

Feedback on PRI #555 — Finite automata for line breaking and segmentation

Public review feedback · 20 September 2026

I maintain [unicode_string](#), a conformant implementation of UAX #14 and UAX #29 written in [Elixir](#), along with the supporting libraries it depends on for Unicode Character Database introspection ([unicode](#)) and UnicodeSet expressions ([unicode_set](#)).

This feedback comes from having implemented the proposal against the Version 18.0 data, compared it directly with my existing hand-written implementation, and then adopted it: the published tables drive all four break types in the released version.

Summary: I support the proposal. The published state machines were more conformant than my hand-written rules, easier to trace back to the standard, and faster for line breaking. Sections 3 and 6.1 describe a category of defect I had not anticipated which you may find useful as additional evidence: implementations that are current with the standard and have transcribed a rule correctly, but evaluate it over an approximated set of characters. The specific issues I raise in section 5 are all things I hit while implementing, and none of them undermines the case for formalising the proposal.

1. Platform context

Elixir runs on the Erlang virtual machine (also known as the BEAM or OTP) — a dynamic functional platform designed for concurrency and fault tolerance rather than for tight loops over text. It will probably never be the right platform for a text editor, an HTML renderer, or any interactive layout engine, and I do not offer these measurements as though it were. They are useful precisely because they come from a platform with none of ICU’s advantages: whatever the state machine does better here, it does better for structural reasons rather than because of hand-tuned C.

2. The cost of a conformant implementation is not mostly the algorithm

Implementing UAX #14 and UAX #29 has required years of my time investing in libraries that have nothing to do with segmentation: a Unicode Character Database introspection library covering the enumerated and binary properties, and a UnicodeSet implementation able to evaluate the expressions the standard uses to define its classes. I expect this is true of most language platforms outside the small set that can link ICU. The emerging [TR #61](#) is a significant and positive step towards improving this situation.

This bears directly on the proposal. Publishing the state machines shifts work from “transcribe and maintain thirty-odd interacting rules” to “evaluate a list of UnicodeSet expressions and index two tables” — and the second is a much smaller and more mechanical surface. I think it’s a real reduction in the barrier to entry,

but only to the extent that the data files stay within a subset of UnicodeSet that implementations can be expected to support.

This is not a hypothetical constraint. Segmentation libraries claiming conformance exist in several ecosystems, but general-purpose UnicodeSet implementations outside ICU do not seem to exist (from my searches) — and the symbol files in this proposal are written in UnicodeSet. Section 6 surveys what appears to be actually available; issue 5d is the specific request that follows from it.

3. Where my implementation has been, and what actually went wrong

My first segmentation implementation compiled each rule into a pair of regular expressions evaluated at every candidate position. It was very slow — some inputs took seconds — but it was almost line-for-line traceable to the standard, and maintenance was correspondingly easy: when a rule changed, one regular expression changed.

I replaced it with a *direct-coded rule engine*: the rules compiled into ordered guards and function clauses over a small carried state, scanning the string once. That was one to three orders of magnitude faster, and it is what I shipped until this month. But the rules became dispersed across a long ordered decision function plus a carried state vector, and the correspondence with the standard weakened considerably.

Preparing this feedback turned up eight defects in that engine, plus one rule I was applying that UAX #14 had removed several versions earlier. It was still 99.81% compliant but that's not the same thing as compliant, performant and maintainable. What I did not expect was how the defects divided:

Three were transcription failures in my implementation — a rule absent or incompletely expressed. LB10 was not implemented at all, LB28a was absent, and LB25 did not track the `NU (SY | IS)*` run it is defined over. These are the failures from my hand-transcription, and they are the ones a careful reading of the annex would catch but it isn't easy!

