Internet Engineering Task Force (IETF)
Request for Comments: 8228
Category: Informational
ISSN: 2070-1721

A. Freytag

August 2017

Guidance on Designing Label Generation Rulesets (LGRs) Supporting Variant Labels

Abstract

Rules for validating identifier labels and alternate representations of those labels (variants) are known as Label Generation Rulesets (LGRs); they are used for the implementation of identifier systems such as Internationalized Domain Names (IDNs). This document describes ways to design LGRs to support variant labels. In designing LGRs, it is important to ensure that the label generation rules are consistent and well behaved in the presence of variants. The design decisions can then be expressed using the XML representation of LGRs that is defined in RFC 7940.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc8228.

Table of Contents

1. Introduction

Label Generation Rulesets (LGRs) that define the set of permissible labels may be applied to identifier systems that rely on labels, such as the Domain Name System (DNS) [RFC1034] [RFC1035]. To date, LGRs have mostly been used to define policies for implementing Internationalized Domain Names (IDNs) using IDNA2008 [RFC5890] [RFC5891] [RFC5892] [RFC5893] [RFC5894] in the DNS. This document aims to discuss the generation of LGRs for such circumstances, but the techniques and considerations here are almost certainly applicable to a wider range of internationalized identifiers.

In addition to determining whether a given label is eligible, LGRs may also define the condition under which alternate representations of these labels, so-called "variant labels", may exist and their status (disposition). In the most general sense, variant labels are typically labels that are either visually or semantically indistinguishable from another label in the context of the writing system or script supported by the LGR. Unlike merely similar labels, where there may be a measurable degree of similarity, variant labels considered here represent a form of equivalence in meaning or appearance. What constitutes an appropriate variant in any writing system or given context, particularly in the DNS, is assumed to have been determined ahead of time and therefore is not a subject of this document.

Once identified, variant labels are typically delegated to some entity together with the applied-for label, or permanently reserved, based on the disposition derived from the LGR. Correctly defined, variant labels can improve the security of an LGR, yet successfully defining variant rules for an LGR so that the result is well behaved is not always trivial. This document describes the basic considerations and constraints that must be taken into account and gives examples of what might be use cases for different types of variant specifications in an LGR.

This document does not address whether variants are an appropriate means to solve any given issue or the basis on which they should be defined. It is intended to explain in more detail the effects of various declarations and the trade-offs in making design choices. It implicitly assumes that any LGR will be expressed using the XML representation defined in [RFC7940] and therefore conforms to any requirements stated therein. Purely for clarity of exposition, examples in this document use a more compact notation than the XML syntax defined in [RFC7940]. However, the reader is expected to have some familiarity with the concepts described in that RFC (see Section 4).

The user of any identifier system, such as the DNS, interacts with it in the context of labels; variants are experienced as variant labels, i.e., two (or more) labels that are functionally "same as" under the conventions of the writing system used, even though their code point sequences are different. An LGR specification, on the other hand, defines variant mappings between code points and, only in a secondary step, derives the variant labels from these mappings. For a discussion of this process, see [RFC7940].

The designer of an LGR can control whether some or all of the variant labels created from an original label should be allocatable, i.e., available for allocation (to the original applicant), or whether some or all of these labels should be blocked instead, i.e., remain not allocatable (to anyone). This document describes how this choice of label disposition is accomplished (see Section 7).

The choice of desired label disposition would be based on the expectations of the users of the particular zone; it is not the subject of this document. Likewise, this document does not address the possibility of an LGR defining custom label dispositions. Instead, this document suggests ways of designing an LGR to achieve the selected design choice for handling variants in the context of the two standard label dispositions: "allocatable" and "blocked".

The information in this document is based on operational experience gained in developing LGRs for a wide number of languages and scripts using RFC 7940. This information is provided here as a benefit to the wider community. It does not alter or change the specification found in RFC 7940 in any way.

