraint>** of a range specifier, irrespective of its comparators. Tools must report an error for duplicated versions.
- **There is only one star**: "*" must only occur once and alone in a range, without any other constraint or version.

Starting from a de-duplicated and sorted list of constraints, these extra rules apply to the comparators of any two contiguous constraints to be valid:

- "!=" constraint can be followed by a constraint using any comparator, i.e., any of "=", "!=", ">", ">=", "<", "<=" as comparator (or no constraint).

Ignoring all constraints with "!=" comparators:

- A "=" constraint must be followed only by a constraint with one of "=", ">", ">=" as comparator (or no constraint).

And ignoring all constraints with "=" or "!=" comparators, the sequence of constraint comparators must be an alternation of greater and lesser comparators:

- "<" and "<=" must be followed by one of ">", ">=" (or no constraint).
- ">" and ">=" must be followed by one of "<", "<=" (or no constraint).

Tools must report an error for such invalid ranges.

A.4 Parsing and validating version range specifiers

To parse a version range specifier string:

- Remove all spaces and tabs.
- Start from left, and split once on colon ":".
- The left hand side is the URI-scheme that must be lowercase.
  - Tools must validate that the URI-scheme value is **`vers`**.
- The right hand side is the specifier.
- Split the specifier from left once on a slash "/".
- The left hand side is the **`<versioning-scheme>`** that must be lowercase. Tools should validate that the **`<versioning-scheme>`** is a known scheme.
- The right hand side is a list of one or more constraints. Tools must validate that this constraints string is not empty ignoring spaces.
- If the constraints string is equal to "*", the **`<version-constraint>`** is "*". Parsing is done and no further processing is needed for this **`vers`**. A tool should report an error if there are extra characters beyond "*".
- Strip leading and trailing pipes "|" from the constraints string.
- Split the constraints on pipe "|". The result is a list of **`<version-constraint>`**. Consecutive pipes must be treated as one and leading and trailing pipes ignored.
- For each **`<version-constraint>`**:
  - Determine if the **`<version-constraint>`** starts with one of the two comparators:
    - If it starts with ">=", then the comparator is ">=".
    - If it starts with "<=", then the comparator is "<=".
    - If it starts with "!=", then the comparator is "!=".
    - If it starts with "<", then the comparator is "<".
    - If it starts with ">", then the comparator is ">".
    - Remove the comparator from **`<version-constraint>`** string start. The remaining string is the version.
  - Otherwise the version is the full **`<version-constraint>`** string (which implies an equality comparator of "=")
  - Tools should validate and report an error if the version is empty.
  - If the version contains a percent "%" character, apply URL quoting rules to unquote this string.
  - Append the parsed (comparator, version) to the constraints list.

Finally:

- The results are the **`<versioning-scheme>`** and the list of **`<comparator, version>`** constraints.

Tools should optionally validate and simplify the list of **`<comparator, version>`** constraints once parsing is complete:

- Sort and validate the list of constraints.
- Simplify the list of constraints.

A.4.1 Version constraints simplification

Tools can simplify a list of **`<version-constraint>`** using this approach:

These pairs of contiguous constraints with these comparators are valid:

- != followed by anything
- =, <, or <= followed by =, !=, >, or >=
- >, or >= followed by !=, <, or <=

These pairs of contiguous constraints with these comparators are redundant and invalid (ignoring any != since they can show up anywhere):

- =, < or <= followed by < or <=: this is the same as < or <=
- > or >= followed by =, > or >=: this is the same as > or >=

A procedure to remove redundant constraints can be:

- Start from a list of constraints of comparator and version, sorted by version and where each version occurs only once in any constraint.
- If the constraints list contains a single constraint (star, equal or anything) return this list and simplification is finished.
- Split the constraints list in two sub lists:
  - a list of "unequal constraints" where the comparator is "!="
  - a remainder list of "constraints" where the comparator is not "!="
- If the remainder list of "constraints" is empty, return the "unequal constraints" list and simplification is finished.
- Iterate over the constraints list, considering the current and next contiguous constraints, and the previous constraint (e.g., before current) if it exists:
  - If current comparator is ">" or ">=" and next comparator is "=", ">" or ">=", discard next constraint
  - If current comparator is "=", "<" or "<=" and next comparator is <" or <=", discard current constraint. Previous constraint becomes current if it exists.
  - If there is a previous constraint:
    - If previous comparator is ">" or ">=" and current comparator is "=", ">" or ">=", discard current constraint
    - If previous comparator is "=", "<" or "<=" and current comparator is <" or <=", discard previous constraint
- Concatenate the "unequal constraints" list and the filtered "constraints" list
- Sort by version and return.