Five were property derivation failures — the rule was present and its shape was right, but it was evaluated over the wrong or an insufficient set of characters:

Rule	Property the rule is defined over	What my implementation did
LB1	<code>General_Category</code> , to resolve <code>SA</code> to <code>CM</code> or <code>AL</code>	Resolved all <code>SA</code> to <code>AL</code>
LB15a, LB15b	<code>General_Category</code> , for initial (<code>Pi</code>) and final (<code>Pf</code>) quotation	Treated all <code>QU</code> alike
LB19, LB19a	<code>General_Category</code> and <code>East_Asian_Width</code>	Suppressed breaks around every <code>QU</code>
LB30	<code>East_Asian_Width</code> , excluding <code>F</code> , <code>W</code> , <code>H</code>	Applied the rule to all <code>OP</code> and <code>CP</code>
LB30b	<code>Extended_Pictographic</code> intersected with <code>General_Category=Cn</code>	Omitted that alternative

This distinction is relevant for the proposal because the two categories are not equally tractable. A transcription failure is visible in the code: the rule is missing, and comparing the code against Section 6 finds it. A property derivation failure is invisible. The rule is present, it reads correctly, and it produces

wrong answers only for the characters where the property distinction bites — which are, by construction, the characters an implementer is least likely to have in a test case.

Every one of the five is a place where the annex expresses a class as a set operation over UCD properties, and I approximated the set. That is precisely what the symbol files in this proposal remove: `QUMpimPf` is `[\p{Line_Break=Quotation}-\p{gc=Pi}-\p{gc=Pf}]`, and there is nothing left to approximate. Thirty-nine of the seventy-two line-break symbols reference `General_Category` or `East_Asian_Width` — more than half of the symbol table exists to make exactly the distinctions I got wrong.

Section 6.1 gives an example of the same failure mode in a much more mature implementation than mine.

4. Implementing the proposed DFA, and adopting it

I implemented section 9.3.2 against the Version 18.0 data — the review drop at first, and unchanged against the released files once 18.0.0 shipped on 16 September — as roughly 130 lines of engine plus a compile-time reader for the three data files per break type. Both engines were then given equivalent optimisation, so the comparison measures the two approaches rather than the effort spent on each.

Conformance — the more important result:

Corpus	Direct-coded rules	Published DFA
UCD grapheme, word, sentence	100%	100%
UCD <code>LineBreakTest</code>	19,309 / 19,346 (99.81%)	19,346 / 19,346 (100%)
ICU <code>rbbitst.txt</code> , line	176 / 240	177 / 240
ICU <code>rbbitst.txt</code> , sentence, root locale	32 / 33	33 / 33

The DFA reached full conformance on data I had been failing for years, from tables plus a generic engine, with no rule transcription at all.

Performance — 1,800 bytes of Latin text, milliseconds per pass:

Break	Direct-coded	DFA	DFA relative
line	1.529	0.777	51% of the time
grapheme	0.614	0.403	66%
sentence	0.336	0.311	93%
word	0.420	0.539	128%

Line breaking takes roughly half the time, grapheme breaking about two thirds, sentence breaking is a wash, and word breaking is about 30% slower. Memory allocation follows the same pattern — the DFA allocates $2.8\times$ less for line breaking and $1.6\times$ more for word breaking.

An important qualification. Those figures are *after* giving the DFA the same two optimisations my direct-coded engine already had: a table mapping Latin-1 codepoints directly to symbols instead of binary-searching the ranges, and a precomputed table answering, for every pair of ASCII characters, whether the machine breaks between them. Without those the DFA was three to four times *slower* than the direct-coded engine. The published tables are not fast by themselves; they are a structure that can be made fast, and an implementer should expect to do that work. I think that this as a fair trade, because the tuning is generic — it is the same two tables for all four break types and it does not have to be revisited when the rules change — whereas tuning a direct-coded engine means touching rule logic.

Readability and traceability improve a lot. This is a strong positive and, for me, on a par with the conformance result. A defect in a table-driven engine is a defect in one of three data files or in a hundred lines of shared engine; a defect in a direct-coded engine can be anywhere in thirty interacting rules, and — as section 3 shows — can sit undetected for years.