2. Variant Relations

A variant relation is fundamentally a "same as" relation; in other words, it is an equivalence relation. Now, the strictest sense of "same as" would be equality, and for any equality, we have both symmetry

A = B => B = A

and transitivity

A = B and B = C => A = C

The variant relation with its functional sense of "same as" must really satisfy the same constraint. Once we say A is the "same as" B, we also assert that B is the "same as" A. In this document, the symbol "~" means "has a variant relation with". Thus, we get

A ~ B => B ~ A

Likewise, if we make the same claim for B and C (B ~ C), then we get A ~ C, because if B is the "same as" both A and C, then A must be the "same as" C:

A ~ B and B ~ C => A ~ C

3. Symmetry and Transitivity

Not all potential relations between labels constitute equivalence, and those that do not are not transitive and may not be symmetric. For example, the degree to which labels are confusable is not transitive: two labels can be confusingly similar to a third without necessarily being confusable with each other, such as when the third one has a shape that is "in between" the other two. In contrast, a relation based on identical or effectively identical appearance would meet the criterion of transitivity, and we would consider it a variant relation. Examples of variant relations include other forms of equivalence, such as semantic equivalence.

Using [RFC7940], a set of mappings could be defined that is neither symmetric nor transitive; such a specification would be formally valid. However, a symmetric and transitive set of mappings is strongly preferred as a basis for an LGR, not least because of the benefits from an implementation point of view; for example, if all mappings are symmetric and transitive, it greatly simplifies the check for collisions between labels with variants. For this reason, we will limit the discussion in this document to those relations that are symmetric and transitive. Incidentally, it is often straightforward to verify mechanically whether an LGR is symmetric and/or transitive and to compute any mappings required to make it so (but see Section 15).

4. A Word on Notation

[RFC7940] defines an XML schema for Label Generation Rulesets in general and variant code points and sequences in particular (see Section 18). That notation is rather verbose and can easily obscure salient features to anyone not trained to read XML. For this reason, this document uses a symbolic shorthand notation in presenting the examples for discussion. This shorthand is merely a didactic tool

for presentation and is not intended as an alternative to or replacement for the XML syntax that is used in formally specifying an LGR under [RFC7940].

When it comes time to capture the LGR in a formal definition, the notation used for any of the examples in this document can be converted to the XML format as described in Section 18.

5. Variant Mappings

So far, we have treated variant relations as simple "same as" relations, ignoring that each relation representing equivalence would consist of a symmetric pair of reciprocal mappings. In this document, the symbol "-->" means "maps to".

A ~ B => A --> B, B --> A

In an LGR, these mappings are not defined directly between labels but between code points (or code point sequences; see Section 17). In the transitive case, given

A ~ B => A --> B, B --> A

A ~ C => A --> C, C --> A

we also get

B ~ C => B --> C, C --> B

for a total of six possible mappings. Conventionally, these are listed in tables in order of the source code point, like so:

```
A --> B
A --> C
B --> A
B --> C
C --> A
C --> B
```

As we can see, A, B, and C can each be mapped two ways.

6. Variant Labels

To create a variant label, each code point in the original label is successively replaced by all variant code points defined by a mapping from the original code point. For a label AAA (the letter "A" three times), the variant labels (given the mappings from the transitive example above) would be

```
AAB
ABA
ABB
BAA
BAB
BBA
BBB
AAC
...
CCC
```

So far, we have merely defined what the variant labels are, but we have not considered their possible dispositions. In the next section, we discuss how to set up the variant mappings so that some variant labels are mutually exclusive (blocked), but some may be allocated to the same applicant as the original label (allocatable).

7. Variant Types and Label Dispositions

Assume we wanted to allow a variant relation between code points O and A, and perhaps between O and B or O and C as well. Assuming transitivity, this would give us:

```
O ~ A ~ B ~ C
```

Now, further assume that we would like to distinguish the case where someone applies for OOO from the case where someone applies for the label ABC. In this case, we would like to allocate only the applied-for label OOO, but in the latter case, we would like to also allow the allocation of either the label OOO or the variant label ABC, or both, but not of any of the other possible variant labels, like OAO, BCO, or the like. (A real-world example might be the case where O represents an unaccented letter, while A, B, and C might represent various accented forms of the same letter. Because unaccented letters are a common fallback, there might be a desire to allocate an unaccented label as a variant, but not the other way around.)

How would we specify such a distinction?

