Bidirectional Domain Names

Steven Atkin
IBM, Austin, TX
atkin@us.ibm.com

Ryan Stansifer
Florida Tech, Melbourne, FL
ryan@cs.fit.edu

Mohsen Alsharif
Florida Tech, Melbourne, FL
alsharif@usa.com

Abstract

Unicode's ability to represent multilingual text makes it a good candidate for establishing the basis for a domain name structure. Unicode brings not only an encoding framework, but also support for things like bidirectional scripts. The collection of Unicode's character equivalences is both desirable and at times necessary given Unicode's goal of encoding natural language text. These equivalences, however may present problems in the context of domain names.

Unicode's Bidirectional Algorithm as currently specified is unsuitable for determining an appropriate display ordering for multilingual domain names. The Bidirectional Algorithm possesses a set of implicit assumptions about the usage of common characters that are not applicable to domain names. Domain names use the same repertoire of characters that appear in text, but requires a different algorithm for handling them in domain names.

In this paper we propose how domain names can accommodate different reading orders. In particular, this paper offers an algorithm for determining the display order "reading" of multilingual domain names. Additionally, we relate this notion to Unicode's Bidirectional Algorithm.

Keywords
Domain Names, Unicode, Bidirectional Data, Multilingual Display

I. INTRODUCTION

The transition from the now ubiquitous monolingual ASCII based domain name system to a truly multilingual extendable system has been long awaited [3]. Indeed, it may have already begun without waiting for standards [2]. This move brings the dream of the multilingual web one step closer. Nevertheless, this transition must be approached cautiously as decisions made today may have long lasting effects.

These decisions include the set of characters for constructing names, the base character encoding, and the code point transmission protocol. Nonetheless, there are certain constraints that must be honored regardless of these decisions. For example, domain names that are legal today must still remain legal in the new domain name system.

The most natural starting point for choosing the allowable set of characters from which domain names may be constructed is to start with the character repertoire available in Unicode/ISO10646 [4]. The range of characters available in Unicode is vast and accommodates most modern written scripts. In contrast to ASCII, this includes scripts such as Arabic, Farsi, and Hebrew. On the surface extending the current domain name system may not seem to be much of a challenge, given that all we are doing is adding more characters. However, unlike ASCII which only encodes scripts written left-to-right, Unicode encodes scripts written right-to-left as well as those written left-to-right. It may well be necessary to combine characters from different scripts. However, when these scripts are intermixed their display becomes uncertain, due to the conflicting directions.

In creating a new domain name system display ambiguities cannot be tolerated. The display of domain names cannot simply be left totally to the discretion of the user or application. This would certainly lead to confusion. Unfortunately, this problem has already occurred in the display of bidirectional natural language text [1]. In order to alleviate this situation an algorithm must be created that guarantees that there are no such ambiguities. Additionally, this algorithm must be both simple to understand, easy to implement, and inexpensive to execute.

This paper presents an algorithm for unambiguously determining the display order of bidirectional domain names. This paper will not delve into all aspects of creating domain names. In particular we do not discuss encoding of Unicode into domain name octets. One obvious strategy is UTF-8, but other encoding forms may be more beneficial. The paper starts by examining the class of characters that should be included along with those that should be excluded from bidirectional domain names. This is followed by an analysis of the current approaches for displaying bidirectional data. The bidirectional domain name algorithm is then presented.

In order to simplify comprehension of the examples in this paper. The following convention is used: lowercase Latin let-

ters a-z indicate Latin letters, uppercase Latin letters A-M represent Arabic letters, uppercase Latin letters N-Z represent Hebrew letters, the digits 0-4 indicate European numerals, the digits 5-9 indicate Arabic numerals, and the hyphen-minus (European terminator) is represented by "—". See Table1. This is the same convention used by Unicode to discuss the input and output of the Unicode Bidirectional Algorithm [5].

Table 1: Bidirectional character mappings

Direction Type	Mapping
L	a-z
AL	A-M
R	N-Z
AN	5-9
EN	0-4
ET	—

II. Proposed Domain Name Character Set

The richness of characters available in Unicode is certainly an asset when used to encode natural language text. Nevertheless, this richness is something that is not necessarily desirable when encoding domain names. The various ways in which characters can be constructed in Unicode, precomposed and decomposed makes the representation of domain names unnecessarily complex.