It is adopted and released. All four break types dispatch to the published tables in my `unicode_string` library version 2.4. Two things had to be built around the tables to make that work: the machine answers *where is the next boundary* but not *is there a boundary at this position*, so a boundary predicate has to re-scan from the previous boundary; and a compiled symbol table has nowhere to put a CLDR locale tailoring, which is issue 5h.

The tables also insulate against churn below them, which is a benefit worth claiming that proposal does not. Between adopting the tables and now, the two libraries I build on both changed substantially: the property library gained `@missing` defaults for fifteen further properties, so every property's values now account for all 1,114,112 code points where five did before, and the UnicodeSet library realigned to draft UTS #61, changing quoting, octal and control escapes, `[-]`, string members and the semantics of `Age`. I re-derived the symbol tables across both changes and compared 392,490 boundary decisions per break type — a sweep of the whole code space against six partner characters — and the results are digest-identical. The proposal argues that adopting a new Unicode version becomes a data update; this says the tables hold steady against movement in the layers an implementer stands on as well, which is perhaps the harder half of the maintenance problem.

5. Specific issues encountered while implementing

5a. The pseudocode is underspecified where the data defines no `eot` symbol

`{eot}` appears in `SentenceBreakSymbols.txt` and `LineBreakSymbols.txt` but not in `GraphemeClusterBreakSymbols.txt` or `WordBreakSymbols.txt`. Section 9.3.2 unconditionally executes `Symbol_Ahead :=` The symbol containing the string "eot" when the position reaches the end of the document, which has no reference for two of the four machines. I had to invent the termination rule, and my first attempt was wrong in a way the UCD corpus caught only for word breaking.

It would be very helpful to state what an implementation should do at end of text when the data defines no `eot` symbol.

5b. `Last_Accepting_Position` can be read before it is written

`Last_Accepting_Position` is declared without an initial value. If the first symbol encountered has no transition from `START`, line 33 assigns from it having never set it. The sensible behaviour is that a segment is at least one character — but it must be one *character*: my first implementation floored it at one code unit, which splits a multi-byte UTF-8 sequence and corrupts the remainder of the string. Updating 9.3.2 would prevent both mistakes.

5c. Dictionary symbols identify that a dictionary is needed, but not which one

This is the issue with the most practical consequence for me.

`WordBreakSymbols.txt` defines `XX_Hiragana|XX_Han|XX_ComplexContext` as a single symbol spanning Hiragana, Han and `Complex_Context`; `LineBreakSymbols.txt` merges every `Complex_Context` script into `SAmMnmMc` and `SA_Mn|SA_Mc`. But Chinese/Japanese, Thai, Lao, Khmer and Burmese are five different dictionaries (in UCU - which is where I sourced the dictionaries I'm using in my implementation). On encountering a dictionary symbol an implementation learns that dictionary-based breaking is required and must then re-derive the script itself to discover which dictionary to load — which is exactly the classification work the symbol table otherwise performs.

I would suggest either splitting dictionary symbols so that each corresponds to one dictionary, or adding a field naming the dictionary required. The existing third field, the non-dictionary equivalent symbol, is well judged and I made use of it; this is a request to extend it rather than to change it.

5d. The `UnicodeSet` subset used by the symbol files should be specified

The symbol definitions use string elements (`{eot}`), `{L00SE_IN}`), chained difference (`{\p{A}-\p{B}-\p{C}}`), POSIX-style properties with spaces around the equals sign (`[:LineBreak = Complex_Context:]`) and boolean properties with an explicit value (`{\p{Extended_Pictographic=True}}`). My `UnicodeSet` implementation supported some of these and not others; three separate fixes were required before a single symbol file would parse end to end.

Since the proposal's purpose is to lower the barrier to conformant implementation, it would help to state normatively which subset of `UnicodeSet` a consumer must support. Alternatively, moving the end-of-text marker and the shadow symbols out of set syntax into their own field would remove the requirement for string elements entirely, which is the least widely implemented of the four.

