GA4GH Variation Representation Specification

Release HEAD

Jul 01, 2020

Contents

The Variation Representation Specification (VRS, pronounced "verse") is a standard developed by the Global Alliance for Genomic Health to facilitate and improve sharing of genetic information. The Specification consists of a JSON Schema for representing many classes of genetic variation, conventions to maximize the utility of the schema, and a Python implementation that promotes adoption of the standard.

CHAPTER 1

Introduction

Maximizing the personal, public, research, and clinical value of genomic information will require that clinicians, researchers, and testing laboratories exchange genetic variation data reliably. The Variation Representation Specification (VRS, pronounced "verse") — written by a partnership among national information resource providers, major public initiatives, and diagnostic testing laboratories — is an open specification to standardize the exchange of variation data.

Here we document the primary contributions of this specification for variation representation:

- **Terminology and information model.** Definitions for biological terms may be abstract or intentionally ambiguous, often accurately reflecting scientific uncertainty or understanding at the time. Abstract and ambiguous terms are not readily translatable into a representation of knowledge. Therefore, the specification begins with precise computational definitions for biological concepts that are essential to representing sequence variation. The VRS information model specifies how the computational definitions are to be represented in fields, semantics, objects, and object relationships.
- **Machine readable schema.** To be useful for information exchange, the information model should be realized in a schema definition language. The VRS schema is currently implemented using JSON Schema, however it is intended to support translations to other schema systems (e.g. XML, OpenAPI, and GraphQL). The schema repository includes language-agnostic tests for ensuring schema compliance in downstream implementations.
- **Conventions that promote reliable data sharing.** VRS recommends conventions regarding the use of the schema and that facilitate data sharing. For example, VRS recommends using fully justified allele normalization using an algorithm inspired by NCBI's SPDI project.
- **Globally unique computed identifiers.** This specification also recommends a specific algorithm for constructing distributed and globally-unique identifiers for molecular variation. Importantly, this algorithm enables data providers and consumers to computationally generate consistent, globally unique identifiers for variation without a central authority.
- **A Python implementation.** We provide a Python package (vr-python) that demonstrates the above schema and algorithms, and supports translation of existing variant representation schemes into VRS for use in genomic data sharing. It may be used as the basis for development in Python, but it is not required in order to use VRS.

The machine readable schema definitions and example code are available online at the VRS repository (https://github.com/ga4gh/vr-spec).

Readers may wish to view a *complete example* before reading the specification.

For a discussion of VRS with respect to existing standards, such as HGVS, SPDI, and VCF, see *Relationship of VRS to existing standards*.

CHAPTER 2

Terminology & Information Model

When biologists define terms in order to describe phenomena and observations, they rely on a background of human experience and intelligence for interpretation. Definitions may be abstract, perhaps correctly reflecting uncertainty of our understanding at the time. Unfortunately, such terms are not readily translatable into an unambiguous representation of knowledge.

For example, “allele” might refer to “an alternative form of a gene or locus” [Wikipedia], “one of two or more forms of the DNA sequence of a particular gene” [ISOGG], or “one of a set of coexisting sequence alleles of a gene” [Sequence Ontology]. Even for human interpretation, these definitions are inconsistent: does the definition precisely describe a specific change on a specific sequence, or, rather, a more general change on an undefined sequence? In addition, all three definitions are inconsistent with the practical need for a way to describe sequence changes outside regions associated with genes.

The computational representation of biological concepts requires translating precise biological definitions into data structures that can be used by implementers. This translation should result in a representation of information that is consistent with conventional biological understanding and, ideally, be able to accommodate future data as well. The resulting *computational representation* of information should also be cognizant of computational performance, the minimization of opportunities for misunderstanding, and ease of manipulating and transforming data.

Accordingly, for each term we define below, we begin by describing the term as used by biologists (**biological definition**) as available. When a term has multiple biological definitions, we explicitly choose one of them for the purposes of this specification. We then provide a computer modelling definition (**computational definition**) that reformulates the biological definition in terms of information content. We then translate each of these computational definitions into precise specifications for the (**logical model**). Terms are ordered “bottom-up” so that definitions depend only on previously-defined terms.

Note: The keywords “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.

2.1 Data Model Notes and Principles

- VRS uses snake_case to represent compound words. Although the schema is currently JSON-based (which would typically use camelCase), VRS itself is intended to be neutral with respect to languages and database.
- VRS objects are value objects. Two objects are considered equal if and only if their respective attributes are equal. As value objects, VRS objects are used as primitive types and SHOULD NOT be used as containers for related data. Instead, related data should be associated with VRS objects through identifiers. See *Computed Identifiers*.
- Error handling is intentionally unspecified and delegated to implementation. VRS provides foundational data types that enable significant flexibility. Except where required by this specification, implementations may choose whether and how to validate data. For example, implementations MAY choose to validate that particular combinations of objects are compatible, but such validation is not required.
- We recognize that a common desire may be to have human-readable identifiers associated with VRS objects. We recommend using the _id field (see *Optional Attributes* below) to create a lookup for any such identifiers (see *example usage*), and provide reference methods for creating VRS identifiers from other common variant formats (see the *HGVS translation example*).

2.2 Optional Attributes

- VRS attributes use a leading underscore to represent optional attributes that are not part of the value object. Such attributes are not considered when evaluating equality or creating computed identifiers. Currently, the only such attribute in the specification is the *_id* attribute.
- The *_id* attribute is available to identifiable objects, and MAY be used by an implementation to store the identifier for a VRS object. If used, the stored *_id* element MUST be a *CURIE*. If used for creating a *Truncated Digest (sha512t24u)* for parent objects, the stored element must be a *GA4GH Computed Identifier*.

2.3 Primitive Concepts

2.3.1 CURIE

Biological definition

None.

Computational definition

A CURIE formatted string. A CURIE string has the structure `prefix:reference` (W3C Terminology).

Implementation guidance

- All identifiers in VRS MUST be a valid Compact URI (CURIE), regardless of whether the identifier refers to GA4GH VRS objects or external data.
- For GA4GH VRS objects, this specification RECOMMENDS using globally unique *Computed Identifiers* for use within *and* between systems.
- For external data, CURIE-formatted identifiers MUST be used. When an appropriate namespace exists at identifiers.org, that namespace MUST be used. When an appropriate namespace does not exist at identifiers.org, support is implementation-dependent. That is, implementations MAY choose whether and how to support informal or local namespaces.

- Implemantions MUST use CURIE identifiers verbatim and MUST NOT be modified in any way (e.g., case-folding). Implementations MUST NOT expose partial (parsed) identifiers to any client.

Example

Identifiers for GRCh38 chromosome 19:

```
ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl
refseq:NC_000019.10
grch38:19
```

See *Identifier Construction* for examples of CURIE-based identifiers for VR objects.

2.3.2 Residue

Biological definition

A residue refers to a specific monomer within the polymeric chain of a protein or nucleic acid (Source: Wikipedia Residue page).

Computational definition

A character representing a specific residue (i.e., molecular species) or groupings of these ("ambiguity codes"), using one-letter IUPAC abbreviations for nucleic acids and amino acids.

2.3.3 Sequence

Biological definition

A contiguous, linear polymer of nucleic acid or amino acid residues.

Computational definition

A character string of *Residues* that represents a biological sequence using the conventional sequence order (5'-to-3' for nucleic acid sequences, and amino-to-carboxyl for amino acid sequences). IUPAC ambiguity codes are permitted in Sequences.

Information model

A Sequence is a string, constrained to contain only characters representing IUPAC nucleic acid or amino acid codes.

Implementation guidance

- Sequences MAY be empty (zero-length) strings. Empty sequences are used as the replacement Sequence for deletion Alleles.
- Sequences MUST consist of only uppercase IUPAC abbreviations, including ambiguity codes.
- A Sequence provides a stable coordinate system by which an *Allele* MAY be located and interpreted.
- A Sequence MAY have several roles. A "reference sequence" is any Sequence used to define an *Allele*. A Sequence that replaces another Sequence is called a "replacement sequence".
- In some contexts outside VRS, "reference sequence" may refer to a member of set of sequences that comprise a genome assembly. In the VR specification, any sequence may be a "reference sequence", including those in a genome assembly.
- For the purposes of representing sequence variation, it is not necessary that Sequences be explicitly "typed" (i.e., DNA, RNA, or AA).

2.4 Non-variation classes

2.4.1 Interval (Abstract Class)

Biological definition

None.

Computational definition

The *Interval* abstract class defines a range on a *Sequence*, possibly with length zero, and specified using *Interbase Coordinates*. An Interval MAY be a *SimpleInterval* with a single start and end coordinate. *Future Location and Interval types* will enable other methods for describing where *Variation* occurs. Any of these MAY be used as the Interval for Location.

VRS Uses Interbase Coordinates

GA4GH VRS uses interbase coordinates when referring to spans of sequence.

Interbase coordinates refer to the zero-width points before and after *residues*. An interval of interbase coordinates permits referring to any span, including an empty span, before, within, or after a sequence. See *Interbase Coordinates* for more details on this design choice. Interbase coordinates are always zero-based.

SimpleInterval

Computational definition

An *Interval (Abstract Class)* with a single start and end coordinate.

Information model

Field	Type	Limits	Description
type	string	1..1	Interval type; MUST be set to '**SimpleInterval**'
start	uint64	1..1	start position
end	uint64	1..1	end position

Implementation guidance

- Implementations MUST enforce values 0 start end. In the case of double-stranded DNA, this constraint holds even when a feature is on the complementary strand.
- VRS uses Interbase coordinates because they provide conceptual consistency that is not possible with residue-based systems (see *rationale*). Implementations will need to convert between interbase and 1-based inclusive residue coordinates familiar to most human users.
- Interbase coordinates start at 0 (zero).
- The length of an interval is *end - start*.
- An interval in which start == end is a zero width point between two residues.
- An interval of length == 1 MAY be colloquially referred to as a position.
- Two intervals are *equal* if the their start and end coordinates are equal.
- Two intervals *intersect* if the start or end coordinate of one is strictly between the start and end coordinates of the other. That is, if:

  - b.start < a.start < b.end OR
  - b.start < a.end < b.end OR
  - a.start < b.start < a.end OR
  - a.start < b.end < a.end
- Two intervals a and b *coincide* if they intersect or if they are equal (the equality condition is REQUIRED to handle the case of two identical zero-width Intervals).
- <start, end>=*<0,0>* refers to the point with width zero before the first residue.
- <start, end>=*<i,i+1>* refers to the *i+1th* (1-based) residue.
- <start, end>=*<N,N>* refers to the position after the last residue for Sequence of length *N*.
- See example notebooks in GA4GH VR Python Implementation.

Example

```
{
  "end": 44908822,
  "start": 44908821,
  "type": "SimpleInterval"
}
```

2.4.2 Location (Abstract Class)

Biological definition

As used by biologists, the precision of “location” (or “locus”) varies widely, ranging from precise start and end numerical coordinates defining a Location, to bounded regions of a sequence, to conceptual references to named genomic features (e.g., chromosomal bands, genes, exons) as proxies for the Locations on an implied reference sequence.

Computational definition

The *Location* abstract class refers to position of a contiguous segment of a biological sequence. The most common and concrete Location is a *SequenceLocation*, i.e., a Location based on a named sequence and an Interval on that sequence. Additional *Intervals and Locations* may also be conceptual or symbolic locations, such as a cytoband region or a gene. Any of these may be used as the Location for Variation.

Implementation Guidance

- Location refers to a position. Although it MAY imply a sequence, the two concepts are not interchangable, especially when the location is non-specific (e.g., a range) or symbolic (a gene).

SequenceLocation

Biological definition

A specified subsequence within another sequence that is used as a reference sequence.

Computational definition

A Location subclass for describing a defined *Interval (Abstract Class)* on a named *Sequence*.

Information model

Field	Type	Limits	Description
_id	*CURIE*	0..1	Location Id; MUST be unique within document
type	string	1..1	Location type; MUST be set to '**SequenceLocation**'
sequence_id	*CURIE*	1..1	An id mapping to the *Computed Identifiers* of the external database Sequence containing the sequence to be located.
interval	*Interval (Abstract Class)*	1..1	Position of feature on reference sequence specified by sequence_id.