A.4.2 Checking if a version is contained within a range

To check if a "tested version" is contained within a version range:

- Start from a parsed a version range specifier with:
  - a versioning scheme
  - a list of constraints of comparator and version, sorted by version and where each version occurs only once in any constraint.
- If the constraint list contains only one item and the comparator is "*", then the "tested version" is IN the range. Check is finished.
- Select the version equality and comparison procedures suitable for this versioning scheme and use these for all version comparisons performed below.
- If the "tested version" is equal to the any of the constraint versions where the constraint comparator is for equality (any of "=", "<=", or ">=") then the "tested version" is in the range. Check is finished.
- If the "tested version" is equal to the any of the constraint versions where the constraint comparator is "=!" then the "tested version" is NOT in the range. Check is finished.
- Split the constraint list in two sub lists:
  - a first list where the comparator is "=" or "!="
  - a second list where the comparator is neither "=" nor "!="
- Iterate over the current and next contiguous constraints pairs (aka. pairwise) in the second list.

- For each current and next constraint:
  - If this is the first iteration and current comparator is "<" or <=" and the "tested version" is less than the current version then the "tested version" is IN the range. Check is finished.
  - If this is the last iteration and next comparator is ">" or >=" and the "tested version" is greater than the next version then the "tested version" is IN the range. Check is finished.
  - If current comparator is ">" or >=" and next comparator is "<" or <=" and the "tested version" is greater than the current version and the "tested version" is less than the next version then the "tested version" is IN the range. Check is finished.
  - If current comparator is "<" or <=" and next comparator is ">" or >=" then these versions are out the range. Continue to the next iteration.
- Reaching here without having finished the check before means that the "tested version" is NOT in the range.

A.4.3 Notes and caveats

- Comparing versions from two different versioning schemes is an error. Even though there may be some similarities between the **`semver`** version of an npm and the **`deb`** version of its Debian packaging, the way versions are compared specific to each versioning scheme and may be different. Tools should report an error in this case.
- All references to sorting or ordering of version constraints means sorting by version. And sorting by versions always implies using the versioning scheme-specified version comparison and ordering.

A.5 Some of the known versioning schemes

These are a few known versioning schemes for some common Package-URL types (aka. **`ecosystem`**).

- **deb**: Debian and Ubuntu https://www.debian.org/doc/debian-policy/ch-relationships.html Debian uses these comparators: <<, <=, =, >= and >>.
- **rpm**: RPM distros https://rpm-software-management.github.io/rpm/manual/dependencies.html A simplified rmpvercmp version comparison routine is used by Arch Linux Pacman.
- **gem**: RubyGems https://guides.rubygems.org/patterns/#semantic-versioning which is similar to **`node-semver`** for its syntax, but does not use semver versions.
- **npm**: npm uses node-semver which is based on semver with its own range notation https://github.com/npm/node-semver#ranges A similar but different scheme is used by Rust https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html and several other package types may use **`node-semver`**-like ranges. But most of these related schemes are not strictly the same as what is implemented in **`node-semver`**. For instance PHP **`composer`** may need its own scheme as this is not strictly **`node-semver`**.
- **composer**: PHP https://getcomposer.org/doc/articles/versions.md
- **pypi**: Python https://www.python.org/dev/peps/pep-0440/
- **cpan**: Perl https://perlmaven.com/how-to-compare-version-numbers-in-perl-and-for-cpan-modules
- **golang**: Go modules https://golang.org/ref/mod#versions use **`semver`** versions with a specific minimum version resolution algorithm.
- **maven**: Apache Maven supports a math interval notation which is rarely seen in practice http://maven.apache.org/enforcer/enforcer-rules/versionRanges.html
- **nuget**: NuGet https://docs.microsoft.com/en-us/nuget/concepts/package-versioning#version-ranges Note that Apache Maven and NuGet are following a similar approach with a math-derived intervals syntax as in https://en.wikipedia.org/wiki/Interval_(mathematics)
- **gentoo**: Gentoo https://wiki.gentoo.org/wiki/Version_specifier
- **alpine**: Alpine linux https://gitlab.alpinelinux.org/alpine/apk-tools/-/blob/master/src/version.c which is using Gentoo-like conventions.