Thankfully there is now the [UTS #61 Unicode Set Notation](#) draft. The proposal could cite UTS #61 at a named version and state which part of it a symbol file consumer must implement.

Naming the part matters, for two reasons visible in the draft itself. It marks version qualifiers, property comparisons and regular expression matching as supported only in the Consortium's own tools and not recommended for general-purpose APIs, so “implements UTS #61” is not the bar an implementer of these

tables needs to clear, and saying so would save them work. And it proposes moving `\N{...}` out of standalone sets into range elements, which the draft notes breaks compatibility with some ICU versions — so the notation is still moving, and a bare reference without a version would age badly.

The draft also states that it “mostly follows the ICU4C implementation”. That is worth being explicit about in the proposal too, because it tells an implementer without ICU what they are actually being asked to reproduce.

5e. Publish symbol and state indices

The document states that names are opaque and “would likely be replaced by indices in any optimized implementation”. I did exactly that. Publishing the indices alongside the names would spare every implementation inventing its own interning scheme, and would make the tables diffable between versions — worth having, given that 3,405 of the 7,592 line-break transitions changed between 17.0 and 18.0.

5f. The tables are already ahead of the prose, which argues for the proposal

The Version 18.0 grapheme tables encode the revised GB9c, under which a linker opens a conjunct sequence with no preceding `Indic_Conjunct_Break=Consonant`. The draft UAX #29 states plainly that “the regular expression(s) need to be updated to reflect the change to Rule GB9c. This was not done in time for the Unicode 18 beta”. My implementation followed the prose and was wrong; `GraphemeBreakTest-18.0.0.txt` agreed with the tables.

That is the proposal’s own argument demonstrated on a live example, and maybe it’s worth citing it. It also raises a question worth answering explicitly: when the tables and the prose disagree, which is normative?

5g. For dictionary-based scripts, the dictionary dominates and the standard says little about it

Once a script requires dictionary lookup, the rule engine stops being the bottleneck and the choice between rules and tables stops mattering much. Two measurements from this work:

- Adding a cheap pre-filter to skip the dictionary pass for text that cannot contain a dictionary script — the four scripts occupy U+0E01..U+17FF, so a single scan for UTF-8 lead bytes `0xE0 / 0xE1` excludes them — made line breaking **1.85× faster** in *both* implementations. The dictionary pass had been costing more than all of the line-breaking rules combined, on text containing no dictionary script at all.
- For Japanese word breaking the DFA’s advantage narrows to 1.9×, because both implementations spend most of their time in trie lookups rather than in rule dispatch.

So the benefit of the proposal is real but concentrated in rule-driven segmentation. For CJK and South-East Asian text, what limits a conformant implementation is not the rules but the dictionary: when to trigger it, what span to hand it, and which dictionary to use. Section 9.3.4 addresses triggering, and the non-dictionary equivalent symbol is a good mechanism, but the scope of a dictionary run and the choice of dictionary are left to the implementer. Issue 5c is the concrete form of this gap. Anything the the proposal can add here would help implementations outside ICU more than further work on the state machines themselves.

One concrete example of what the silence costs. A dictionary hands back the spans it found, and it is left to the implementer to decide which of those span edges are breaks. Taking them all is wrong: the rules had already produced the surrounding segment as a single unit, so a boundary at the edge of a dictionary run can contradict them. (followed by Thai is the case that showed it — LB14 forbids a break after an opening bracket, but the dictionary returns the bracket as its own span, and emitting that span as a segment breaks where the rules do not. The rule I settled on is that a dictionary may only break between two characters of its own script, and everything else defers to the rules; I would have taken that from the standard had it been stated. Section 9.3.4 says when to call a dictionary but not what to do with what it returns.