The answer lies in labeling the mappings A --> O, B --> O, and C --> O with the type "allocatable" and the mappings O --> A, O --> B, and O --> C with the type "blocked". In this document, the symbol "x-->" means "maps with type blocked", and the symbol "a-->" means "maps with type allocatable". Thus:

```
O  x--> A
O  x--> B
O  x--> C
A  a--> O
B  a--> O
C  a--> O
```

When we generate all permutations of labels, we use mappings with different types depending on which code points we start from. The set of all permuted variant labels would be the same, but the disposition of the variant label depends on which label we start from (we call that label the "original" or "applied-for" label).

In creating an LGR with variants, all variant mappings should always be labeled with a type ([RFC7940] does not formally require a type, but any well-behaved LGR would be fully typed). By default, these types correspond directly to the dispositions for variant labels, with the most restrictive type determining the disposition of the variant label. However, as we shall see later, it is sometimes useful to assign types from a wider array of values than the final dispositions for the labels and then define explicitly how to derive label dispositions from them.

8. Allocatable Variants

If we start with AAA and use the mappings from Section 7, the permutation OOO will be the result of applying the mapping A a--> O at each code point. That is, only mappings with type "a" (allocatable) were used. To know whether we can allocate both the label OOO and the original label AAA, we track the types of the mappings used in generating the label.

We record the variant types for each of the variant mappings used in creating the permutation in an ordered list. Such an ordered list of variant types is called a "variant type list". In running text, we often show it enclosed in square brackets. For example, [a x -] means the variant label was derived from a variant mapping with the "a" variant type in the first code point position, "x" in the second code point position, and the original code point in the third position ("-" means "no variant mapping").

For our example permutation, we get the following variant type list (brackets dropped):

```
AAA --> OOO : a a a
```

From the variant type list, we derive a "variant type set", denoted by curly braces, that contains an unordered set of unique variant types in the variant type list. For the variant type list for the given permutation, [a a a], the variant type set is { a }, which has a single element "a".

Deciding whether to allow the allocation of a variant label then amounts to deriving a disposition for the variant label from the variant type set created from the variant mappings that were used to create the label. For example, the derivation

```
if "all variants" = "a" => set label disposition to "allocatable"
```

would allow OOO to be allocated, because the types of all variant mappings used to create that variant label from AAA are "a".

The "all-variants" condition is tolerant of an extra "-" in the variant set (unlike the "only-variants" condition described in Section 10). So, had we started with AOA, OAA, or AAO, the variant set for the permuted variant OOO would have been { a - } because in each case one of the code points remains the same code point as the original. The "-" means that because of the absence of a mapping O --> O, there is no variant type for the O in each of these labels.

The "all-variants" = "a" condition ignores the "-", so using the derivation from above, we find that OOO is an allocatable variant for each of the labels AOA, OAA, or AAO.

Allocatable variant labels, especially large numbers of allocatable variants per label, incur a certain cost to users of the LGR. A well-behaved LGR will minimize the number of allocatable variants.

9. Blocked Variants

Blocked variants are not available to another registrant. They therefore protect the applicant of the original label from someone else registering a label that is the "same as" under some user-perceived metric. Blocked variants can be a useful tool even for scripts for which no allocatable labels are ever defined.

If we start with OOO and use the mappings from Section 7, the permutation AAA will have been the result of applying only mappings with type "blocked", and we cannot allocate the label AAA, only the original label OOO. This corresponds to the following derivation:

```
if "any variants" = "x" => set label disposition to "blocked"
```

Additionally, to prevent allocating ABO as a variant label for AAA, we need to make sure that the mapping A --> B has been defined with type "blocked", as in

```
A  x--> B
```

so that

```
AAA --> ABO: - x a.
```

Thus, the set {x a} contains at least one "x" and satisfies the derivation of a blocked disposition for ABO when AAA is applied for.

If an LGR results in a symmetric and transitive set of variant labels, then the task of determining whether a label or its variants collide with another label or its variants can be implemented very efficiently. Symmetry and transitivity imply that sets of labels that are mutual variants of each other are disjoint from all other such sets. Only labels within the same set can be variants of each other. Identifying the variant set can be an O(1) operation, and enumerating all variants is not necessary.