These are generic schemes, to use sparingly for special cases:

- **generic**: a generic version comparison algorithm (which will be specified later, likely based on a split on any wholly alpha or wholly numeric segments and dealing with digit and string comparisons, like is done in libversion)
- **none**: a generic versioning scheme for a range containing no version. **`vers:none/*`** is the only valid vers form for this scheme.
- **all**: a generic versioning scheme for a range containing all versions. **`vers:all/*`** is the only valid vers form

for this scheme.
- **intdot**: a generic versioning scheme that allows version components to be specified as integers separated by dots, e.g. **`10.234.5.12`**. Versions specified in this scheme consist of ASCII digits only, formatted with only non-negative integers, and ignoring leading zeros. Interpretation of the version should stop at the first character that is not a digit or a dot.
- **lexicographic**: a generic versioning scheme that compares versions based on lexicographic order, interpreted as UTF-8. Strings should be compared bytewise as unsigned bytes without normalization. UTF-8 encoding is defined in https://datatracker.ietf.org/doc/html/rfc3629.
- **semver**: a generic scheme that uses the same syntax as **`semver`**. It follows the MAJOR.MINOR.PATCH format and is defined in the Semantic Versioning Specification 2.0.0, see https://semver.org/spec/v2.0.0.html.
- **datetime**: a generic scheme that uses a timestamp for comparison. The timestamp must adhere to RFC3339, section 5.6, see https://www.rfc-editor.org/rfc/rfc3339#section-5.6.

A separate document will provide details for each versioning scheme and:

- how to convert its native range notation to the **`vers`** notation and back.
- how to compare and sort two versions in a range.

This versioning schemes document will also explain how to convert CVE and OSV ranges to **`vers`**.

A.6 Implementations

- Python: https://github.com/nexB/univers
- Java: https://github.com/nscuro/versatile
- Yours!

A.7 Related efforts and alternative

- CUDF defines a generic range notation similar to Debian and integer version numbers from the sequence of versions for universal dependencies resolution https://www.mancoosi.org/cudf/primer/
- OSV is an "Open source vulnerability DB and triage service." It defines vulnerable version range semantics using a minimal set of comparators for use with package "ecosystem" and version range "type". https://github.com/google/osv
- libversion is a library for general purpose version comparison using a unified procedure designed to work with many package types. https://github.com/repology/libversion
- unified-range is a library for uniform version ranges based on the Maven version range spec. It supports Apache Maven and npm ranges https://github.com/snyk/unified-range
- dephell specifier is a library to parse and evaluate version ranges and "work with version specifiers (can parse PEP-440, SemVer, Ruby, NPM, Maven)" https://github.com/dephell/dephell_specifier

A.8 Why not reuse existing version range notations?

Most existing version range notations are tied to a specific version string syntax and are therefore not readily applicable to other contexts. For example, the use of elements such as tilde and caret ranges in RubyGems, npm or Dart notations implies that a certain structure exists in the version string (semver or semver- like). The inclusion of these additional comparators is a result of the history and evolution in a given package ecosystem to address specific needs.

In practice, the unified and reduced set of comparators and syntax defined for **`vers`** has been designed such that all these notations can be converted to a **`vers`** and back from a **`vers`** to the original notation.

In contrast, this would not be possible with existing notations. For instance, the Python notation may not work with npm semver versions and reciprocally.

There are likely to be a few rare cases where round tripping from and to **`vers`** may not be possible, and in any case round tripping to and from **`vers`** should produce equivalent results and even if not strictly the same original strings.

Another issue with existing version range notations is that they are primarily designed for dependencies and not for vulnerable ranges. In particular, a vulnerability may exist for multiple "version branches" of a given package such as with Django 2.x and 3.x. Several version range notations have difficulties to communicate these as typically all the version constraints must be satisfied. In contrast, a vulnerability can affect multiple disjoint version ranges of a package and any version satisfying these constraints would be vulnerable: it may not be possible to express this with a notation designed exclusively for dependent versions resolution.

Finally, one of the goals of this spec is to be a compact yet obvious Package-URL companion for version ranges. Several existing and closely related notations designed for vulnerable ranges are verbose specifications designed for use in API with larger JSON documents.