Implementation guidance

- **For a *Sequence* of length *n*:**
  - 0 *interval.start* *interval.end* *n*
  - interbase coordinate 0 refers to the point before the start of the Sequence
  - interbase coordinate n refers to the point after the end of the Sequence.
- Coordinates MUST refer to a valid Sequence. VRS does not support referring to intronic positions within a transcript sequence, extrapolations beyond the ends of sequences, or other implied sequence.

Important: HGVS permits variants that refer to non-existent sequence. Examples include coordinates extrapolated beyond the bounds of a transcript and intronic sequence. Such variants are not representable using VR and MUST be projected to a genomic reference in order to be represented.

Example

```
{
  "interval": {
    "end": 44908822,
    "start": 44908821,
    "type": "SimpleInterval"
  },
  "sequence_id": "ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl",
  "type": "SequenceLocation"
}
```

2.4.3 State (Abstract Class)

Biological definition

None.

Computational definition

State objects are one of two primary components specifying a VR *Allele* (in addition to *Location (Abstract Class)*), and the designated components for representing change (or non-change) of the features indicated by the Allele Location. As an abstract class, State currently encompasses single and contiguous *Sequence* changes (see *SequenceState*), with additional types under consideration (see *State Classes*).

SequenceState

Biological definition

None.

Computational definition

The *SequenceState* class specifically captures a *Sequence* as a *State (Abstract Class)*. This is the State class to use for representing "ref-alt" style variation, including SNVs, MNVs, del, ins, and delins.

Information model

Field	Type	Limits	Description
type	string	1..1	State type; MUST be set to '**SequenceState**'
sequence	string	1..1	The string of sequence residues that is to be used as the state for other types.

Example

```
{
  "sequence": "T",
  "type": "SequenceState"
}
```

2.5 Variation

The Variation class is the conceptual root of all types of variation, both current and future. Variation subclasses are:

Biological definition

In biology, variation is often used to mean genetic variation, describing the differences observed in DNA among individuals.

Computational definition

The *Variation* abstract class is the top-level object in the *Variation Representation Specfication (VRS) Schema Diagram* and represents the concept of a molecular state. The representation and types of molecular states are widely varied, and there are several *Variation Classes* currently under consideration to capture this diversity. The primary Variation subclass defined by the VRS HEAD specification is the *Allele*, with the *Text* subclass for capturing other Variations that are not yet covered.

2.5.1 Allele

Biological definition

One of a number of alternative forms of the same gene or same genetic locus. In the context of biological sequences, "allele" refers to one of a set of specific changes within a *Sequence*. In the context of VR, Allele refers to a Sequence or Sequence change with respect to a reference sequence, without regard to genes or other features.

Computational definition

An Allele is an assertion of the *State* of a biological sequence at a *Location*.

Information model

Field	Type	Limits	Description
_id	*CURIE*	0..1	Variation Id; MUST be unique within document
type	string	1..1	Variation type; MUST be set to '**Allele**'
location	*Location (Abstract Class)*	1..1	Where Allele is located
state	*State (Abstract Class)*	1..1	State at location

Implementation guidance

- The *State* and *Location* subclasses respectively represent diverse kinds of sequence changes and mechanisms for describing the locations of those changes, including varying levels of precision of sequence location and categories of sequence changes.
- Implementations MUST enforce values interval.end sequence_length when the Sequence length is known.
- Alleles are equal only if the component fields are equal: at the same location and with the same state.
- Alleles MAY have multiple related representations on the same Sequence type due to normalization differences.
- Implementations SHOULD normalize Alleles using *"justified" normalization* whenever possible to facilitate comparisons of variation in regions of representational ambiguity.
- Implementations MUST normalize Alleles using *"justified" normalization* when generating a *Computed Identifiers*.
- When the alternate Sequence is the same length as the interval, the lengths of the reference Sequence and imputed Sequence are the same. (Here, imputed sequence means the sequence derived by applying the Allele to the reference sequence.) When the replacement Sequence is shorter than the length of the interval, the imputed Sequence is shorter than the reference Sequence, and conversely for replacements that are larger than the interval.
- When the replacement is "" (the empty string), the Allele refers to a deletion at this location.
- The Allele entity is based on Sequence and is intended to be used for intragenic and extragenic variation. Alleles are not explicitly associated with genes or other features.
- Biologically, referring to Alleles is typically meaningful only in the context of empirical alternatives. For modelling purposes, Alleles MAY exist as a result of biological observation or computational simulation, i.e., virtual Alleles.
- "Single, contiguous" refers the representation of the Allele, not the biological mechanism by which it was created. For instance, two non-adjacent single residue Alleles could be represented by a single contiguous multi-residue Allele.
- The terms "allele" and "variant" are often used interchangeably, although this use may mask subtle distinctions made by some users.
  - In the genetics community, "allele" may also refer to a haplotype.
  - "Allele" connotes a state whereas "variant" connotes a change between states. This distinction makes it awkward to use variant to refer to the concept of an unchanged position in a Sequence and was one of the factors that influenced the preference of "Allele" over "Variant" as the primary subject of annotations.
  - See *Use "Allele" rather than "Variant"* for further details.
- When a trait has a known genetic basis, it is typically represented computationally as an association with an Allele.
- This specification's definition of Allele applies to all Sequence types (DNA, RNA, AA).

Example

```
{
   "location": {
      "interval": {
         "end": 44908822,
         "start": 44908821,
         "type": "SimpleInterval"
      },
      "sequence_id": "ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl",
```

(continues on next page)

(continued from previous page)

```
      "type": "SequenceLocation"
   },
   "state": {
      "sequence": "T",
      "type": "SequenceState"
   },
   "type": "Allele"
}
```

2.5.2 Text

Biological definition

None

Computational definition

The *Text* subclass of *Variation* is intended to capture textual descriptions of variation that cannot be parsed by other Variation subclasses, but are still treated as variation.

Information model

Field	Type	Limits	Description
_id	*CURIE*	0..1	Variation Id; MUST be unique within document
type	string	1..1	Variation type; MUST be set to '**Text**'
definition	string	1..1	The textual variation representation not parsable by other subclasses of Variation.

Implementation guidance

- An implementation MUST represent Variation with subclasses other than Text if possible.
- An implementation SHOULD define or adopt conventions for defining the strings stored in Text.definition.
- If a future version of VRS is adopted by an implementation and the new version enables defining existing Text objects under a different Variation subclass, the implementation MUST construct a new object under the other Variation subclass. In such a case, an implementation SHOULD persist the original Text object and respond to queries matching the Text object with the new object.
- Additional Variation subclasses are continually under consideration. Please open a GitHub issue if you would like to propose a Variation subclass to cover a needed variation representation.

Example

```
{
  "definition": "APOE loss",
  "type": "Text"
}
```

2.5.3 VariationSet

Biological definition

Sets of variation are used widely, such as sets of variants in dbSNP or ClinVar that might be related by function.

Computational definition

An unconstrained set of Variation objects or references.

Information model

Field	Type	Limits	Description
_id	*CURIE*	0..1	Identifier of the VariationSet.
type	string	1..1	Variation type; MUST be "VariationSet" (default)
members	*Variation*[] or *CURIE*[]	0..*	List of Variation objects or identifiers. Attribute is required, but MAY be empty.

Implementation Guidance

- The VariationSet identifier MAY be computed as described in *Computed Identifiers*, in which case the identifier effectively refers to a static set because a different set of members would generate a different identifier.
- *members* may be specified as Variation objects or CURIE identifiers.
- CURIEs MAY refer to entities outside the *ga4gh* namespace. However, objects that use non-*ga4gh* identifiers MAY NOT use the *Computed Identifiers* mechanism.
- VariationSet identifiers computed using the GA4GH *Computed Identifiers* process do *not* depend on whether the Variation objects are inlined or referenced, and do *not* depend on the order of members.
- Elements of *members* must be subclasses of Variation, which permits sets to be nested.
- Recursive sets are not meaningful and are not supported.
- VariationSets may be empty.

Example

Inlined Variation objects:

```
{
  "members": [
    {
      "location": {
        "interval": {
          "end": 11,
          "start": 10,
          "type": "SimpleInterval"
        },
        "sequence_id": "ga4gh:SQ.01234abcde",
        "type": "SequenceLocation"
      },
      "state": {
        "sequence": "C",
        "type": "SequenceState"
      },
      "type": "Allele"
    },
    {
      "location": {
        "interval": {
          "end": 21,
          "start": 20,
          "type": "SimpleInterval"
        },
        "sequence_id": "ga4gh:SQ.01234abcde",
```

(continues on next page)

(continued from previous page)

```
        "type": "SequenceLocation"
      },
      "state": {
        "sequence": "C",
        "type": "SequenceState"
      },
      "type": "Allele"
    },
    {
      "location": {
        "interval": {
          "end": 31,
          "start": 30,
          "type": "SimpleInterval"
        },
        "sequence_id": "ga4gh:SQ.01234abcde",
        "type": "SequenceLocation"
      },
      "state": {
        "sequence": "C",
        "type": "SequenceState"
      },
      "type": "Allele"
    }
  ],
  "type": "VariationSet"
}
```

Referenced Variation objects:

```
{
  "members": [
    "ga4gh:VA.6xjH0Ikz88s7MhcyN5GJTa1p712-M10W",
    "ga4gh:VA.7k2lyIsIsoBgRFPlfnIOeCeEgj_2BO7F",
    "ga4gh:VA.ikcK330gH3bYO2sw9QcTsoptTFnk_Xjh"
  ],
  "type": "VariationSet"
}
```

The GA4GH computed identifier for these sets is *ga4gh:VS.WVC_R7OJ688EQX3NrgpJfsf_ctQUsVP3*, regardless of the whether the Variation objects are inlined or referenced, and regardless of order. See *Computed Identifiers* for more information.

CHAPTER 3

Schema

3.1 Overview

3.2 Machine Readable Specifications

The machine readable VRS Specification is written using JSON Schema.

The schema itself is written in YAML (`vr.yaml`) and converted to JSON (`vr.json`).

Contributions to the schema MUST be written in the YAML document.