10. Pure Variant Labels

Now, if we wanted to prevent allocation of AOA when we start from AAA, we would need a rule disallowing a mix of original code points and variant code points; this is easily accomplished by use of the "only-variants" qualifier, which requires that the label consist entirely of variants and that all the variants are from the same set of types.

```
if "only-variants" = "a" => set label disposition to "allocatable"
```

The two code points A in AOA are not arrived at by variant mappings, because the code points are unchanged and no variant mappings are defined for A --> A. So, in our example, the set of variant mapping types is

```
AAA --> AOA:  - a -
```

but unlike the "all-variants" condition, "only-variants" requires a variant type set { a } corresponding to a variant type list [a a a] (no - allowed). By adding a final derivation

```
else if "any-variants" = "a" => set label disposition to "blocked"
```

and executing that derivation only on any remaining labels, we disallow AOA when starting from AAA but still allow OOO.

Derivation conditions are always applied in order, with later derivations only applying to labels that did not match any earlier conditions, as indicated by the use of "else" in the last example. In other words, they form a cascade.

11. Reflexive Variants

But what if we started from AOA? We would expect the original label OOO to be allocatable, but, using the mappings from Section 7, the variant type set would be

```
AOA --> OOO:  a - a
```

because the middle O is unchanged from the original code point. Here is where we use a reflexive mapping. Realizing that O is the "same as" O, we can map it to itself. This is normally redundant, but adding an explicit reflexive mapping allows us to specify a disposition on that mapping:

```
O  a--> O
```

With that, the variant type list for AOA --> OOO becomes:

```
AOA --> OOO: a a a
```

and the label OOO again passes the derivation condition

```
if "only-variants" = "a" => set label disposition to "allocatable"
```

as desired. This use of reflexive variants is typical whenever derivations with the "only-variants" qualifier are used. If any code point uses a reflexive variant, a well-behaved LGR would specify an appropriate reflexive variant for all code points.

12. Limiting Allocatable Variants by Subtyping

As we have seen, the number of variant labels can potentially be large, due to combinatorics. Sometimes it is possible to divide variants into categories and to stipulate that only variant labels with variants from the same category should be allocatable. For some LGRs, this constraint can be implemented by a rule that disallows code points from different categories to occur in the same allocatable label. For other LGRs, the appropriate mechanism may be dividing the allocatable variants into subtypes.

To recap, in the standard case, a code point C can have (up to) two types of variant mappings

```
C  x--> X
C  a--> A
```

where a--> means a variant mapping with type "allocatable" and x--> means "blocked". For the purpose of the following discussion, we name the target code point with the corresponding uppercase letter.

Subtyping allows us to distinguish among different types of allocatable variants. For example, we can define three new types: "s", "t", and "b". Of these, "s" and "t" are mutually incompatible, but "b" is compatible with either "s" or "t" (in this case, "b" stands for "both"). A real-world example for this might be variant mappings appropriate for "simplified" or "traditional" Chinese variants, or appropriate for both.

With subtypes defined as above, a code point C might have (up to) four types of variant mappings

```
C  x--> X
C  s--> S
C  t--> T
C  b--> B
```

and explicit reflexive mappings of one of these types

```
C  s--> C
C  t--> C
C  b--> C
```

As before, all mappings must have one and only one type, but each code point may map to any number of other code points.

We define the compatibility of "b" with "t" or "s" by our choice of derivation conditions as follows

```
if "any-variants" = "x" =>  blocked
else if "only-variants" = "s" or "b" =>  allocatable
else if "only-variants" = "t" or "b" =>  allocatable
else if "any-variants" = "s" or "t" or "b" =>  blocked
```

An original label of four code points

```
CCCC
```

may have many variant labels, such as this example listed with its corresponding variant type list:

```
CCCC --> XSTB : x s t b
```

This variant label is blocked because to get from C to B required x-->. (Because variant mappings are defined for specific source code points, we need to show the starting label for each of these examples, not merely the code points in the variant label.) The variant label

```
CCCC --> SSBB : s s b b
```

is allocatable, because the variant type list contains only allocatable mappings of subtype "s" or "b", which we have defined as being compatible by our choice of derivations. The actual set of variant types {s, b} has only two members, but the examples are easier to follow if we list each type. The label

```
CCCC --> TTBB : t t b b
```

is again allocatable, because the variant type set {t, b} contains only allocatable mappings of the mutually compatible allocatable subtypes "t" or "b". In contrast,

```
CCCC --> SSTT : s s t t
```

is not allocatable, because the type set contains incompatible subtypes "t" and "s" and thus would be blocked by the final derivation.