A.8.1 Why not use the OSV Ranges?

See:

- https://ossf.github.io/osv-schema/

`vers` and the OSSF OSV schema vulnerable ranges are equivalent and **`vers`** provides a compact range notation while OSV provides a more verbose JSON notation.

`vers` borrows the design from and was informed by the OSV schema spec and its authors.

OSV uses a minimalist set of only three comparators:

- "=" to enumerate versions,
- ">=" for the version that introduced a vulnerability, and
- "<" for the version that fixed a vulnerability.

OSV Ranges support neither ">" nor "!=" comparators making it difficult to express some ranges that must exclude a version. This may not be an issue for most vulnerable ranges yet:

- this makes it difficult or impossible to precisely express certain dependency and vulnerable ranges when a version must be excluded and the set of existing versions is not yet known,
- this make some ranges more verbose such as with the CVE v5 API ranges notation that can include their upper limit and would need two constraints.

Another high level difference between the two specifications is the codes used to qualify a range package "ecosystem" value that resembles closely the Package-URL package "type" used in **`vers`**. This spec will provide a strict mapping between the OSV ecosystem and the **`vers`** versioning schemes values.

A.8.2 Why not use the CVE v5 API Ranges?

See:

- https://github.com/CVEProject/cve-schema/blob/master/schema/v5.0/CVE_JSON_5.0_schema.json#L303
- https://github.com/CVEProject/cve-schema/blob/master/schema/v5.0/CVE_JSON_5.0_schema.json#L123

The version 5 of the CVE JSON data format defines version ranges with a starting version, a versionType, and an upper limit for the version range as lessThan or lessThanOrEqual or as an enumeration of versions. The versionType and the package collectionURL possible values are only indicative and left out of this specification and both seem strictly equivalent to the Package-URL "type" on the one hand and the**`vers`** versioning scheme on the other hand.

The semantics and expressiveness of each range are similar and **`vers`** provides a compact notation rather than a more verbose JSON notation. **`vers`** supports strictly the conversion of any CVE v5 range to its notation and further provides a concrete list of well known versioning schemes. **`vers`** design was informed by the CVE v5 API schema spec and its authors.

When CVE v5 becomes active, this spec will provide a strict mapping between the CVE **`versionType`** and the **`vers`** versioning schemes values. Furthermore, this spec and the Package-URL "types" should be updated accordingly to provide a mapping with the upcoming CVE **`collectionURL`** that will be effectively used.

There is one issue with CVE v5: it introduces a new trailing "*" notation that does not exist in most version ranges notations and may not be computable easily in many cases. The description of the "lessThan" property is:

> The non-inclusive upper limit of the range. This is the least version NOT in the range. The usual version syntax is expanded to allow a pattern to end in an asterisk (**), indicating an arbitrarily large number in the version ordering. For example, {version: 1.0 lessThan: 1.**} would describe the entire 1.X branch for most range kinds, and {version: 2.0, lessThan: *} describes all versions starting at 2.0, including 3.0, 5.1, and so on.

The conversion to **`vers`** range should be:

- with a version 1.0 and "lessThan": "*", the **`vers`** equivalent is: **`>=1.0`**.
- with a version 1.0 and "lessThan": "2.*", the **`vers`** equivalent can be computed for **`semver`** versions as **`>=1.0|<2`** but this is not accurate because the versioning schemes have different rules. For instance, pre-release may be treated in some case as part of the v1. branch and in some other cases as part of the v2. branch. It is not clear if with "2.*" the CVE v5 spec means:
  - **`<2`**
  - or something that excludes any version string that starts with **`2.`**

And in this case, with the expression "lessThan": "2.*" using a **`semver`** version, it is not clear if **`2.0.0-alpha`** is "lessThan"; semver sorts it before **`2.0`** and after **`1.0`**, e.g., in **`semver 2.0.0-alpha`** is "less than" **`2`**.

A.8.3 Why not use the NVD CPE Ranges?