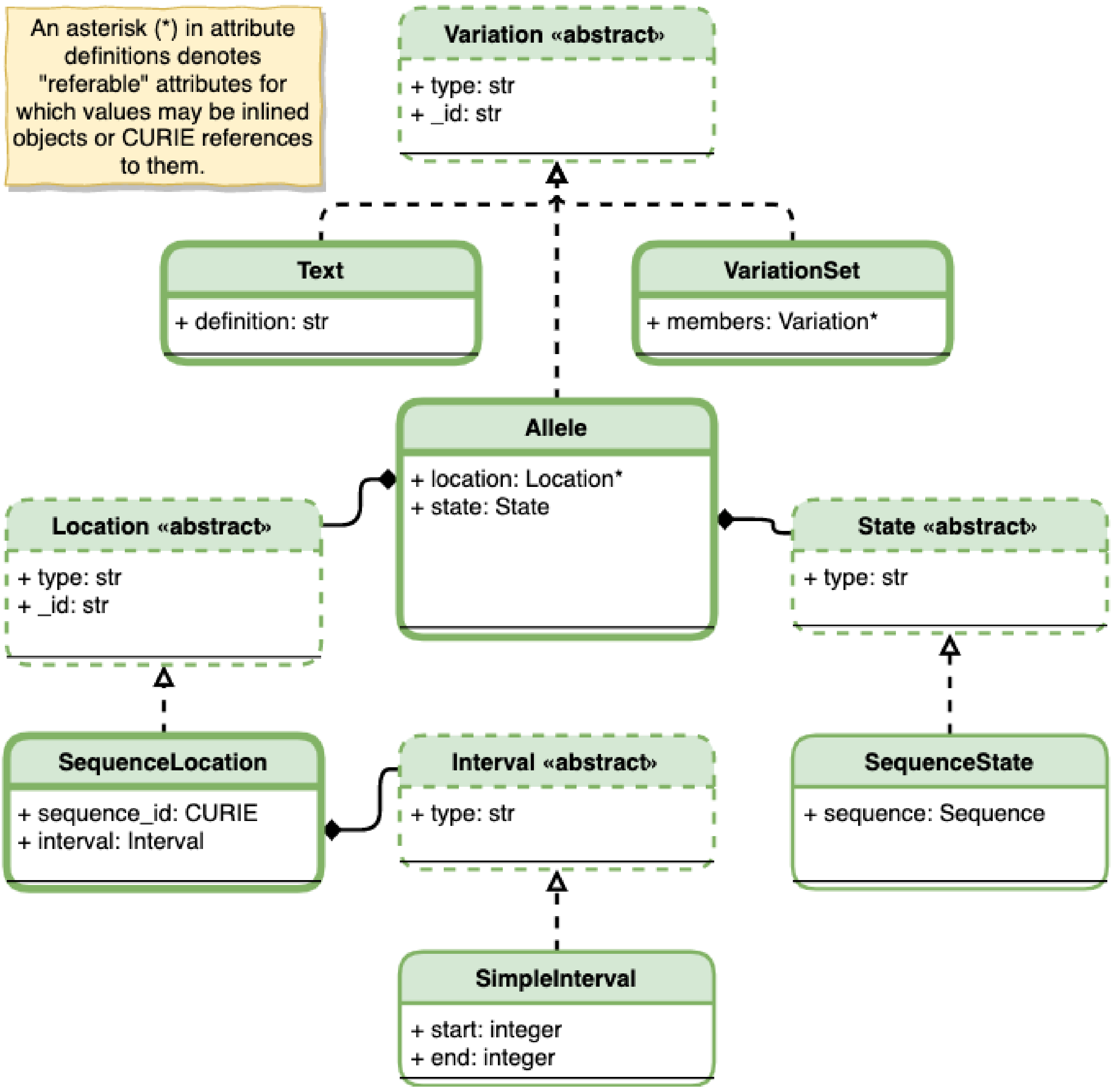

Fig. 1: Variation Representation Specfication (VRS) Schema Diagram
VRS describes several classes for representing biological sequence variation. Classes are shown as boxes. Inheritance and composition are shown with lines connecting classes. Dashed borders denote abstract classes, which are not instantiated. Four abstract classes – *Variation*, *Location (Abstract Class)*, *State (Abstract Class)*, and *Interval (Abstract Class)* – enable specializations of concepts in this and future versions of VRS. Bold borders denote identifiable classes – that may be referenced with an identifier. Identifiable objects have an optional *_id* attribute. Thin solid borders denote classes that not identifiable; these classes exist only to structure data within identifiable classes. All classes have a string *type*. Dashed arrow lines denote inheritance; subclasses inherit all attributes from their parent. Inherited attributes are not shown in this diagram. Solid lines with diamonds denote composition of one class with objects from another. An asterisk on a class attribute definition denotes an attribute that may contain either the object or a CURIE identifier to that object.

CHAPTER 4

Implementation Guide

This section describes the data and algorithmic components that are REQUIRED for implementations of VRS.

- *Required External Data*: All implementations will require access to sequences and sequence accessions. The Required External Data section provides guidance on the abstract functionality that is required in order to implement VR.
- *Normalization*: Expands Alleles to the maximal region of representational ambiguity.
- *Computed Identifiers*: Generate globally unique identifiers based solely on the variation definition.

4.1 Required External Data

All VRS implementations will require external data regarding sequences and sequence metadata. The choices of data sources and access methods are left to implementations. This section provides guidance about how to implement required data and helps implementers estimate effort. This section is descriptive only: it is not intended to impose requirements on interface to, or sources of, external data. For clarity and completeness, this section also describes the contexts in which external data are used.

4.1.1 Contexts

- **Conversion from other variant formats** When converting from other variation formats, implementations MUST translate primary database accessions or identifiers (*e.g.,* `NM_000551.3` or `refseq:NM_000551.3`) to a GA4GH VRS sequence identifier (`ga4gh:SQ.v_QTc1p-MUYdgrRv4LMT6ByXIOsdw3C_`)
- **Conversion to other variant formats** When converting to other variation formats, implementations SHOULD translate GA4GH VR sequence identifier (`ga4gh:SQ.v_QTc1p-MUYdgrRv4LMT6ByXIOsdw3C_`) to primary database identifiers (`refseq:NM_000551.3`) that will be more readily recognized by users.
- **Normalization** During *Normalization*, implementations will need access to sequence length and sequence contexts.

4.1.2 Data Services

The following tables summarizes data required in the above contexts:

Table 1: Data Service Desciptions

Data Service	Description	Contexts
sequence	For a given sequence identifier and range, return the corresponding subsequence.	normalization
sequence length	For a given sequence identifier, return the length of the sequence	normalization
identifier translation	For a given sequence identifier and target namespace, return all identifiers in the target namespace that are equivelent to the given identifier.	Conversion to/from other formats

Note: Construction of the GA4GH computed identifier for a sequence is described in *Computed Identifiers*.

4.1.3 Suggested Implementation

In order to maximize portability and to insulate implementations from decisions about external data sources, implementers should consider writing an abstract data proxy interface that to define a service, and then implement this interface for each data backend to be supported. The *vr-python: GA4GH VRS Python Implementation* DataProxy class provides an example of this design pattern and sample replies.

The DataProxy interface defines three methods:

- `get_sequence(identifier, start, end)`: Given a sequence identifier and start and end coordinates, return the corresponding sequence segment.
- `get_metadata(identifier)`: Given a sequence identifier, return a dictionary of length, alphabet, and known aliases.
- `translate_sequence_identifier(identifier, namespace)`: Given a sequence identifier, return all aliases in the specified namespace. Zero or more aliases may be returned.

GA4GH VR Python Implementation implements the DataProxy interface using a local SeqRepo instance backend and using a SeqRepo REST Service backend. A GA4GH refget implementation has been started, but is pending interface changes to support lookup using primary database accesssions.

Examples

The following examples are taken from VR Python Notebooks:

```python
from ga4gh.vr.extras.dataproxy import SeqRepoRESTDataProxy
seqrepo_rest_service_url = "http://localhost:5000/seqrepo"
dp = SeqRepoRESTDataProxy(base_url=seqrepo_rest_service_url)

def get_sequence(identifier, start=None, end=None):
    """returns sequence for given identifier, optionally limited
    to interbase <start, end> interval"""
    return dp.get_sequence(identifier, start, end)
def get_sequence_length(identifier):
```

(continues on next page)

(continued from previous page)

```
    """return length of given sequence identifier"""
    return dp.get_metadata(identifier)["length"]
def translate_sequence_identifier(identifier, namespace):
    """return for given identifier, return *list* of equivalent identifiers in given␣
↪namespace"""
    return dp.translate_sequence_identifier(identifier, namespace)
```

```
get_sequence_length("ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl")
58617616
```

```
start, end = 44908821-25, 44908822+25
get_sequence("ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl", start, end)
'CCGCGATGCCGATGACCTGCAGAAGCGCCTGGCAGTGTACCAGGCCGGGGC'
```

```
translate_sequence_identifier("GRCh38:19", "ga4gh")
['ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl']
```

```
translate_sequence_identifier("ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl", "GRCh38")
['GRCh38:19', 'GRCh38:chr19']
```

4.2 Normalization

Certain insertion or deletion alleles may be represented ambiguously when using conventional sequence normalization, resulting in significant challenges when comparing such alleles.

VRS describes a “fully-justified” normalization algorithm inspired by NCBI’s Variant Overprecision Correction Algorithm[1]. Fully-justified normalization expands such ambiguous representation over the entire region of ambiguity, resulting in an *unambiguous* representation that may be readily compared with other alleles.

VRS RECOMMENDS that Alleles at precise locations are normalized to a fully justified form unless there is a compelling reason to do otherwise.

The process for fully justifying two alleles (reference sequence and alternate sequence) at an interval is outlined below.

1. Trim sequences:
    - Remove suffixes common to all alleles, if any. Decrement the interval end position by the length of the trimmed suffix.
    - Remove prefixes common to all alleles, if any. Increment the interval start position by the length of the trimmed prefix.
    - If neither allele is empty, the allele pairs represent a alleles that do not have common prefixes or suffixes. Normalization is not applicable and the trimmed alleles are returned.
2. Determine bounds of ambiguity:
    - Left roll: While the terminal base of all non-empty alleles is equal to the base *prior* to the current position, circularly permute all alleles *rightward* and move the current position *leftward*. When terminating, return *left_roll*, the number of steps rolled leftward.
    - Right roll: Symmetric case of left roll, returning *right_roll*, the number of steps rolled rightward.
3. Update position and alleles:

[1] Holmes, J. B., Moyer, E., Phan, L., Maglott, D. & Kattman, B. L. *SPDI: Data Model for Variants and Applications at NCBI.* Bioinformatics (2020 March 15). doi:10.1093/bioinformatics/btz856

- To each trimmed allele, prepend the *left_roll* bases prior to the trimmed allele position and append the *right_roll* bases after the trimmed allele position.
- Expand the trimmed allele position by decrementing the start by *left_roll* and incrementing the end by *right_roll*.

Table 2: **VRS Justified Normalization** A demonstration of fully justifying an insertion allele.

Steps	Interbase Position and Alleles	Resulting Allele Set (All alleles in this column result in the same empirical sequence change.)
0. Given allele `S:g.5_6delinsCAGCA` defined on reference sequence S=TCAGCAGCT	(4,6) (“CA”, “CAGCA”)	$TCAG\left[\frac{CA}{CAGCA}\right]GCT$
1. Trimming Remove prefix common to all alleles, if any, and update start position. Remove suffix common to all alleles, if any, and update end position. **Note:** This example shows removing C prefix and A suffix. Equivalently in this case, CA prefix or CA suffix could be removed.	(5,5) (“”, “AGC”)	$TCAGC\left[\frac{}{AGC}\right]AGCT$
2. Condition: One allele must be empty. If the reference allele is empty, the allele set represents an insertion in the reference. If the alternate allele is empty, the allele set represents a deletion in the reference. If neither is true, the allele set represents a substitution, which is not subject to further normalization.		
3. Roll Left Begin with trimmed alleles . While the terminal base of all non-empty alleles equals the base prior to the current position, circularly permute all alleles right one step and move the start left one position. Shown: The 4 incremental steps of rolling left.	(1,1) (“”, “CAG”)	$TCAGC\left[\frac{}{AGC}\right]AGCT$ $TCAG\left[\frac{}{CAG}\right]CAGCT$ $TCA\left[\frac{}{GCA}\right]GCAGCT$ $TC\left[\frac{}{AGC}\right]AGCAGCT$ $T\left[\frac{}{CAG}\right]CAGCAGCT$ $\Rightarrow left\_roll = 4$
4. Roll Right Symmetric case of step 3.	(8,8) (“”, “AGC”)	$TCAGC\left[\frac{}{AGC}\right]AGCT$ $TCAGCA\left[\frac{}{GCA}\right]GCT$ $TCAGCAG\left[\frac{}{CAG}\right]CT$ $TCAGCAGC\left[\frac{}{AGC}\right]T$ $\Rightarrow right\_roll = 3$
5. Update position and alleles to fully justify within region of ambiguity.	(1,8)	$TCAGC\Big[\Big]AGCT$

References

4.3 Computed Identifiers

VRS provides an algorithmic solution to deterministically generate a globally unique identifier from a VRS object itself. All valid implementations of the VRS Computed Identifier will generate the same identifier when the objects are identical, and will generate different identifiers when they are not. The VRS Computed Digest algorithm obviates centralized registration services, allows computational pipelines to generate "private" ids efficiently, and makes it easier for distributed groups to share data.

A VRS Computed Identifier for a VRS concept is computed as follows:

- If the object is an *Allele*, *normalize* it.
- Generate binary data to digest. If the object is a *Sequence* string, encode it using UTF-8. Otherwise, serialize the object using *Digest Serialization*.
- *Generate a truncated digest* from the binary data.
- *Construct an identifier* based on the digest and object type.

The following diagram depicts the operations necessary to generate a computed identifier. These operations are described in detail in the subsequent sections.