The variant labels

```
  CCCC --> CSBB : c s b b
  CCCC --> CTBB : c t b b
```

are only allocatable based on the subtype for the C --> C mapping, which is denoted here by "c" and (depending on what was chosen for the type of the reflexive mapping) could correspond to "s", "t", or "b".

If the subtype is "s", the first of these two labels is allocatable; if it is "t", the second of these two labels is allocatable; if it is "b", both labels are allocatable.

So far, the scheme does not seem to have brought any huge reduction in allocatable variant labels, but that is because we tacitly assumed that C could have all three types of allocatable variants "s", "t", and "b" at the same time.

In a real-world example, the types "s", "t", and "b" are assigned so that each code point C normally has, at most, one non-reflexive variant mapping labeled with one of these subtypes, and all other mappings would be assigned type "x" (blocked). This holds true for most code points in existing tables (such as those used in current IDN Top-Level Domains (TLDs)), although certain code points have exceptionally complex variant relations and may have an extra mapping.

13. Allowing Mixed Originals

If the desire is to allow original labels (but not variant labels) that are s/t mixed, then the scheme needs to be slightly refined to distinguish between reflexive and non-reflexive variants. In this document, the symbol "r-n" means "a reflexive (identity) mapping of type 'n'". The reflexive mappings of the preceding section thus become:

```
C  r-s--> C
C  r-t--> C
C  r-b--> C
```

With this convention, and redefining the derivations

```
if "any-variants" = "x" =>  blocked
else if "only-variants" = "s" or "r-s" or "b" or "r-b" => allocatable
else if "only-variants" = "t" or "r-t" or "b" or "r-b" => allocatable
else if "any-variants" = "s" or "t" or "b"  => blocked
else => allocatable
```

any labels that contain only reflexive mappings of otherwise mixed type (in other words, any mixed original label) now fall through, and their disposition is set to "allocatable" in the final derivation.

In a well-behaved LGR, it is preferable to explicitly define the derivation for allocatable labels instead of using a fall through. In the derivation above, code points without any variant mappings fall through and become allocatable by default if they are part of an original label. Especially in a large repertoire, it can be difficult to identify which code points are affected. Instead, it is preferable to mark them with their own reflexive mapping type "neither" or "r-n".

```
C  r-n--> C
```

With that, we can change

```
else =>  allocatable
```

to

```
else if "only-variants" = "r-s" or "r-t" or "r-b" or "r-n"
     =>  allocatable
else => invalid
```

This makes the intent more explicit, and by ensuring that all code points in the LGR have a reflexive mapping of some kind, it is easier to verify the correct assignment of their types.

14. Handling Out-of-Repertoire Variants

At first, it may seem counterintuitive to define variants that map to code points that are not part of the repertoire. However, for zones for which multiple LGRs are defined, there may be situations where labels valid under one LGR should be blocked if a label under another LGR is already delegated. This situation can arise whether or not the repertoires of the affected LGRs overlap and, where repertoires overlap, whether or not the labels are both restricted to the common subset.

In order to handle this exclusion relation through definition of variants, it is necessary to be able to specify variant mappings to some code point X that is outside an LGR’s repertoire, R:

```
C  x--> X : where C = elementOf(R) and X != elementOf(R)
```

Because of symmetry, it is necessary to also specify the inverse mapping in the LGR:

```
X  x--> C : where X != elementOf(R) and C = elementOf(R)
```

This makes X a source of variant mappings, and it becomes necessary to identify X as being outside the repertoire, so that any attempt to apply for a label containing X will lead to a disposition of "invalid", just as if X had never been listed in the LGR. The mechanism to do this uses reflexive variants but with a new type of reflexive mapping of "out-of-repertoire-var", shown as "r-o-->":

```
X  r-o--> X
```

This indicates X != elementOf(R), as long as the LGR is provided with a suitable derivation, so that any label containing "r-o-->" is assigned a disposition of "invalid", just as if X was any other code point not part of the repertoire. The derivation used is:

```
if "any-variant" = "out-of-repertoire-var" => invalid
```

It is inserted ahead of any other derivation of the "any-variant" kind in the chain of derivations. As a result, instead of the minimum two symmetric variants, for any out-of-repertoire variants, there are a minimum of three variant mappings defined:

```
C  x--> X
X  x--> C
X  r-o--> X
```

where C = elementOf(R) and X != elementOf(R).