This complexity presents two significant problems for encoding domain names, name registration and name equivalence. Historically these have never been a problem, because it made no difference whether the registration of a domain name was based upon characters or code points. In ASCII there is no distinction between characters and code points, however in Unicode such a distinction becomes necessary at times.

In Unicode, characters that contain diacritic marks may be represented in two ways, precomposed form and decomposed form. Characters in precomposed form are represented by a single code point, while characters in decomposed form are constructed from multiple code points. For example, the latin capital letter u with diaeresis and acute can be encoded. See Figure 1, lines 1-3. In all cases the same visual output is produced irrespective of the sequence of code points. [4]

Figure 1: Latin capital letter u with diaeresis and acute

Ű U01D7 Ű (1)

Ű U00DC,U0301 Ü ´ (2)

Ű U0055,U0308,U0301 U ¨ ´ (3)

This has a big impact on the clear representation of data and especially domain names. If domain names are registered by characters and not by code points then domain name servers or clients will be required to perform some form of normalization. If domain names are registered via code points then normalization becomes a non problem. On the other hand, it forces the registration of multiple names that really represent the same name.

To complicate matters Unicode also encodes some characters that are merely glyph variants of other characters. This situation also requires some form of normalization. For example, the two character sequence "fi" may be represented in two ways in Unicode. See Figure 2. Line 1 on Figure 2 encodes the "fi" sequence using a single code point, while on line 2 the "fi" sequence is encoded using two code points. In either case both character sequences encode the same semantic content. The only difference being the glyph used to render the sequence.

Figure 2: The fi ligature

UFB01 fi (1)

U0066,U0069 fi (2)

In order to simplify the construction of domain names the authors recommend that decomposed characters only be used in cases where there is no corresponding precomposed character, Unicode Normal Form C [6]. This greatly simplifies the task of determining name equivalence, as each domain name has a unique representation. Additionally, those characters that are glyph variants of other characters (compatibility characters) should not be used in domain names either. At first this may seem too restrictive, however this is nothing more than an artificial restriction. The authors argue that there is no need for compatibility characters, as domain name distinction is not based upon visual appearance. Naturally, some may argue that these characters are necessary for legacy data conversion. This is not a concern, to domain names as they are encoded in ASCII now.

The authors view multilingual domain names simply as an extension of the current domain name character set [3]. In keeping with this strategy only additional letters and digits are added. The Ayna domain name registration system, however greatly restricts the characters that can be used in domain names [1]. Their registration system only allows European numerals and does not permit the intermixing of different script systems within a domain name[1]. Nonetheless, our approach does not have such limitations.

Control codes are excluded from domain names today (sensibly enough) there is no reason to include them in multilingual domain names. These include the bidirectional controls as well (LRE, LRO, LRM, RLE, RLO, RLM, and PDF) [5]. The purpose of these controls is to override the behavior of Unicode's Bidirectional Algorithm. In most situations Unicode's Bidirectional Display Algorithm produces acceptable results when rendering natural language text. The use of the controls is only required in the rarest of situations, and thus their elimination outweighs any potential benefits.

1. Information about Ayna can be found at http://registrar.ayna.com/ayna_html

Naturally the set of allowable domain name characters must expand to include Arabic and Hebrew letters, however Unicode has many code points for the Arabic writing system and the Hebrew writing system. Not all of these code points are required in the context of domain names.

There are a number of Arabic characters that can be safely excluded from domain names. Specifically, these include the Arabic presentation forms, UFB50-UFDFF and UFE70-UFEFC. It is safe to exclude these characters, as they only represent ligatures and glyph variants of the base nominal Arabic characters. Additionally, the Arabic points U064B-U0652, U0653-U0655, and U0670 should also be excluded. In most cases the Arabic points are only used as pronunciation guides. If the points were to be included, then names that differed only in their use of points would be treated as if they were distinct and different names. This is like the English homograph "bow" (the arrow) and "bow" (the ship) which are ambiguous. Removing the Arabic points eliminates such problems, with the understanding that not every Arabic word would be able to be represented. The Koranic annotation signs U06D6-U06ED can also be eliminated from domain names, as they are not used to distinguish one name from another.