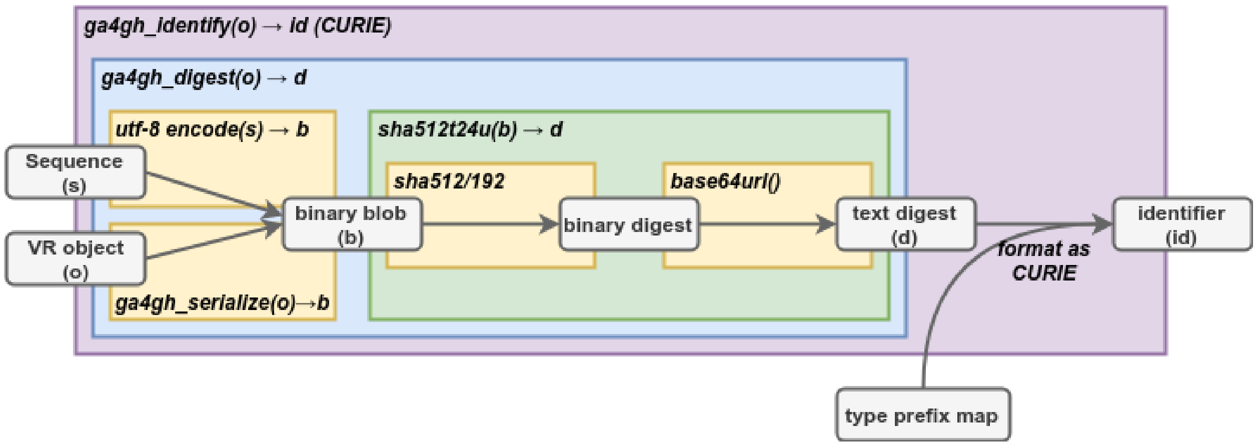

Fig. 1: Serialization, Digest, and Computed Identifier Operations
Entities are shown in gray boxes. Functions are denoted by bold italics. The yellow, green, and blue boxes, corresponding to the `sha512t24u`, `ga4gh_digest`, and `ga4gh_identify` functions respectively, depict the dependencies among functions. `SHA512/192` is SHA-512 truncated at 192 bits using the systematic name recommended by SHA-512 (§5.3.6). base64url is the official name of the variant of Base64 encoding that uses a URL-safe character set. [figure source]

Note: Most implementation users will need only the `ga4gh_identify` function. We describe the `ga4gh_serialize`, `ga4gh_digest`, and `sha512t24u` functions here primarily for implementers.

4.3.1 Requirements

Implementations MUST adhere to the following requirements:

- Implementations MUST use the normalization, serialization, and digest mechanisms described in this section when generating GA4GH Computed Identifiers. Implementations MUST NOT use any other normalization, serialization, or digest mechanism to generate a GA4GH Computed Identifier.
- Implementations MUST ensure that all nested objects are identified with GA4GH Computed Identifiers. Implementations MAY NOT reference nested objects using identifiers in any namespace other than `ga4gh`.

Note: The GA4GH schema MAY be used with identifiers from any namespace. For example, a SequenceLocation may be defined using a *sequence_id* = `refseq:NC_000019.10`. However, an implementation of the Computed Identifier algorithm MUST first translate sequence accessions to GA4GH `SQ` accessions to be compliant with this specification.

4.3.2 Digest Serialization

Digest serialization converts a VRS object into a binary representation in preparation for computing a digest of the object. The Digest Serialization specification ensures that all implementations serialize variation objects identically, and therefore that the digests will also be identical. VRS provides validation tests to ensure compliance.

Important: Do not confuse Digest Serialization with JSON serialization or other serialization forms. Although Digest Serialization and JSON serialization appear similar, they are NOT interchangeable and will generate different GA4GH Digests.

Although several proposals exist for serializing arbitrary data in a consistent manner ([Gibson], [OLPC], [JCS]), none have been ratified. As a result, VRS defines a custom serialization format that is consistent with these proposals but does not rely on them for definition; it is hoped that a future ratified standard will be forward compatible with the process described here.

The first step in serialization is to generate message content. If the object is a string representing a *Sequence*, the serialization is the UTF-8 encoding of the string. Because this is a common operation, implementations are strongly encouraged to precompute GA4GH sequence identifiers as described in *Required External Data*.

If the object is a composite VRS object, implementations MUST:

- ensure that objects are referenced with identifiers in the `ga4gh` namespace
- replace nested identifiable objects (i.e., objects that have id properties) with their corresponding *digests*
- order arrays of digests and ids by Unicode Character Set values
- filter out fields that start with underscore (e.g., *_id*)
- filter out fields with null values

The second step is to JSON serialize the message content with the following REQUIRED constraints:

- encode the serialization in UTF-8
- exclude insignificant whitespace, as defined in RFC8259§2
- order all keys by Unicode Character Set values
- use two-char escape codes when available, as defined in RFC8259§7

The criteria for the digest serialization method was that it must be relatively easy and reliable to implement in any common computer language.

Example

```
allele = models.Allele(location=models.SequenceLocation(
    sequence_id="ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl",
    interval=simple_interval),
    state=models.SequenceState(sequence="T"))
ga4gh_serialize(allele)
```

Gives the following *binary* (UTF-8 encoded) data:

```
{"location":"u5fspwVbQ79QkX6GHLF8tXPCAXFJqRPx","state":{"sequence":"T","type":
↪"SequenceState"},"type":"Allele"}
```

For comparison, here is one of many possible JSON serializations of the same object:

```
allele.for_json()
```

```
{
  "location": {
    "interval": {
      "end": 44908822,
      "start": 44908821,
      "type": "SimpleInterval"
    },
    "sequence_id": "ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl",
    "type": "SequenceLocation"
  },
  "state": {
    "sequence": "T",
    "type": "SequenceState"
  },
  "type": "Allele"
}
```

4.3.3 Truncated Digest (sha512t24u)

The sha512t24u truncated digest algorithm computes an ASCII digest from binary data. The method uses two well-established standard algorithms, the SHA-512 hash function, which generates a binary digest from binary data, and Base64 URL encoding, which encodes binary data using printable characters.

Computing the sha512t24u truncated digest for binary data consists of three steps:

1. Compute the SHA-512 digest of a binary data.
2. Truncate the digest to the left-most 24 bytes (192 bits). See *Truncated Digest Timing and Collision Analysis* for the rationale for 24 bytes.
3. Encode the truncated digest as a base64url ASCII string.

```
>>> import base64, hashlib
>>> def sha512t24u(blob):
        digest = hashlib.sha512(blob).digest()
        tdigest = digest[:24]
        tdigest_b64u = base64.urlsafe_b64encode(tdigest).decode("ASCII")
        return tdigest_b64u
>>> sha512t24u(b"ACGT")
'aKF498dAxcJAqme6QYQ7EZ07-fiw8Kw2'
```

4.3.4 Identifier Construction

The final step of generating a computed identifier for a VRS object is to generate a W3C CURIE formatted identifier, which has the form:

```
prefix ":" reference
```

The GA4GH VRS constructs computed identifiers as follows:

```
"ga4gh" ":" type_prefix "." <digest>
```

Warning: Do not confuse the W3C CURIE `prefix` ("ga4gh") with the type prefix.

Type prefixes used by VRS are:

type_prefix	VRS class name
SQ	Sequence
VA	Allele
VS	VariationSet
VSL	Sequence Location
VT	Text

For example, the identifer for the allele example under *Digest Serialization* gives:

`ga4gh:VA.EgHPXXhULTwoP4-ACfs-YCXaeUQJBjH_`

4.3.5 References

4.4 Example

This section provides a complete, language-neutral example of essential features of VR. In this example, we will translate an HGVS-formatted variant, `NC_000013.11:g.32936732G>C`, into its VR format and assign a globally unique identifier.

4.4.1 Translate HGVS to VR

Polymorphism in VR

VRS uses polymorphism extensively in order to provide a coherent top-down structure for variation while enabling precise models for variation data.

For example, Allele is a kind of Variation, SequenceLocation is a kind of Location, and SequenceState is a kind of State. See *Future Plans* for the roadmap of VRS data classes and relationships.

All VRS objects contain a `type` attribute, which is used to discriminate polymorphic objects.

The hgvs string `NC_000013.11:g.32936732G>C` represents a single base substitution on the reference sequence NC_000013.11 (human chromosome 13, assembly GRCh38) at position 32936732 from the reference nucleotide `G` to `C`.

In VR, a contiguous change is represented using an *Allele* object, which is composed of a *Location* and of the *State* at that location. Location and State are abstract concepts: VRS is designed to accommodate many kinds of Locations based on sequence position, gene names, cytogentic bands, or other ways of describing locations. Similarly, State may refer to a specific sequence change, copy number change, or complex sequence event.

In this example, we will use a *SequenceLocation*, which is composed of a sequence identifier and a *SimpleInterval*.

In VR, all identifiers are a Compact URI (CURIE). Therefore, NC_000013.11 MUST be written as the string `refseq:NC_000013.11` to make explicit that this sequence is from RefSeq . VRS does not restrict which data sources may be used, but does recommend using prefixes from identifiers.org.

VRS uses *Interbase Coordinates*. Interbase coordinates *always* use intervals to refer to sequence spans. For the purposes of this example, interbase coordinates *look* like the more familiar 0-based, right-open numbering system. (Please read about *Interbase Coordinates* if you are interested in the significant advantages of this design choice over other coordinate systems.)

The *SimpleInterval* for the position `32936732` is

```
{
  "end": 32936732,
  "start": 32936731,
  "type": "SimpleInterval"
}
```

The interval is then 'placed' on a sequence to create the *SequenceLocation*:

```
{
  "interval": {
    "end": 32936732,
    "start": 32936731,
    "type": "SimpleInterval"
  },
  "sequence_id": "refseq:NC_000013.11",
  "type": "SequenceLocation"
}
```

A *SequenceState* objects consists simply of the replacement sequence, as follows:

```
{
  "sequence": "C",
  "type": "SequenceState"
}
```

We are now in a position to construct an *Allele* object using the objects defined above:

```
{
  "location": {
    "interval": {
      "end": 32936732,
      "start": 32936731,
      "type": "SimpleInterval"
    },
    "sequence_id": "refseq:NC_000013.11",
    "type": "SequenceLocation"
  },
  "state": {
    "sequence": "C",
    "type": "SequenceState"
  },
```

(continues on next page)

(continued from previous page)

```
  "type": "Allele"
}
```

This Allele is a fully-compliant VRS object that is parsable using the VRS JSON Schema.

Note: VRS is verbose! The goal of VRS is to provide a extensible framework for representation of sequence variation in computers. VRS objects are readily parsable and have precise meaning, but are often larger than other representations and are typically less readable by humans. This tradeoff is intentional!

4.4.2 Generate a computed identifer

A key feature of VRS is an easily-implemented algorithm to generate computed, digest-based identifiers for variation objects. This algorithm permits organizations to generate the same identifier for the same allele without prior coordination, which in turn facilitates sharing, obviates centralized registration services, and enables identifiers to be used in secure settings (such as diagnostic labs).

Generating a computed identifier requires that all nested objects also use computed identifiers. In this example, the sequence identifier MUST be transformed into a digest-based identifer as described in *Computed Identifiers*. In practice, implmentations SHOULD precompute sequence digests or SHOULD use an existing service that does so. (See *Required External Data* for a description of data that are needed to implement VR.) In this case, `refseq:NC_000013.11` maps to `ga4gh:SQ._0wi-qoDrvram155UmcSC-zA5ZK4fpLT`. All VRS computed identifiers begin with the `ga4gh` prefix and use a type prefix (`SQ`, here) to denote the type of object. The VRS sequence identifier is then substituted directly into the Allele's location object:

```
{
  "location": {
    "interval": {
      "end": 32936732,
      "start": 32936731,
      "type": "SimpleInterval"
    },
    "sequence_id": "ga4gh:SQ._0wi-qoDrvram155UmcSC-zA5ZK4fpLT",
    "type": "SequenceLocation"
  },
  "state": {
    "sequence": "C",
    "type": "SequenceState"
  },
  "type": "Allele"
}
```

This, too, is a valid VRS Allele.

Note: Using VRS sequence identifiers collapses differences between alleles due to trivial differences in reference naming. The same variation reported on NC_000013.11, CM000675.2, GRCh38:13, GRCh38.p13:13 would appear to be distinct variation; using a digest identifer will ensure that variation is reported on a single sequence identifier. Furthermore, using digest-based sequence identifiers enables the use of custom reference sequences.

The first step in constructing a computed identifier is to create a binary digest serialization of the Allele. Details are provided in *Computed Identifiers*. For this example the binary object looks like:

```
'{"location":"v9K0mcjQVugxTDIcdi7GBJ_R6fZ1lsYq","state":{"sequence":"C","type":
↪"SequenceState"},"type":"Allele"}'
(UTF-8 encoded)
```

Important: The binary serialization is governed by constraints that guarantee that different implementations will generate the same binary "blob". Do not confuse binary digest serialization with JSON serialization, which is used elsewhere with VRS schema.