See:

- https://nvd.nist.gov/vuln/vulnerability-detail-pages#divRange
- https://nvd.nist.gov/developers/vulnerabilities#divResponse
- https://csrc.nist.gov/schema/nvd/feed/1.1/nvd_cve_feed_json_1.1.schema

The version ranges notation defined in the JSON schema of the CVE API payload uses these four fields: **`versionStartIncluding`**, **`versionStartExcluding`**, **`versionEndIncluding`** and **`versionEndExcluding`**. For example:

```
"versionStartIncluding": "7.3.0",
"versionEndExcluding": "7.3.31",
"versionStartExcluding" : "9.0.0",
"versionEndIncluding" : "9.0.46",
```

In addition to these ranges, the NVD publishes a list of concrete CPEs with versions resolved for a range with daily updates at https://nvd.nist.gov/vuln/data-feeds#cpeMatch

Note that the NVD CVE configuration is a complex specification that goes well beyond version ranges and is used to match comprehensive configurations across multiple products and version ranges. **`vers`** focus is exclusively versions.

In contrast with **`vers`** compact notation, the NVD JSON notation is more verbose, yet **`vers`** supports strictly the conversion of any CPE range.

A.8.4 Why not use node-semver ranges?

See:

- https://github.com/npm/node-semver#ranges

The node-semver spec is similar to but much more complex than this spec. This is an AND of ORs constraints with a few practical issues:

- A space means "AND", therefore white spaces are significant. Having significant white spaces in a string makes normalization more complicated and may be a source of confusion if you remove the spaces from the string. **`vers`** avoids the ambiguity of spaces by ignoring them.
- The advanced range syntax has grown to be rather complex using hyphen ranges, stars ranges, carets and tilde constructs that are all tied to the JavaScript and npm ways of handling versions in their ecosystem and are bound furthermore to the semver semantics and its npm implementation. These are not readily reusable elsewhere. The multiple comparators and modifiers make the notation grammar more complex to parse and process for a machine and harder to read for human.

Notations that are directly derived from node-semver as used in Rust and PHP Composer have the same issues.

A.8.5 Why not use Python PEP-0440 ranges?

See:

- https://www.python.org/dev/peps/pep-0440/#version-specifiers

The Python pep-0440 "Version Identification and Dependency Specification" provides a comprehensive specification for Python package versioning and a notation for "version specifiers" to express the version constraints of dependencies.

This specification is similar to this **`vers`** spec, with more operators and aspects specific to the versions used only in the Python ecosystem.

- In particular pep-0440 uses tilde, triple equal and wildcard star operators that are specific to how two Python versions are compared.
- The comma separator between constraints is a logical "AND" rather than an "OR". The "OR" does not exist in the syntax making some version ranges harder to express, in particular for vulnerabilities that may affect several exact versions or ranges for multiple parallel release branches. Ranges such as "Django 1.2 or later, or Django 2.2 or later or Django 3.2 or later" are difficult to express without an "OR" logic.

A.8.6 Why not use RubyGems requirements notation?

See:

- https://guides.rubygems.org/patterns/#declaring-dependencies

The RubyGems specification suggests but does not enforce using semver. It uses operators similar to the **`node-semver`** spec with the difference of the ">" AKA. PESSIMISTIC OPERATOR VS. A PLAIN "" tilde used in node-semver. This operator implies some semver-like versioning, yet gem versions are not strictly semver. This makes the notation complex to implement and impractical to reuse in places that do not use the same Ruby-specific semver-like semantics.

A.8.7 Why not use fewer comparators with only =, >= and <?

For instance, the OSV schema adopts a reduced set of only three comparators:

- "=" is implied when used to enumerate vulnerable versions
- ">=" (greater or equal) is for the version that introduces a vulnerability
- "<" (lesser) is for the version that fixes a vulnerability

This approach is simpler and works well for most vulnerable ranges but it faces limitations when converting from other notations:

- ">" cannot be converted reliably to ">=" unless you know all the versions and these will never change.
- "<=" cannot be converted reliably to "<" unless you know all the versions and these will never change.

- "!=" cannot be converted reliably: there is no ">" comparator to create an unequal equivalent of "><"; and a combo of ">=" and "<" is not equivalent to inequality unless you know all the versions and these will never change.

A.8.8 Why not use richer comparators such as tilde, caret and star?