5h. A compiled symbol table has nowhere to put a locale tailoring

CLDR tailors the break classes per locale. `el.xml` adds U+003B and U+037E to `$STerm` so that Greek breaks at the erotimatiko; `ja.xml`, `zh.xml` and `zh_Hant.xml` move

`Line_Break=Conditional_Japanese_Starter` out of `$NS` and into `$ID`, the tailoring usually called CJK loose line breaking. An implementation that reads the CLDR segment rules applies these by substituting a variable before it compiles anything. An implementation that consumes the published tables cannot: the symbol table is fixed when the tables are generated, and the locale is not known until a call is made.

I worked around it by rewriting the tailored characters to standard characters carrying the class the locale wants, before the machine runs — U+003B to U+0021 for Greek, each conditional Japanese starter to a CJK ideograph — choosing substitutes that encode to the same number of UTF-8 bytes so that every offset the machine reports still indexes the original string and segments can be sliced from it untouched. It works, and it costs one pass over the input. But it is a trick rather than a mechanism, and it holds only because a same-length substitute happens to exist for each tailoring I need.

Publishing per-locale symbol variants alongside the root tables would be the complete answer, at the cost of a good deal more data. Short of that, stating in 9.3.2 how a consumer is expected to apply a CLDR class tailoring to a compiled symbol table would at least make the intended approach explicit. At present the proposal is silent on it, and an implementer could reasonably conclude that tailoring is out of scope — which, for line breaking in Japanese, it is not.

6. Implementations outside ICU

Section 2 argues that the cost of a conformant implementation falls mostly outside the segmentation algorithm itself. The state of the ecosystem bears that out, and the three layers are in very different shape.

6.1 Segmentation

Erlang/OTP is the sharpest illustration I found of the failure mode described in section 3, and it is not an implementation that can be dismissed as unmaintained or behind.

OTP's `unicode_util` module reports `spec_version() =:= {17, 0}` — the current version of the standard (until this week). It implements GB9c: the generated module contains a section headed `%% Handle Indic Conjunct Break`, with `is_indic_linker/1` and `is_indic_consonant/1` predicates. The rule is present,

correctly shaped, and current.

It nevertheless disagrees with UAX #29, because of where those two predicates come from. The generator, `gen_unicode_mod.escript`, derives them from `IndicSyllabicCategory.txt`:

```
Linkers = maps:get(virama, GBP) ++ maps:get(invisible_stack, GBP),
Consonants = maps:get(consonant, GBP) ++ maps:get(vowel_independent, GBP) ++ [{16#1B0B, 16#1B0C}
```

GB9c is defined over `Indic_Conjunct_Break`, not over `Indic_Syllabic_Category`. `InCB=Linker` contains 23 codepoints; every Indic virama gives 41. The consequence is that OTP joins conjuncts in scripts where the UCD does not: U+0CCD KANNADA SIGN VIRAMA is `InCB=Extend`, so `String.graphemes/1` returns one cluster for a Kannada conjunct where UAX #29 returns two. Gurmukhi, Tamil and Sinhala behave the same way. Devanagari, Bengali, Gujarati, Oriya, Telugu and Malayalam agree, because their viramas are in both sets.

`Indic_Syllabic_Category=Virama` is an entirely reasonable-looking place to find viramas. It is simply not the property the rule is defined over, and nothing in the annex’s prose makes that mistake conspicuous. This is the same category of failure as five of my own eight defects, in a mature, current, widely-deployed implementation — and it is exactly the failure a published symbol table makes impossible, because the symbol definition names the property rather than describing the class in prose.

I offer this as a case you may find useful, since it does not depend on an implementation being (much) behind the current version.

6.1.1 Other implementations

I was very curious to see what the state of implementation in other language systems looks like. The results below are the result of web search alone and did not involve anything except reading documentation (no testing or verification was done).