The GA4GH digest for the above blob is computed using the first 192 bits (24 bytes) of the SHA-512 digest, base64url encoded. Conceptually, the function is:

```
base64url( sha512( blob )[:24] )
```

In this example, the value returned is `n9ax-9x6gOC0OEt73VMYqCBfqfxG1XUH`.

A GA4GH Computed Identifier has the form:

```
"ga4gh" ":" <type_prefix> "." <digest>
```

The `type_prefix` for a VRS Allele is `VA`, which results in the following computed identifier for our example:

```
ga4gh:VA.n9ax-9x6gOC0OEt73VMYqCBfqfxG1XUH
```

Variation and Location objects contain an OPTIONAL `_id` attribute which implementations may use to store any CURIE-formatted identifier. *If* an implementation returns a computed identifier with objects, the object might look like the following:

```
{
  "_id": "ga4gh:VA.n9ax-9x6gOC0OEt73VMYqCBfqfxG1XUH",
  "location": {
    "interval": {
      "end": 32936732,
      "start": 32936731,
      "type": "SimpleInterval"
    },
    "sequence_id": "ga4gh:SQ._0wi-qoDrvram155UmcSC-zA5ZK4fpLT",
    "type": "SequenceLocation"
  },
  "state": {
    "sequence": "C",
    "type": "SequenceState"
  },
  "type": "Allele"
}
```

This example provides a full VR-compliant Allele with a computed identifier.

Note: The `_id` attribute is optional. If it is used, the value MUST be a CURIE, but it does NOT need to be a GA4GH Computed Identifier. Applications MAY choose to implement their own identifier scheme for private or public use. For example, the above `_id` could be a serial number assigned by an application, such as `acmecorp:v0000123`.

4.4.3 What's Next?

This example has shown a full example for a relatively simple case. VRS provides a framework that will enable much more complex variation. Please see *Future Plans* for a discussion of variation classes that are intened in the near future.

The *Implementations* section lists libraries and packages that implement VRS.

VRS objects are value objects. An important consequence of this design choice is that data should be associated *with* VRS objects via their identifiers rather than embedded *within* those objects. The appendix contains an example of *associating annotations with variation*.

CHAPTER 5

Appendices

5.1 Relationship of VRS to existing standards

Because a primary objective of the GA4GH Variation Representation Specification (VRS) effort is to unify disparate efforts to represent biological sequence variation, it is important to describe how this document relates to previous work in order to avoid “reinventing the wheel”.

The Variant Call Format (VCF) is the de facto standard for representing alleles, particularly for use during primary analysis in high-throughput sequencing pipelines. VCF permits a wide range of annotations on alleles, such as quality and likelihood scores. VCF is a file-based format and is exclusively for genomic alleles. In contrast, the VRS data model abstractly represents Alleles, Haplotypes, and Genotypes on all sequence types, is independent of medium, and is well-suited to secondary analyses, allele interpretation, aggregation, and system-level interoperability.

The HGVS nomenclature recommendations describe how sequence variation should be presented to human beings. In addition to representing a wide variety of sequence changes from single residue variation through large cytogenetic events, HGVS attempts to also encode in strings notions of biological mechanism (e.g., inversion as a kind of deletion-insertion event), predicted events (e.g., parentheses for computing protein sequence), and complex states (e.g., mosaicism). In practice, HGVS recommendations are difficult to implement fully and consistently, leading to ambiguity in presentation. In contrast, the VRS is a formal specification that improves consistency of representation among computer systems. VRS is currently less expressive than HGVS for rarer cases of variation, such as cytogenetic variation or context-based allele representations (e.g., insT written as dupT when the insertion follows a T). Future versions of the specification will seek to address limitations while preserving principles of conceptual clarity and precision.

The Sequence Ontology (SO) is a set of terms and relationships used to describe the features and attributes of biological sequence. The focus of the SO has been the annotation of, or placement of ‘meaning’, onto genomic sequence regions. The VRS effort seeks to use the same descriptive definitions where possible, and to inform the refinement of SO.

The Genotype Ontology (GENO) builds on the SO to include richer modeling of genetic variation at different levels of granularity that are captured in genotype representations. Unlike the SO which is used primarily for annotation of genomic features, GENO was developed by the Monarch Initiative to support semantic data models for integrated representation of genotypes and genetic variants described in human and model organism databases. The core of the GENO model decomposes a genotype specifying sequence variation across an entire genome into smaller components of variation (e.g. allelic composition at a particular locus, haplotypes, gene alleles, and specific sequence alterations).

GENO also enables description of biological attributes of these genetic entities (e.g. zygosity, phase, copy number, parental origin, genomic position), and their causal relationships with phenotypes and diseases.

ClinVar is an archive of clinically reported relationships between variation and phenotypes along with interpretations and supporting evidence. Data in ClinVar are submitted primarily by diagnostic labs. ClinVar includes expert reviews and data links to other clinically-relevant resources at NCBI. The VRS specification is expected to facilitate data submissions by providing unified guidelines for data structure and allele normalization.

ClinGen provides a centralized database of genomic and phenotypic data provided by clinicians, researchers, and patients. It standardizes clinical annotation and interpretation of genomic variants and provides evidence-based expert consensus for curated genes and variants. ClinGen has informed the VRS effort and is committed to harmonizing and collaborating on the evolution of the VRS specification to achieve improved data sharing.

HL7 FHIR Genomics, Version 2 Clinical Genomics Implementation Guide, CDA Genetic Test Report: There are several standards developed under the HL7 umbrella that include a genomics component. The FHIR Genomics component was released as part of the overall FHIR specification (latest is Release 3) based on standardized use cases. The HL7 Clinical Genomics (CG) Work Group focuses on developing standards for clinical genomic data and related relevant information within EHRs. The specifications developed by the CG work group primarily utilize the HL7 v2 messaging standard and the newer HL7 FHIR (Fast Healthcare Interoperability Resources) framework.

The SPDI format created to represent alleles in NCBI's Variation Services has four components: the sequence identifier, which is specified with a sequence accession and version; the 0-based interbase coordinate where the deletion starts; the deleted sequence (or its length) and the inserted sequence. The Variation Services return the minimum deleted sequence required to avoid over precision. For example, a deletion of one G in a run of 4 is specified with deleted and inserted sequences of GGGG and GGG respectively, avoiding the need to left or right shift the minimal representation. This reduces ambiguity, but can lead to long allele descriptions.

5.2 Associating Annotions with VRS Objects

This example demonstrates how to associate information with VR objects. Although the examples use the GA4GH VR Python Implementation library, the principles apply regardless of implementation.

Information is never embedded within VRS objects. Instead, it is associated with those objects by means of their ids. This approach to annotations scales better in size and distributes better across multiple data sources.

```
import collections
from ga4gh.vr import ga4gh_identify, models
from ga4gh.vr.extras.dataproxy import SeqRepoRESTDataProxy
from ga4gh.vr.extras.translator import Translator

# Requires seqrepo REST interface is running on this URL (e.g., using docker image)
seqrepo_rest_service_url = "http://localhost:5000/seqrepo"
dp = SeqRepoRESTDataProxy(base_url=seqrepo_rest_service_url)

tlr = Translator(data_proxy=dp)
```

```
# Declare some data as human-readable RS id labels with HGVS expressions
data = (
    ("rs7412C",   "NC_000019.10:g.44908822="),
    ("rs7412T",   "NC_000019.10:g.44908822C>T"),
    ("rs429358C", "NC_000019.10:g.44908684="),
    ("rs429358T", "NC_000019.10:g.44908684T>C")
)
```

```
# Parse the HGVS expressions and generate three dicts:
# alleles[allele_id]  allele object
# rs_names[allele_id]  rs label
# hgvs_name[allele_id]  original hgvs expression

# For convenience, also build
# rs_to_id[rs_name]  allele_id

alleles = {}
rs_names = {}
hgvs_names = collections.defaultdict(lambda: dict())
for rs, hgvs_expr in data:
    allele = tlr.from_hgvs(hgvs_expr)
    allele_id = ga4gh_identify(allele)
    alleles[allele_id] = allele
    hgvs_names[allele_id] = hgvs_expr
    rs_names[allele_id] = rs

rs_to_id = {r: i for i, r in rs_names.items()}
```

```
# Now, build a new set of annotations: allele frequencies
# This is more complicated because it maps to a map of frequences
# It should be clear that other frequencies could be easily added here
# or as a separate data source
freqs = {
    "gnomad": {
        "global": {
            rs_to_id["rs7412C"]: 0.9385,
            rs_to_id["rs7412T"]: 0.0615,
            rs_to_id["rs429358C"]: 0.1385,
            rs_to_id["rs429358T"]: 0.8615,
        }
    }
}
```

```
# It might be convenient to save these data
# A saved document might have structure like this:
doc = {
    "alleles": alleles,
    "hgvs_names": hgvs_names,
    "rs_names": rs_names,
    "freqs": freqs
}
```

```
# For the benefit of pretty printing, let's replace the allele objects with their␣
↪dict representations
doc["alleles"] = {i: a.as_dict() for i, a in doc["alleles"].items()}
import json
print(json.dumps(doc, indent=2))
```

```
{
  "alleles": {
    "ga4gh:VA.UUvQpMYU5x8XXBS-RhBhmipTWe2AALzj": {
      "location": {
        "interval": {
          "end": 44908822,
          "start": 44908821,
          "type": "SimpleInterval"
        },
        "sequence_id": "ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl",
        "type": "SequenceLocation"
      },
      "state": {
        "sequence": "C",
        "type": "SequenceState"
      },
      "type": "Allele"
    },
    "ga4gh:VA.EgHPXXhULTwoP4-ACfs-YCXaeUQJBjH_": {
      "location": {
        "interval": {
          "end": 44908822,
          "start": 44908821,
          "type": "SimpleInterval"
        },
        "sequence_id": "ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl",
        "type": "SequenceLocation"
      },
      "state": {
        "sequence": "T",
        "type": "SequenceState"
      },
      "type": "Allele"
    },
    "ga4gh:VA.LQrGFIOAP8wEAybwNBo8pJ3yIG7tXWoh": {
      "location": {
        "interval": {
          "end": 44908684,
          "start": 44908683,
          "type": "SimpleInterval"
        },
        "sequence_id": "ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl",
        "type": "SequenceLocation"
      },
      "state": {
        "sequence": "T",
        "type": "SequenceState"
      },
      "type": "Allele"
    },
    "ga4gh:VA.iXjilHZiyCEoD3wVMPMXG3B8BtYfL88H": {
      "location": {
        "interval": {
          "end": 44908684,
          "start": 44908683,
          "type": "SimpleInterval"
        },
        "sequence_id": "ga4gh:SQ.IIB53T8CNeJJdUqzn9V_JnRtQadwWCbl",
        "type": "SequenceLocation"
      },
      "state": {
        "sequence": "C",
        "type": "SequenceState"
      },
      "type": "Allele"
    }
  },
  "hgvs_names": {
    "ga4gh:VA.UUvQpMYU5x8XXBS-RhBhmipTWe2AALzj": "NC_000019.10:g.44908822=",
    "ga4gh:VA.EgHPXXhULTwoP4-ACfs-YCXaeUQJBjH_": "NC_000019.10:g.44908822C>T",
    "ga4gh:VA.LQrGFIOAP8wEAybwNBo8pJ3yIG7tXWoh": "NC_000019.10:g.44908684=",
    "ga4gh:VA.iXjilHZiyCEoD3wVMPMXG3B8BtYfL88H": "NC_000019.10:g.44908684T>C"
  },
  "rs_names": {
    "ga4gh:VA.UUvQpMYU5x8XXBS-RhBhmipTWe2AALzj": "rs7412C",
    "ga4gh:VA.EgHPXXhULTwoP4-ACfs-YCXaeUQJBjH_": "rs7412T",
    "ga4gh:VA.LQrGFIOAP8wEAybwNBo8pJ3yIG7tXWoh": "rs429358C",
    "ga4gh:VA.iXjilHZiyCEoD3wVMPMXG3B8BtYfL88H": "rs429358T"
  },
  "freqs": {
    "gnomad": {
      "global": {
        "ga4gh:VA.UUvQpMYU5x8XXBS-RhBhmipTWe2AALzj": 0.9385,
        "ga4gh:VA.EgHPXXhULTwoP4-ACfs-YCXaeUQJBjH_": 0.0615,
        "ga4gh:VA.LQrGFIOAP8wEAybwNBo8pJ3yIG7tXWoh": 0.1385,
        "ga4gh:VA.iXjilHZiyCEoD3wVMPMXG3B8BtYfL88H": 0.8615
      }
    }
  }
}
```

5.3 Design Decisions

VRS contributors confronted numerous trade-offs in developing this specification. As these trade-offs may not be apparent to outside readers, this section highlights the most significant ones and the rationale for our design decisions, including:

5.3.1 Variation Rather than Variant

The abstract *Variation* class is intentionally not labeled "Variant", despite this being the primary term used in other molecular variation exchange formats (e.g. Variant Call Format, HGVS Sequence Variant Nomenclature). This is because the term "Variant" as used in the Genetics community is intended to describe discrete changes in nucleotide / amino acid sequence. "Variation", in contrast, captures other classes of molecular variation, including epigenetic alteration and transcript abundance. Capturing these other classes of variation is a *future goal* of VRS, as there are many annotations that will require these variation classes as the subject.