Because no variant label with any code point outside the repertoire could ever be allocated, the only logical choice for the non-reflexive mappings to out-of-repertoire code points is "blocked".

15. Conditional Variants

Variant mappings are based on whether code points are "same as" to the user. In some writing systems, code points change shape based on where they occur in the word (positional forms). Some code points have matching shapes in some positions but not in others. In such cases, the variant mapping exists only for some possible positions or, more generally, only for some contexts. For other contexts, the variant mapping does not exist.

For example, take two code points that have the same shape at the end of a label (or in final position) but not in any other position. In that case, they are variants only when they occur in the final position, something we indicate like this:

```
final: C --> D
```

In cursively connected scripts, like Arabic, a code point may take its final form when next to any following code point that interrupts the cursive connection, not just at the end of a label. (We ignore the isolated form to keep the discussion simple; if included, "final" might be "final-or-isolate", for example).

From symmetry, we expect that the mapping D --> C should also exist only when the code point D is in final position. (Similar considerations apply to transitivity.)

Sometimes a code point has a final form that is practically the same as that of some other code point while sharing initial and medial forms with another.

```
final: C --> D
!final: C --> E
```

Here, the case where the condition is the opposite of final is shown as "!final".

Because shapes differ by position, when a context is applied to a variant mapping, it is treated independently from the same mapping in other contexts. This extends to the assignment of types. For example, the mapping C --> F may be "allocatable" in final position but "blocked" in any other context:

```
final:  C  a--> F
!final: C  x--> F
```

Now, the type assigned to the forward mapping is independent of the reverse symmetric mapping or any transitive mappings. Imagine a situation where the symmetric mapping is defined as F a--> C, that is, all mappings from F to C are "allocatable":

```
final: F  a--> C
!final: F  a-->C
```

Why not simply write F a--> C? Because the forward mapping is divided by context. Adding a context makes the two forward variant mappings distinct, and that needs to be accounted for explicitly in the reverse mappings so that human and machine readers can easily

verify symmetry and transitivity of the variant mappings in the LGR. (This is true even though the two opposite contexts of "final" and "!final" should together cover all possible cases.)

16. Making Conditional Variants Well Behaved

To ensure that LGR with contextual variants is well behaved, it is best to always use "fully qualified" variant mappings that always agree in the names of the context rules for forward and reverse mappings. It is also necessary to ensure that no label can match more than one context for the same mapping. Using mutually exclusive contexts, such as "final" and "!final", is an easy way to ensure that.

However, it is not always necessary to define dual or multiple contexts that together cover all possible cases. For example, here are two contexts that do not cover all possible positional contexts:

```
final: C --> D
initial: C --> D.
```

A well-behaved LGR using these two contexts would define all symmetric and transitive mappings involving C, D, and their variants consistently in terms of the two conditions "final" and "initial" and ensure that both cannot be satisfied at the same time by some label.

In addition to never defining the same mapping with two contexts that may be satisfied by the same label, a well-behaved LGR never combines a variant mapping with a context with the same variant mapping without a context:

```
context: C --> D
C --> D
```

Inadvertent mixing of conditional and unconditional variants can be detected and flagged by a parser, but verifying that two formally distinct contexts are never satisfied by the same label would depend on the interaction between labels and context rules, which means that it will be up to the LGR designer to ensure that the LGR is well behaved.

A well-behaved LGR never assigns conditions on a reflexive variant, as that is effectively no different from having a context on the code point itself; the latter is preferred.

Finally, for symmetry to work as expected, the context must be defined such that it is satisfied for both the original code point in the context of the original label and for the variant code point in the variant label. In other words, the context should be "stable under variant substitution" anywhere in the label.