In Hebrew the cantillation marks U0591-U05AF and Hebrew points UFB0-U5C4 can be excluded as they are predominately used as pronunciation guides and for indicating the underlying structure of text. Additionally, the Arabic and Hebrew punctuation characters are also excluded from domain names as they are currently not permitted. The list of acceptable Arabic and Hebrew characters are listed in Table2.

Table 2: Acceptable Arabic and Hebrew characters

Unicode Range	Script	Notes
U05D0-U05F4	Hebrew	ISO8859-8
U0621-U064A	Arabic	ISO8859-6
U0660-U0669	Arabic	Arabic-Indic digits
U0671-U06D3,U06D5	Arabic	Extended Arabic letters
U06F0-U06FE	Arabic	Persian, Urdu, and Sindhi

III. Bidirectional Text in Domain Names

Unicode's ability to intermix the various script systems of the world makes the creation of multilingual documents no more difficult than the creation of monolingual documents. This new found freedom, however does come with a cost. When various script systems are intermixed their display (generally) becomes unclear. We consider the type left-to-right (English, etc.) and the typically right-to-left writing system Arabic.

Unicode provides an algorithm for determining the appropriate display order given an arbitrary sequence of characters in logical order. The algorithm is based upon a set of implicit heuristics along with a set of explicit control code overrides. These control codes are used in cases where the implicit rules do not yield an appropriate display order. [5]

Naturally one would assume that since Unicode characters are going to be used in domain names then Unicode's Bidirectional Algorithm should also be used. Upon closer examination it becomes apparent that this position is inappropriate. The input to Unicode's algorithm carries with it a set of assumptions. The primary one being the input is natural language text in general. This assumption, however is not necessarily true in the case of domain names. A domain name does not resemble a paragraph of multilingual text. Different assumptions apply. This contextual difference causes several problems when one attempts to apply the Unicode Bidirectional Algorithm to domain names.

The first problem to be encountered is the use of the full stop character, U002E in domain names. When a full stop occurs in natural language text its purpose cannot be immediately determined. The meaning is dependent upon the context in which it is used. It may indicate the end of a sentence, an abbreviation, or even a floating point number. See rules W4 and W5 in Unicode Standard Annex #9 [5]. When a full stop, however is present in a domain name its meaning is clear. The meaning of the full stop never varies across domain names. The full stop always serves to separate a domain name into its individual parts or "labels". Furthermore, the full stop establishes the hierarchy of the individual labels. In domain names there is a strict hierarchy regarding the ordering of the labels. The most general part of the domain name is always the right-most label, while the most specific part of the name appears as the left-most label. This requires a domain name to be read in a general left-to-right direction.

When the Unicode Bidirectional Algorithm's rules are applied to text, it is done on a per paragraph basis. Each paragraph is rendered independently of each other. Unfortunately, when the Unicode Bidirectional Algorithm is applied to domain names each label is not rendered independently of the the others. Each label may influence the rendering of the others. The authors claim the full stop character should act as if it were the start of a new paragraph in the context of domain names. Additionally, each domain name should be rendered in an overall left-to-right reading direction so as to preserve label hierarchy.

The Unicode Bidirectional Algorithm determines the general reading direction of a paragraph in one of two ways. The first method is based upon a higher order protocol explicitly stating the reading direction. The second makes use of an implicit rule whereby the first strong directional character determines the overall reading direction. In this context the term "strong" indicates a character that is either a left-to-right character or a right-to-left character. This implicit rule, however causes problems for rendering domain names. This is illustrated in Figure 3.

The text on Line 1 of Figure 3 is a domain name in logical order. Line 3 is the corresponding output from the Unicode Bidirectional Algorithm. In this example the presence of an Arabic character in the first label forces the entire domain name to take on an overall right-to-left reading. This unfortunately mangles the hierarchical structure of the domain name. It is no longer possible to universally determine which label is the most specific and which is the most general. Some may argue that if the overall reading direction is known, in this instance right-to-left, then the hierarchy of the individual

labels can be determined. This statement is not true in multilingual domain names, however.