5.3.2 Allele Rather than Variant

The most primitive sequence assertion in VRS is the *Allele* entity. Colloquially, the words "allele" and "variant" have similar meanings and they are often used interchangeably. However, the VR contributors believe that it is essential to distinguish the state of the sequence from the change between states of a sequence. It is imperative that precise terms are used when modelling data. Therefore, within VR, Allele refers to a state and "variant" refers to the change from one Allele to another.

The word "variant", which implies change, makes it awkward to refer to the (unchanged) reference allele. Some systems will use an HGVS-like syntax (e.g., NC_000019.10:g.44906586G>G or NC_000019.10:g.44906586=) when referring to an unchanged residue. In some cases, such "variants" are even associated with allele frequencies. Similarly, a predicted consequence is better associated with an allele than with a variant.

5.3.3 Alleles are Fully Justified

In order to standardize the presentation of sequence variation, computed ids from VRS require that Alleles be fully justified from the description of the NCBI Variant Overprecision Correction Algorithm (VOCA). Furthermore, normalization rules must be identical for all sequence types; although this need not be a strict requirement, there is no reason to normalize using different strategies based on sequence type.

The choice of algorithm was relatively straightforward: VOCA is published, easily understood, easily implemented, and covers a wide range of cases.

The choice to fully justify is a departure from other common variation formats. The HGVS nomenclature recommendations, originally published in 1998, require that alleles be right normalized (3' rule) on all sequence types. The Variant Call Format (VCF), released as a PDF specification in 2009, made the conflicting choice to write variants left (5') normalized and anchored to the previous nucleotide.

Fully-justified alleles represent an alternate approach. A fully-justified representation does not make an arbitrary choice of where a variant truly occurs in a low-complexity region, but rather describes the final and unambiguous state of the resultant sequence.

5.3.4 Interbase Coordinates

Sequence ranges use an interbase coordinate system. Interbase coordinate conventions are used in this terminology because they provide conceptual consistency that is not possible with residue-based systems.

Important: The choice of what to count–residues versus inter-residue positions–has significant semantic implications for coordinates. Because interbase coordinates and the corresponding 0-based residue-counted coordinates are numerically identical in some circumstances, uninitiated readers often conflate the choice of numerical base with the choice of residue or inter-residue counting. Whereas the choice of numerical base is inconsequential, the semantic advantages of interbase are significant.

When humans refer to a range of residues within a sequence, the most common convention is to use an interval of ordinal residue positions in the sequence. While natural for humans, this convention has several shortcomings when dealing with sequence variation.

For example, interval coordinates are interpreted as exclusive coordinates for insertions, but as inclusive coordinates for substitutions and deletions; in effect, the interpretation of coordinates depends on the variant type, which is an unfortunate coupling of distinct concepts.

5.3.5 Modelling Language

The VRS collaborators investigated numerous options for modelling data, generating code, and writing the wire protocol. Required and desired selection criteria included:

- language-neutral – or at least C/C++, java, python
- high-quality tooling/libraries

- high-quality code generation
- documentation generation
- **supported constructs and data types**
  - typedefs/aliases
  - enums
  - lists, maps, and maps of lists/maps
  - nested objects
- protocol versioning (but not necessarily automatic adaptation)

Initial versions of the VRS logical model were implemented in UML, protobuf, and swagger/OpenAPI, and JSON Schema. We have implemented our schema in JSON Schema. Nonetheless, it is anticipated that some adopters of the VRS logical model may implement the specification in other protocols.

5.3.6 Serialization Strategy

There are many packages and proposals that aspire to a canonical form for json in many languages. Despite this, there are no ratified or *de facto* winners. Many packages have similar names, which makes it difficult to discern whether they are related or not (often not). Although some packages look like good single-language candidates, none are ready for multi-language use. Many seem abandoned. The need for a canonical json form is evident, and there was at least one proposal for an ECMA standard.

Therefore, we implemented our own *serialization format*, which is very similar to Gibson Canonical JSON (not to be confused with OLPC Canonical JSON).

5.4 Development Process

5.4.1 Versioning

VRS will follow GA4GH recommendation for semantic versioning with semver 2.0. The VRS GitHub repository will maintain the latest development code on the master branch for community review (see *Release Cycle*).

5.4.2 Release Cycle

Planned Features

Feature requests from the community are made through the generation of GitHub issues on the VRS repository, which are open for public review and discussion. Feature requests identified to support an unmet need are scheduled for discussion in our weekly VR calls. These discussions are used to inform whether or not a feature will be planned for development. The *Project Leadership* is responsible for making the final determination on whether a feature is to be added to VRS.

Requirements Gathering

Once a feature is planned for production, an issue requesting community feedback on use cases and technical requirements will be constructed (see example requirement issues).

Feature Development

Features will be developed to meet gathered requirements. Features ready for public review MAY be merged into the master branch by pull request through review and approval by at least one (non-authoring) *Project Maintainer*. Merged commits MAY be tagged as alpha releases when needed.

Version Review and Release

After completion of all planned features for a new minor or major version, a request for community review will be indicated by a beta release of the new version. Community stakeholders involved in the feature requests and requirements gathering for the included features are notified by Project Maintainers for review and approval of the release. After a community review period of at least two weeks, the Project Leadership will review and address any raised concerns for the reviewed version.

After passing review, new minor versions are released to production. If any features in the reviewed version are deemed to be significant additions to the specification by the Project Leadership, or if it is a major version change, instead a release candidate version will be released and submitted for GA4GH product approval. After approval, the new version is released to production.

5.4.3 Leadership

Project Leadership

As a product of the Genomic Knowledge Standards (GKS) Work Stream, project leadership is comprised of the Work Stream leadership:

- Alex Wagner (@ahwagner)
- Andy Yates (@andrewyatz)
- Bob Freimuth (@rrfreimuth)
- Javier Lopez (@javild)
- Larry Babb (@larrybabb)
- Matt Brush (@mbrush)
- Melissa Konopko (@MKonopko)
- Reece Hart (@reece)

Project Maintainers

Project maintainers are the leads of the GKS Variation Representation working group:

- Alex Wagner (@ahwagner)
- Larry Babb (@larrybabb)
- Reece Hart (@reece)

5.5 Future Plans

5.5.1 Overview

VRS covers a fundamental subset of data types to represent variation, thus far predominantly related to the replacement of a subsequence in a reference sequence. Increasing its applicability will require supporting more complex types of variation, including:

- alternative coordinate types such as nested ranges
- feature-based coordinates such as genes, cytogenetic bands, and exons
- copy number variation
- structural variation
- mosaicism and chimerism
- rule-based variation

The following sections provide a preview of planned concepts under way to address a broader representation of variation.

5.5.2 Intervals and Locations

VRS uses *Interval (Abstract Class)* and *Location (Abstract Class)* subclasses to define where variation occurs. The schema is designed to be extensible to new kinds of Intervals and Locations in order to support, for example, fuzzy coordinates or feature-based locations.

NestedInterval

Biological definition

None

Computational definition

An *Interval (Abstract Class)* comprised of an *inner* and *outer* *SimpleInterval*. The *NestedInterval* allows for the definition of "fuzzy" range endpoints by designating a potentially included region (the *outer* SimpleInterval) and required included region (the *inner* SimpleInterval).

Information model

Field	Type	Limits	Description
type	string	1..1	Interval type; MUST be set to '**NestedInterval**'
inner	*SimpleInterval*	1..1	known interval
outer	*SimpleInterval*	1..1	potential interval

Implementation guidance

- Implementations MUST enforce values *0 outer.start inner.start inner.end outer.end*. In the case of double-stranded DNA, this constraint holds even when a feature is on the complementary strand.

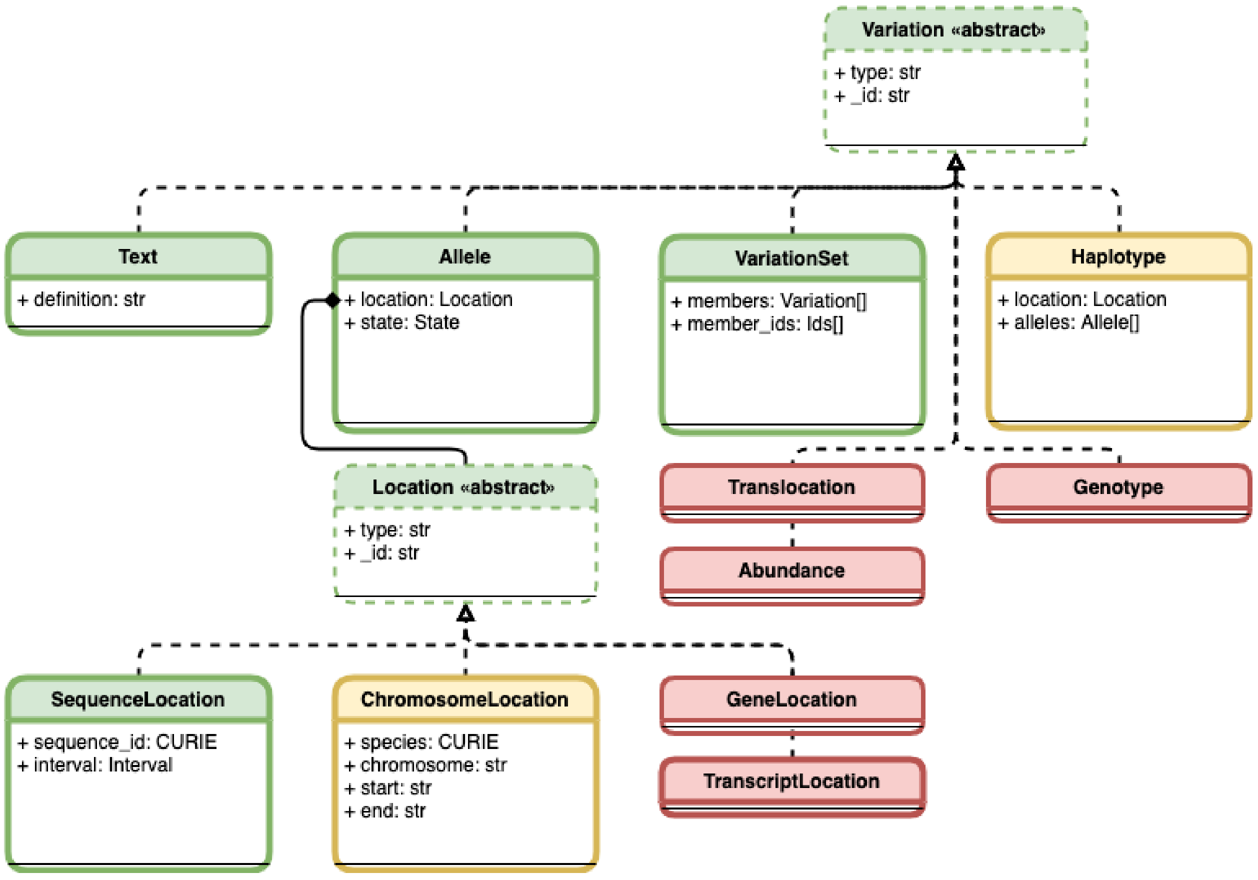

Fig. 1: An illustration of planned components for the VRS Schema.
Version 1.0 components are colored green. Components that are undergoing testing and evaluation and are candidates for the next release cycle are colored yellow. Components that are planned but still undergoing requirement gathering and initial development are colored red. The VRS Schema requires the use of multiple composite objects, which are grouped under four abstract classes: *Variation*, *Location (Abstract Class)*, *State (Abstract Class)*, and *Interval (Abstract Class)*. These classes and their relationships to the representation of Variation are illustrated here. All classes have a string type. Dashed borders denote abstract classes. Abstract classes are not instantiated. Thin solid borders denote classes that may be instantiated but are not identifiable. Bold borders denote identifiable objects (i.e., may be serialized and identified by computed identifier). Solid arrow lines denoted inheritance. Subclasses inherit all attributes from their parent. Inherited attributes are not shown.