Positional contexts usually satisfy this last condition; for example, a code point that interrupts a cursive connection would likely share this property with any of its variants. However, as it is possible in principle to define other kinds of contexts, it is necessary to make sure that the LGR is well behaved in this aspect at the time the LGR is designed.

Due to the difficulty in verifying these constraints mechanically, it is essential that an LGR designer document the reasons why the LGR can be expected to meet them and the details of the techniques used to ensure that outcome. This information should be found in the description element of the LGR.

In summary, conditional contexts can be useful for some cases, but additional care must be taken to ensure that an LGR containing conditional contexts is well behaved. LGR designers would be well advised to avoid using conditional contexts and to prefer unconditional rules whenever practical, even though it will doubtlessly reduce the number of labels practically available.

17. Variants for Sequences

Variant mappings can be defined between sequences or between a code point and a sequence. For example, one might define a "blocked" variant between the sequence "rn" and the code point "m" because they are practically indistinguishable in common UI fonts.

Such variants are no different from variants defined between single code points, except if a sequence is defined such that there is a code point or shorter sequence that is a prefix (initial subsequence) and both it and the remainder are also part of the repertoire. In that case, it is possible to create duplicate variants with conflicting dispositions.

The following shows such an example resulting in conflicting reflexive variants:

```
A  a--> C
AB  x--> CD
```

where AB is a sequence with an initial subsequence of A. For example, B might be a combining code point used in sequence AB. If B only occurs in the sequence, there is no issue, but if B also occurs by itself, for example:

```
B  a--> D
```

then a label "AB" might correspond to either {A}{B}, that is, the two code points, or {AB}, the sequence, where the curly braces show the sequence boundaries as they would be applied during label validation and variant mapping.

A label AB would then generate the "allocatable" variant label {C}{D} and the "blocked" variant label {CD}, thus creating two variant labels with conflicting dispositions.

For the example of a blocked variant between "m" and "rn" (and vice versa), there is no issue as long as "r" and "n" do not have variant mappings of their own, so that there cannot be multiple variant labels for the same input. However, it is preferable to avoid ambiguities altogether where possible.

The easiest way to avoid an ambiguous segmentation into sequences is by never allowing both a sequence and all of its constituent parts simultaneously as independent parts of the repertoire, for example, by not defining B by itself as a member of the repertoire.

Sequences are often used for combining sequences that consist of a base character B followed by one or more combining marks C. By enumerating all sequences in which a certain combining mark is expected and by not listing the combining mark by itself in the LGR, the mark cannot occur outside of these specifically enumerated contexts. In cases where enumeration is not possible or practicable, other techniques can be used to prevent ambiguous segmentation, for example, a context rule on code points that disallows B preceding C in any label except as part of a predefined sequence or class of sequences. The details of such techniques are outside the scope of this document (see [RFC7940] for information on context rules for code points).

18. Corresponding XML Notation

The XML format defined in [RFC7940] corresponds fairly directly to the notation used for variant mappings in this document. (There is no notation in the RFC for variant type sets). In an LGR document, a simple member of a repertoire that does not have any variants is listed as:

```
<char cp="nnnn" />
```

where nnnn is the [UNICODE] code point value in the standard uppercase hexadecimal notation padded to at least 4 digits and without leading "U+". For a code point sequence of length 2, the XML notation becomes:

```
<char cp="uuuu vvvvv" />
```

Variant mappings are defined by nesting <var> elements inside the <char> element. For example, a variant relation of type "blocked"

```
  C  x--> X
```

is expressed as

```
  <char cp="nnnn">
    <var cp="mmmm" type="blocked" />
  </char>
```

where "x-->" identifies a "blocked" type. (Other types include "a-->" for "allocatable", for example. Here, nnnn and mmmm are the [UNICODE] code point values for C and X, respectively. Either C or X could be a code point sequence or a single code point.