Some existing notations such as used with npm, gem, python, or composer provide syntactic shorthand such as:

- a "pessimistic operator" using tilde, > OR = as in "1.3" OR ">1.2.3"
- a caret ^ prefix as in "^ 1.2"
- using a star in a version segment as in "1.2.*"
- dash-separated ranges as in "1.2 - 1.4"
- arbitrary string equality such as "===1.2"

Most of these notations can be converted without loss to the **`vers`** notation. Furthermore these notations typically assume a well defined version string structure specific to their package ecosystem and are not reusable in another ecosystem that would not use the exact same version conventions.

For instance, the tilde and caret notations demand that you can reliably infer the next version (aka. "bump") from a given version; this is possible only if the versioning scheme supports this operation reliably for all its accepted versions.

A.8.9 Why not use mathematical interval notation for ranges?

Apache Maven and NuGet use a mathematical interval notation with comma-separated "[", "]", "(" and ")" to declare version ranges.

All other known range notations use the more common ">", "<", and "=" as comparators. **`vers`** adopts this familiar approach.

A.9 References

Here are some of the discussions that led to the creation of this specification:

- https://github.com/package-url/purl-spec/issues/66
- https://github.com/package-url/purl-spec/issues/84
- https://github.com/package-url/purl-spec/pull/93
- https://github.com/nexB/vulnerablecode/issues/119
- https://github.com/nexB/vulnerablecode/issues/140
- https://github.com/nexB/univers/pull/11

A.10 License

Annex B
(informative)

Example CLE document


```
{
  "$schema": "https://TODO/cle.v1.0.0.json",
  "identifier": "pkg:npm/example-component",
  "updatedAt": "2021-01-15T00:00:00Z",
  "definitions": {
    "support": [
      {
        "id": "standard",
        "description": "Standard product support policy",
        "url": "https://example.com/support/standard"
      }
    ]
  },
  "events": [
    {
      "id": 5,
      "type": "withdrawn",
      "effective": "2021-01-15T00:00:00Z",
      "published": "2021-01-15T00:00:00Z",
      "eventId": 2,
      "reason": "The endOfSupport date was incorrect.",
      "references": [
        "https://example.com/support-correction"
      ]
    },
    {
      "id": 4,
      "type": "endOfSupport",
      "effective": "2021-01-01T00:00:00Z",
      "published": "2021-01-01T00:00:00Z",
      "versions": [
        {
          "range": "vers:npm/>=1.0.0|<2.0.0"
        }
      ],
      "supportId": "standard"
    },
    {
      "id": 3,
      "type": "componentRenamed",
      "effective": "2020-01-01T00:00:00Z",
      "published": "2020-01-01T00:00:00Z",
      "description": "Component renamed due to acquisition",
      "identifiers": [
        {
          "type": "PURL",
          "value": "pkg:npm/new-component"
        }
      ]
    },
    {
```

```
      "id": 2,
      "type": "endOfSupport",
      "effective": "2020-01-01T00:00:00Z",
      "published": "2020-01-01T00:00:00Z",
      "versions": [
        {
          "range": "vers:npm/>=1.0.0|<2.0.0"
        }
      ],
      "supportId": "standard"
    },
    {
      "id": 1,
      "type": "released",
      "effective": "2019-01-01T00:00:00Z",
      "published": "2019-01-01T00:00:00Z",
      "version": "1.0.0",
      "license": "MIT"
    }
  ]
}
```

Colophon

This Standard is authored on GitHub <https://github.com/Ecma-TC54/ECMA-xxx-CLE> in a plaintext source format called Ecmarkup <https://github.com/bterlson/ecmarkup>. Ecmarkup is an HTML and Markdown dialect that provides a framework and toolset for authoring Ecma specifications in plaintext and processing the specification into a full-featured HTML rendering that follows the editorial conventions for this document. Ecmarkup builds on and integrates a number of other formats and technologies including Grammarkdown <https://github.com/rbuckton/grammarkdown> for defining syntax and Ecmarkdown <https://github.com/domenic/ecmarkdown> for authoring algorithm steps. PDF renderings of this Standard are produced using a print stylesheet which takes advantage of the CSS Paged Media specification and is converted using PrinceXML <https://www.princexml.com/>.

Prior editions of this Standard were authored using Word—the Ecmarkup source text that formed the basis of this edition was produced by converting the ECMAScript 2015 Word document to Ecmarkup using an automated conversion tool.

Software License

Ecma International
Rue du Rhone 114
CH-1204 Geneva
Tel: +41 22 849 6000
Fax: +41 22 849 6001
Web: https://ecma-international.org/