ComplexInterval

Biological definition

Representation of complex coordinates based on relative locations or offsets from a known location. Examples include “left of” a given position and intronic positions measured from intron-exon junctions.

Computational definition

Under development.

Information model

Under development.

CytobandLocation

Biological definition

Imprecise chromosomal locations based on chromosomal staining.

Computational definition

Cytogenetic bands are defined by a chromosome name, band, and sub-band. In VRS, a cytogenetic location is an interval on a single chromsome with a start and end band and subband.

Information model

Under development.

GeneLocation

Biological definition

The symbolic location of a gene.

Computational definition

A gene location is made by reference to a gene identifier from NCBI, Ensembl, HGNC, or other public trusted authority.

Information model

Field	Type	Limits	Description
_id	*CURIE*	0..1	Location Id; MUST be unique within document
type	string	1..1	Location type; MUST be set to ‘**GeneLocation**’
gene_id	*CURIE*	1..1	CURIE-formatted gene identifier using NCBI numeric gene id.

Notes

- *gene_id* MUST be specified as a CURIE, using a CURIE prefix of *“NCBI”* and CURIE reference with the numeric gene id. Other trusted authorities MAY be permitted in future releases.

Implementation guidance

- GeneLocations MAY be converted to *SequenceLocation* using external data. The source of such data and mechanism for implementation is not defined by this specification.

5.5.3 State Classes

Additional *State (Abstract Class)* concepts that are being planned for future consideration in the specification.

CNVState

Note: This concept is being refined. Please comment at https://github.com/ga4gh/vr-spec/issues/46.

Biological definition

Variations in the number of copies of a segment of DNA. Copy number variations cover copy losses or gains and at known or unknown locations (including tandem repeats). Variations MAY occur at precise SequenceLocations, within nested intervals, or at GeneLocations. There is no lower or upper bound on CNV sizes.

Computational definition

Under development.

Information model

Field	Type	Limits	Description
type	string	1..1	State type; MUST be set to '**CNVState**'
location	*Location (Abstract Class)*	1..1	the Location of the copy ('**null**' if unknown)
min_copies	int	1..1	The minimum number of copies
max_copies	int	1..1	The maximum number of copies

5.5.4 Variation Classes

Additional *Variation* concepts that are being planned for future consideration in the specification. See *Variation* for more information.

Translocations

Note: This concept is being refined. Please comment at https://github.com/ga4gh/vr-spec/issues/103

Biological definition

The aberrant joining of two segments of DNA that are not typically contiguous. In the context of joining two distinct coding sequences, translocations result in a gene fusion, which is also covered by this VRS definition.

Computational definition

A joining of two sequences is defined by two *Location (Abstract Class)* objects and an indication of the join "pattern" (advice needed on conventional terminology, if any).

Information model

Under consideration. See https://github.com/ga4gh/vr-spec/issues/28.

Examples

t(9;22)(q34;q11) in BCR-ABL

5.5.5 Rule-based Variation

Some variations are defined by categorical concepts, rather than specific locations and states. These variations go by many terms, including *categorical variants*, *bucket variants*, *container variants*, or *variant classes*. These forms of variation are not described by any broadly-recognized variation format, but modeling them is a key requirement for the representation of aggregate variation descriptions as commonly found in biomedical literature. Our future work will focus on the formal specification for representing these variations with sets of rules, which we currently call *Rule-based Variation*.

RuleLocation

RuleLocation is a subclass of *Location (Abstract Class)* intended to capture locations defined by rules instead of specific contiguous sequences. This includes locations defined by sequence characteristics, e.g. *microsatellite regions*.

RuleState

RuleState is a subclass of *State (Abstract Class)* intended to capture states defined by categorical rules instead of sequence states. This includes *gain- / loss-of-function*, *oncogenic*, and *truncating* variation.

5.6 Proposal for GA4GH-wide Computed Identifier Standard

This appendix describes a proposal for creating a GA4GH-wide standard for serializing data, computing digests on serialized data, and constructing CURIE identifiers from the digests. Essentially, it is a generalization of the *Computed Identifiers* section.

This standard is proposed now because VRS needs a well-defined mechanism for generating identifiers. Changing the identifier mechanism later will create significant issues for VR adopters.

5.6.1 Background

The GA4GH mission entails structuring, connecting, and sharing data reliably. A key component of this effort is to be able to *identify* entities, that is, to associate identifiers with entities. Ideally, there will be exactly one identifier for each entity, and one entity for each identifier. Traditionally, identifiers are assigned to entities, which means that disconnected groups must coordinate on identifier assignment.

The computed identifier scheme proposed in VRS computes identifiers from the data itself. Because identifers depend on the data, groups that independently generate the same variation will generate the same computed identifier for that entity, thereby obviating centralized identifier systems and enabling identifiers to be used in isolated settings such as clinical labs.

The computed identifier mechanism is broadly applicable and useful to the entire GA4GH ecosystem. Adopting a common identifier scheme will make interoperability of GA4GH entities more obvious to consumers, will enable the entire organization to share common entity definitions (such as sequence identifiers), and will enable all GA4GH products to share tooling that manipulate identified data. In short, it provides an important consistency within the GA4GH ecosystem.

As a result, we are proposing that the computed identifier scheme described in VRS be considered for adoption as a GA4GH-wide standard. If the proposal is accepted by the GA4GH executive committee, the current VRS proposal will stand as-is; if the proposal is rejected, the VRS proposal will be modified to rescope the computed identifier mechanism to VR and under admininstration of the VR team.

5.6.2 Proposal

The following algorithmic processes, described in depth in the VR *Computed Identifiers* proposal, are included in this proposal by reference:

- **GA4GH Digest Serialization** is the process of converting an object to a canonical binary form based on JSON and inspired by similar (but unratified) JSON standards. This serialization for is used only for the purposes of computing a digest.
- **GA4GH Truncated Digest** is a convention for using SHA-512, truncated to 24 bytes, and encoding using base64url.
- **GA4GH Identification** is the CURIE-based syntax for constructing a namespaced and typed identifier for an object.

5.6.3 Type Prefixes

A GA4GH identifier is proposed to be constructed according to this syntax:

```
"ga4gh" ":" type_prefix "." digest
```

The *digest* is computed as described above. The type_prefix is a short alphanumeric code that corresponds to the type of object being represented. If this propsal is accepted, this "type prefix map" would be administered by GA4GH. (Currently, this map is maintained in a YAML file within the VRS repository, but it would be relocated on approval of this proposal.)

We propose the following guidelines for type prefixes:

- Prefixes SHOULD be short, approximately 2-4 characters.
- Prefixes SHOULD be for concrete types, not polymorphic parent classes.
- A prefix MUST map 1:1 with a schema type.
- Variation Representation types SHOULD start with V.
- Variation Annotation types SHOULD start with A.

5.6.4 Administration

If accepted, administration of these guidelines should be transferred to a technical steering committee. If not accepted, the VR team will assume administration of the existing prefixes.

5.7 Implementations

The libraries and applications listed below have implemented the GA4GH Variation Representation Specification to store and exchange variation data. They are listed here to demonstrate utility and as a resource for those considering implementing VRS. These packages are not supported by GA4GH.

5.7.1 Libraries

Libraries facilitate the use of the VRS, but do not implement a particular use or application. Although there is only one library currently, it is expected that others will eventually appear as VRS is adopted.

vr-python: GA4GH VRS Python Implementation

The GA4GH VR Python Implementation is an implementation for the GA4GH VRS. It supports all types covered by the VRS, implements Allele normalization and computed identifier generation, and provides "extra" features such as translation from HGVS, SPDI, and VCF formats. See vr-python notebooks for usage examples.

VRS MAY be used without using the Python implementation.

5.7.2 Applications and Web Services

Applications implement VRS to support specific use cases. Projects known to implement VRS are listed below. Descriptions are provided by the application authors.

ClinGen Allele Registry

ClinGen Allele Registry[1] provides identifiers for more than 900 million variants. Each identifier (canonical allele identifiers: CAIds) is an abstract concept which represents a group of identical variants based on alignment. Identifiers are retrievable irrespective of the reference sequence and normalization status.

As a Driver Project for GA4GH, ClinGen Allele Registry implements two standards: RefGet and VRS in the first implementation.

The API endpoints that support data retrieval in this two key standards are summarized in the following table.

HOST: https//reg.clinicalgenome.org/

API Path	Parameters	Response Format	Example	
RefGet				
[GET] /sequence/service-info	-	Refget v1.0.0	/sequence/service-info	
[GET] /sequence/{id}	id => TRUNC512 digest for reference sequence	Refget v1.0.0	/sequence/vYfm5TA_F-_BtIGjfzjGOj8b6IK5hCTx	
[GET] /sequence/{id}/metadata	id => TRUNC512 digest for reference sequence	Refget v1.0.0	/sequence/vYfm5TA_F-_BtIGjfzjGOj8b6IK5hCTx/metadata	
VR				
[GET] /vrAllele?hgvs={hgvs}	hgvs => HGVS expression	VRS v1.0	/vrAllele?hgvs=NC_000007.14:g.55181320A>T /vrAllele?hgvs=NC_000007.14:g.55181220del	

Support for GA4GH refget and VRS specs provided in ClinGen Allele Registry is independent from VR-Python. Support for this community standards is implemented in ClinGen Allele Registry through extension of code written in C++.

BRCA Exchange

BRCA Exchange[2] proposes an API endpoint which will share the variant list in VRS JSON model. Behind the scenes, all variants will be represented according to VRS, in a separate table of the BRCA Exchange database, and the contents

[1] Pawliczek P, Patel RY, et al. *ClinGen Allele Registry links information about genetic variants.* Hum Mutat 11 (2018). doi:10.1002/humu.23637

[2] Cline, M.S., et al. *BRCA Challenge: BRCA Exchange as a global resource for variants in BRCA1 and BRCA2.* PLoS Genet. 2018 Dec 26;14(12):e1007752. doi:10.1371/journal.pgen.1007752

of this table will be served by the BRCA Exchange API. A stand-alone executable will leverage these data to integrate the BRCA Exchange variant set with the ClinGen allele registry.

VICC Meta-knowledgebase

The Variant Interpretation for Cancer Consortium (VICC; https://cancervariants.org) has a collection of ~20K clinical interpretations associated with ~3,500 somatic variations and variation classes in a harmonized meta-knowledgebase[3] (see documentation at http://docs.cancervariants.org). Each interpretation is be linked to one or more variations or a variation class.

As a Driver Project for GA4GH, VICC is contributing to and/or adopting three GA4GH standards: VR, Variant Annotation (VA), and the Data Use Ontology (DUO). VICC supports queries on all VRS computed identifiers at the searchAssociations endpoint (vicc-docs). Features associated with each interpretation are represented as VRS objects.

Example queries:

- **Allele:** https://search.cancervariants.org/api/v1/associations?size=10&from=1&q=ga4gh:VA.mJbjSsW541oOsOtBoX36Mppr6hMjbjFr
- **SequenceLocation:** https://search.cancervariants.org/api/v1/associations?size=10&from=1&q=ga4gh:SL.gJeEs42k4qeXOKy9CJ515c0v2HTu8s4K
- **Text:** https://search.cancervariants.org/api/v1/associations?size=10&from=1&q=ga4gh:VT.9Wer7KrxALcPRDRGVKOEzf9ZEKZpOKK0

References:

5.8 Truncated Digest Timing and Collision Analysis

The GA4GH Digest uses a truncated SHA-512 digest in order to generate a unique identifier based on data that defines the object. This notebook discusses the choice of SHA-512 over other digest methods and the choice of truncation length.