Library	Ecosystem	Scope	Unicode version
@cto.af/linebreak	JavaScript	UAX #14	17.0
SCKelemen/unicode	Go	UAX #9, #11, #14, #24, #29, #31, UTS #15, #39, #51	17.0
unicode-segmentation	Rust	UAX #29 only	17.0
libunibreak	C	UAX #14, UAX #29	15.0 line breaking, 15.1 word and grapheme
rivo/uniseg	Go	UAX #14, UAX #29	15.0
uax-14	Common Lisp	UAX #14	12.1

I think the assertion in *Why do we need this in the Standard?* — that the authors are “unaware of any conformant performant open-source implementation of the line breaking algorithm outside the Consortium since Unicode Version 13.0 (2020)” — is stated more strongly than the evidence supports.

`@cto.af/linebreak` states that it “intends to be fully conformant with UAX #14” and “currently passes ALL of the tests published by Unicode” at Unicode 17.0, and `SCKelemen/unicode` reports “UAX #14: 100% conformance (19,338/19,338 tests)” against the official UCD test files, that figure being exactly the number of cases in `LineBreakTest-17.0.0.txt`. Neither depends on ICU. Neither publishes performance measurements, so the “performant” half of the sentence is untested in both cases; the conformance half does not survive as written.

The argument survives the correction comfortably, and the OTP case above supports it independently of any implementation’s version currency.

6.2 UnicodeSet — effectively absent outside ICU

I could not find a general-purpose UnicodeSet implementation outside ICU, ICU4X and my own. Several ecosystems offer Unicode *property matching* in their regular expression engines — Rust’s `regex`, Python’s `regex`, Java and .NET all accept `\p{...}` — but property matching is not UnicodeSet. None of them offers the set algebra the notation requires: nested sets, intersection and difference with TR35 precedence, and string elements.

This matters directly to the proposal, because the symbol files are written in UnicodeSet. An implementer without ICU must either build UnicodeSet support first, or the data files must stay within a subset that a regular expression engine’s property matching can evaluate. That is the practical form of issue 5d.

UTS #61 may improve this in time, since a normative definition is a great deal easier to implement against than a syntax distributed across two other reports and a reference implementation. It does not change the position today, though, and the tables are proposed for publication now.

6.3 UCD introspection — partial nearly everywhere

Standard libraries generally cover the common properties and stop short of the ones segmentation needs. Python’s `unicodedata` provides general category, bidi class and decomposition, but not `Line_Break`, `Word_Break`, `Sentence_Break` or `Indic_Conjunct_Break`. Go’s `unicode` package provides categories and scripts but not the segmentation classes. Rust distributes the properties across a family of separate crates. Java’s `Character` is similarly partial.

An implementation of either annex therefore begins by building property coverage that the standard assumes is already available.

6.4 My own implementations

The measurements in this document come from these libraries. All three are open source under the Apache 2.0 licence.

Library	Scope
<code>unicode</code>	UCD introspection: enumerated and binary properties, scripts, blocks, categories
<code>unicode_set</code>	UnicodeSet expressions per TR35, UTS #18 Level 1 and draft UTS #61, tested against the ICU reference implementation
<code>unicode_string</code>	UAX #14 and UAX #29 segmentation, driven by the tables this proposal publishes

The implementation of this proposal, the two corpora used for the conformance figures, and the benchmark comparing it with the engine it replaced are all in `unicode_string`.

7. Closing

I have adopted the published tables and shipped them, on the strength of the conformance result and improved maintainability alone; the performance improvement for line breaking is a bonus.

If it is useful, I would offer one further line of support alongside the maintenance argument the proposal already makes. Both pieces of evidence I have been able to gather point the same way: my own implementation tracked the current version and still evaluated five rules over the wrong sets of characters, and Erlang/OTP is current, implements GB9c, and derives its inputs from `Indic_Syllabic_Category` where the annex specifies `Indic_Conjunct_Break`.

Neither of those is a failure of care. They follow from the annexes describing character classes in prose, leaving the reader to reconstruct a set expression. The symbol files do not leave that to the reader, and for me that is as valuable as the state machines themselves.