A reflexive mapping is specified the same way, except that it always uses the same code point value for both the <char> and <var> element, for example:

```
  X  r-o--> X
```

would correspond to

```
<char cp="nnnn"><var cp="nnnn" type="out-of-repertoire-var" /></char>
```

Multiple <var> elements may be nested inside a single <char> element, but their "cp" values must be distinct (unless attributes for context rules are present and the combination of "cp" value and context attributes are distinct).

```
<char cp="nnnn">
  <var cp="kkkk" type="allocatable" />
  <var cp="mmmm" type="blocked" />
</char>
```

A set of conditional variants like

```
final: C  a--> K
!final: C  x--> K
```

would correspond to

```
<var cp="kkkk" when="final" type="allocatable" />
<var cp="kkkk" not-when="final" type="blocked" />
```

where the string "final" references a name of a context rule. Context rules are defined in [RFC7940]; they conceptually correspond to regular expressions. The details of how to create and define these rules are outside the scope of this document. If the label matches the context defined in the rule, the variant mapping is valid and takes part in further processing. Otherwise, it is invalid and ignored. Using the "not-when" attribute inverts the sense of the match. The two attributes are mutually exclusive.

A derivation of a variant label disposition

```
if "only-variants" = "s" or "b" => allocatable
```

is expressed as

```
<action disp="allocatable" only-variants= "s b" />
```

Instead of using "if" and "else if", the <action> elements implicitly form a cascade, where the first action triggered defines the disposition of the label. The order of action elements is thus significant.

For the full specification of the XML format, see [RFC7940].

19. IANA Considerations

This document does not require any IANA actions.

20. Security Considerations

As described in [RFC7940], variants may be used as a tool to reduce certain avenues of attack in security-relevant identifiers by allowing certain labels to be "mutually exclusive or registered only to the same user". However, if indiscriminately designed, variants may themselves contribute to risks to the security or usability of the identifiers, whether resulting from an ambiguous definition or from allowing too many allocatable variants per label.

The information in this document is intended to allow the reader to design a specification of an LGR that is "well behaved" with respect to variants; as used here, this term refers to an LGR that is predictable in its effects to the LGR author (and reviewer) and more reliable in its implementation.

A well-behaved LGR is not merely one that can be expressed in [RFC7940], but, in addition, it actively avoids certain edge cases not prevented by the schema, such as those that would result in ambiguities in the specification of the intended disposition for some variant labels. By applying the additional considerations introduced in this document, including adding certain declarations that are optional under the schema and may not alter the results of processing a label, such an LGR becomes easier to review and its implementations easier to verify.

It should be noted that variants are an important part, but only a part, of an LGR design. There are many other features of an LGR that this document does not touch upon. Also, the question of whether to define variants at all, or what labels are to be considered variants of each other, is not addressed here.

21. References

21.1. Normative References

[RFC7940] Davies, K. and A. Freytag, "Representing Label Generation Rulesets Using XML", RFC 7940, DOI 10.17487/RFC7940, August 2016, <https://www.rfc-editor.org/info/rfc7940>.

21.2. Informative References

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC5891] Klensin, J., "Internationalized Domain Names in Applications (IDNA): Protocol", RFC 5891, DOI 10.17487/RFC5891, August 2010, <https://www.rfc-editor.org/info/rfc5891>.

[RFC5892] Faltstrom, P., Ed., "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)", RFC 5892, DOI 10.17487/RFC5892, August 2010, <https://www.rfc-editor.org/info/rfc5892>.

[RFC5893] Alvestrand, H., Ed. and C. Karp, "Right-to-Left Scripts for Internationalized Domain Names for Applications (IDNA)", RFC 5893, DOI 10.17487/RFC5893, August 2010, <https://www.rfc-editor.org/info/rfc5893>.

[RFC5894] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Background, Explanation, and Rationale", RFC 5894, DOI 10.17487/RFC5894, August 2010, <https://www.rfc-editor.org/info/rfc5894>.

[UNICODE] The Unicode Consortium, "The Unicode Standard", <http://www.unicode.org/versions/latest/>.

Acknowledgments

Contributions that have shaped this document have been provided by Marc Blanchet, Ben Campbell, Patrik Faltstrom, Scott Hollenbeck, Mirja Kuehlewind, Sarmad Hussain, John Klensin, Alexey Melnikov, Nicholas Ostler, Michel Suignard, Andrew Sullivan, Wil Tan, and Suzanne Woolf.

Author’s Address

Asmus Freytag

Email: asmus@unicode.org