Source: Reece Hart, CC-BY

5.8.1 Conclusions

- The computational time for SHA-512 is similar to that of other digest methods. Given that it is believed to distribute input bits more uniformly with no increased computational cost, it should be preferred for our use (and likely most uses).
- 24 bytes (192 bits) of digest is *ample* for VRS uses. Arguably, we could choose much smaller without significant risk of collision.

```
import hashlib
import math
import timeit

from IPython.display import display, Markdown

from ga4gh.vr.extras.utils import _format_time

algorithms = {'sha512', 'sha1', 'sha256', 'md5', 'sha224', 'sha384'}
```

[3] Wagner, A.H., et al. *A harmonized meta-knowledgebase of clinical interpretations of cancer genomic variants.* bioRxiv 366856 (2018). doi:10.1101/366856

5.8.2 Digest Timing

This section provides a rationale for the selection of SHA-512 as the basis for the Truncated Digest.

```
def blob(l):
    """return binary blob of length l (POSIX only)"""
    return open("/dev/urandom", "rb").read(l)

def digest(alg, blob):
    md = hashlib.new(alg)
    md.update(blob)
    return md.digest()

def magic_run1(alg, blob):
    t = %timeit -o digest(alg, blob)
    return t

def magic_tfmt(t):
    """format TimeitResult for table"""
    return "{a} ± {s} ([{b}, {w}])".format(
        a = _format_time(t.average),
        s = _format_time(t.stdev),
        b = _format_time(t.best),
        w = _format_time(t.worst),
    )
```

```
blob_lengths = [100, 1000, 10000, 100000, 1000000]
blobs = [blob(l) for l in blob_lengths]
```

```
table_rows = []
table_rows += [["algorithm"] + list(map(str,blob_lengths))]
table_rows += [["-"] * len(table_rows[0])]
for alg in sorted(algorithms):
    r = [alg]
    for i in range(len(blobs)):
        blob = blobs[i]
        t = timeit.timeit(stmt='digest(alg, blob)', setup='from __main__ import alg,␣
↪blob, digest', number=1000)
        r += [_format_time(t)]
    table_rows += [r]
table = "\n".join(["|".join(map(str,row)) for row in table_rows])
display(Markdown(table))
```

algorithm	100	1000	10000	100000	1000000
md5	1.02 ms	2.51 ms	23.4 ms	145 ms	1.44 s
sha1	1.02 ms	1.91 ms	11.3 ms	101 ms	1 s
sha224	1.21 ms	3.16 ms	23.1 ms	224 ms	2.2 s
sha256	1.18 ms	3.29 ms	23.3 ms	223 ms	2.2 s
sha384	1.17 ms	2.54 ms	16 ms	150 ms	1.47 s
sha512	1.2 ms	2.55 ms	16.1 ms	148 ms	1.47 s

Conclusion: SHA-512 computational time is comparable to that of other digest methods.

This is result was not expected initially. On further research, there is a clear explanation: The SHA-2 series of digests (which includes SHA-224, SHA-256, SHA-384, and SHA-512) is defined using 64-bit operations. When an implementation is optimized for 64-bit systems (as used for these timings), the number of cycles is essentially halved when compared to 32-bit systems and digests that use 32-bit operations. SHA-2 digests are indeed much slower than SHA-1 and MD5 on 32-bit systems, but such legacy platforms is not relevant to the Truncated Digest.

5.8.3 Collision Analysis

Our question: **For a hash function that generates digests of length b (bits) and a corpus of m messages, what is the probability p that there exists at least one collision?** This is the so-called Birthday Problem [6].

Because analyzing digest collision probabilities typically involve choices of mathematical approximations, multiple "answers" appear online. This section provides a quick review of prior work and extends these discussions by focusing the choice of digest length for a desired collision probability and corpus size.

Throughout the following, we'll use these variables:

- P = Probability of collision
- P' = Probability of no collision
- b = digest size, in bits
- s = digest space size, $s = 2^b$
- m = number of messages in corpus

The length of individual messages is irrelevant.

References

- [1] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
- [2] https://tools.ietf.org/html/rfc3548#section-4
- [3] http://stackoverflow.com/a/4014407/342839
- [4] http://stackoverflow.com/a/22029380/342839
- [5] http://preshing.com/20110504/hash-collision-probabilities/
- [6] https://en.wikipedia.org/wiki/Birthday_problem
- [7] https://en.wikipedia.org/wiki/Birthday_attack

Background: The Birthday Problem

Directly computing the probability of one or more collisions, P, in a corpus is difficult. Instead, we first seek to solve for P', the probability that a collision does not exist (i.e., that the digests are unique). Because are only two outcomes, $P + P' = 1$ or, equivalently, $P = 1 - P'$.

For a corpus of size $m = 1$, the probability that the digests of all $m = 1$ messages are unique is (trivially) 1:

$$P' = s/s = 1$$

because there are s ways to choose the first digest from among s possible values without a collision.

For a corpus of size $m = 2$, the probability that the digests of all $m = 2$ messages are unique is:

$$P' = 1 \times (\frac{s-1}{s})$$

because there are $s - 1$ ways to choose the second digest from among s possible values without a collision.

Continuing this logic, we have:

$$P' = \prod_{i=0}^{m-1} \frac{(s-i)}{s}$$

or, equivalently,

$$P' = \frac{s!}{s^m \cdot (s-m)!}$$

When the size of the corpus becomes greater than the size of the digest space, the probability of uniques is zero by the pigeonhole principle. Formally, the above equation becomes:

$$P' = \begin{cases} 1 & \text{if } m = 0 \\ \prod_{i=0}^{m-1} \frac{(s-i)}{s} & \text{if } 1 \leq m \leq s \\ 0 & \text{if } ms \end{cases}$$

For the remainder of this section, we'll focus on the case where $1 \leq m \ll s$. In addition, notice that the brute force computation is not feasible in practice because m and s will be very large (both $\gg 2^9$).

Approximation #1: Taylor approximation of terms of P'

The Taylor series expansion of the exponential function is

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

For $|x| \ll 1$, the expansion is dominated by the first terms and therecore $e^x \approx 1 + x$.

In the above expression for P', note that the product term $(s-i)/s$ is equivalent to $1 - i/s$. Combining this with the Taylor expansion, where $x = -i/s$ ($m \ll s$):

$$\begin{aligned} P' &\approx \prod_{i=0}^{m-1} e^{-i/s} \\ &= e^{-m(m-1)/2s} \end{aligned}$$

(The latter equivalence comes from converting the product of exponents to a single exponent of a summation of $-i/s$ terms, factoring out $1/s$, and using the series sum equivalence $\sum_{j=0}^{n} j = n(n+1)/2$ for $n \geq 0$.)

Approximation #2: Taylor approximation of P'

The above result for P' is also amenable to Taylor approximation. Setting $x = -m(m-1)/2s$, we continue from the previous derivation:

$$\begin{aligned} P' &\approx e^{-(m(m-1)/2s} \\ &\approx 1 + \frac{-m(m-1)}{2s} \end{aligned}$$

Approximation #3: Square approximation

For large m, we can approximate $m(m-1)$ as m^2 to yield

$$P' \approx 1 - m^2/2s$$

Summary of equations

We may now summarize equations to approximate the probability of digest collisions.

Table 1: Summary of Equations

Method	Probability of uniqueness(P')	Probability of collision($P = 1 - P'$)	Assumptions	Source/Comparison
exact	$\prod_{i=0^{m-1}} \frac{(s-i)}{s}$	$1 - P'$	$1 \leq m \leq s$	[1]
Taylor approximation on #1	$e^{-m(m-1)/2s}$	$1 - P'$	$m \ll s$	[1]
Taylor approximation on #2	$1 - \frac{m(m-1)}{2s}$	$\frac{m(m-1)}{2s}$	(same)	[1]
Large square approximation	$1 - \frac{m^2}{2s}$	$\frac{m^2}{2s}$	(same)	[2] (where $s = 2^n$)

- [1] https://en.wikipedia.org/wiki/Birthday_problem
- [2] http://preshing.com/20110504/hash-collision-probabilities/

5.8.4 Choosing a digest size

Now, we turn the problem around:

> **What digest length :math:‘b‘ is required to achieve a collision probability less than :math:‘P‘ for :math:‘m‘ messages?**

From the above summary, we have $P = m^2/2s$ for $m \ll s$. Rewriting with $s = 2^b$, we have the probability of a collision using b bits with m messages (sequences) is:

$$P(b, m) = m^2/2^{b+1}$$

Note that the collision probability depends on the number of messages, but not their size.

Solving for the minimum number of *bits* b as a function of an expected number of sequences m and a desired tolerance for collisions of P:

$$b(m, P) = \log_2\left(\frac{m^2}{P}\right) - 1$$

This equation is derived from equations that assume that $m \ll s$, where $s = 2^b$. When computing $b(m, P)$, we'll require that $m/s \leq 10^{-3}$ as follows:

$$m/s \leq 10^{-3}$$

is approximately equivalent to:

$$m/2^b \leq 2^{-5}$$

$$m \leq 2^{b-5}$$

$$log_2 m \leq b - 5$$

$$b \geq 5 + log_2 m$$

For completeness:

Solving for the number of messages:

$$m(b, P) = \sqrt{P * 2^{b+1}}$$

This equation is not used further in this analysis.

```
def b2B3(b):
    """Convert bits b to Bytes, rounded up modulo 3

    We report modulo 3 because the intent will be to use Base64 encoding, which is
    most efficient when inputs have a byte length modulo 3. (Otherwise, the resulting
    string is padded with characters that provide no information.)

    """
    return math.ceil(b/8/3) * 3

def B(P, m):
    """return the number of bits needed to achieve a collision probability
    P for m messages

    Assumes m << 2^b.

    """
    b = math.log2(m**2 / P) - 1
    if b < 5 + math.log2(m):
        return "-"
    return b2B3(b)
```

```
m_bins = [1E6, 1E9, 1E12, 1E15, 1E18, 1E21, 1E24, 1E30]
P_bins = [1E-30, 1E-27, 1E-24, 1E-21, 1E-18, 1E-15, 1E-12, 1E-9, 1E-6, 1E-3, 0.5]
```

```
table_rows = []
table_rows += [["#m"] + ["P<={P}".format(P=P) for P in P_bins]]
table_rows += [["-"] * len(table_rows[0])]
for n_m in m_bins:
    table_rows += [["{:g}".format(n_m)] + [B(P, n_m) for P in P_bins]]
table = "\n".join(["|".join(map(str,row)) for row in table_rows])
table_header = "### digest length (bytes) required for expected collision probability
↪$P$ over $m$ messages \n"
display(Markdown(table_header +  table))
```

digest length (bytes) required for expected collision probability P over m messages

#m	P<= 1e- 30	P<= 1e- 27	P<= 1e- 24	P<= 1e- 21	P<= 1e- 18	P<= 1e- 15	P<= 1e- 12	P<= 1e- 09	P<= 1e- 06	P<= 0.0 01	P<= 0.5
1e+ 06	18	18	15	15	15	12	12	9	9	9	6
1e+ 09	21	21	18	18	15	15	15	12	12	9	9
1e+ 12	24	24	21	21	18	18	15	15	15	12	12
1e+ 15	27	24	24	24	21	21	18	18	15	15	15
1e+ 18	30	27	27	24	24	24	21	21	18	18	15
1e+ 21	30	30	30	27	27	24	24	24	21	21	18
1e+ 24	33	33	30	30	30	27	27	24	24	24	21
1e+ 30	39	39	36	36	33	33	30	30	30	27	27

5.9 Glossary

computed identifier An identifier that is *generated* from the object's data. Multiple groups who generated *computed identifiers* the same way will generate the same identifier for the same underlying data.

digest, ga4gh_digest A digest is a digital fingerprint of a block of binary data. A digest is always the same size, regardless of the size of the input data. It is statistically extremely unlikely for two fingerprints to match when the underlying data are distinct.

serialization The process of converting an object in memory into a stream of bytes that may be sent via the network, saved in a database, or written to a file.

Bibliography

[Gibson] Gibson Canonical JSON

[OLPC] OLPC Canonical JSON

[JCS] JSON Canonicalization Scheme

Index