In many cases it is impossible to tell the overall reading direction by merely looking at the output. It turns out that it is possible to obtain the same output "display order" given two distinct inputs in logical order. In this example the input on lines 1 and 2 produce the same output on line 3. In this case the most specific part of the name on line 1 is "ABC", while on line 2 it is "ibm". This does not indicate that there is a flaw in Unicode's algorithm, rather it only further illustrates the hidden assumptions concerning the intended use of the Unicode Bidirectional Algorithm.

Normally in natural language text processing this is not a problem given that the two can be distinguished by their physical justification on the screen, either right or left. This luxury, however is not afforded to domain names. When a domain name appears in printed text there is no generally accepted way to indicate the overall reading direction.

Nonetheless, some may argue that if the entire domain name is in Arabic then the label hierarchy should be reversed. The problem in adopting this strategy occurs when the entire domain name is not from the same script, as is the case in this example. The authors suggest that the output on line 4 in Figure 3 is more desirable. This output is consistent with the current structure of domain names. In this case the full stop characters are ignored, and the Bidirectional Algorithm is applied to each of the individual labels of the domain name. Naturally one might assume that Unicode's Bidirectional Algorithm may still be appropriate, given that it is run independently on each of the individual labels. This strategy also presents problems, however.

Figure 3: Using a full stop in a domain name

ABC.ibm.com (1)

ibm.com.ABC (2)

ibm.com.CBA (3)

CBA.ibm.com (4)

The problem with this approach involves the use of the hyphen-minus character "-", U002D. In the Unicode Bidirectional Algorithm the hyphen-minus is assigned to the European Terminator character class. Unfortunately, this causes the character to behave as if it were an European numeral when adjacent to European numerals. See rule W5 in Unicode Standard Annex #9 [5]. This behavior may be acceptable when processing natural language, but is unacceptable when processing domain names. In domain names the predominant usage of the hyphen-minus is as white space and not as an European terminator. The example in Figure 4 illustrates the effect of European digits surrounding the hyphen-minus characters.

Line 1 on Figure 4 is a single domain name label in logical order. Line 2 is the same label in display order, this is the output of the Unicode Bidirectional Algorithm. The text on Line 3 is also in display order, however this output is obtained when the hyphen-minus characters are treated as white space characters.

Figure 4: Using a hyphen minus in a domain name

NOP--123 (1)

--123PON (2)

123--PON (3)

The last remaining problem occurs when an individual label contains characters with varying directions. In this situation the reading order of a label may become ambiguous. This is illustrated in Figure 5. Line 1 in Figure 5 is an individual label in display order. Unfortunately there are two possible readings "logical order" associated with this output, lines 2 and 3 in Figure 5. If however we assume that in this mixed case a label always takes a general left-to-right reading then there is only one possible reading. The authors contend that this policy is consistent with the overall left-to-right reading of a domain name. Nevertheless, the Unicode algorithm still maps the two logical inputs to the single display output even when the overall reading direction is fixed to left-to-right (higher order protocol). This situation potentially causes problems for domain name resolution.

Figure 5: Label with varying directions

abcFED (1)

abcDEF (2)

DEFabc (3)

The authors believe that domain name registration will be made in logical order. This policy is consistent with how bidirectional data is generally stored in files today. If we permit the Unicode Bidirectional Algorithm to be used for the display of domain names, then there may be situations when a domain name cannot be resolved even when it appears to be entered correctly. One solution to this situation is to register multiple logical names that yield the same display order. The authors argue that a better approach is to create an algorithm that is one-to-one. In this algorithm each display order is mapped to one and only one logical input and each logical input is mapped to one and only one display output. This policy comes with some associated cost, however. There maybe cases where the reading may seem unnatural. The authors believe that this will occur infrequently and that the benefits outweigh any potential misreading.

IV. Algorithm for Domain Names

The authors believe the primary goal of the domain name display algorithm is to unambiguously represent multilingual

domain names. There are additional goals, however that the authors judge are necessary for a successful solution:

- The algorithm must provide a one-to-one mapping between names in logical order and names in display order.
- The output should be consistent with Unicode's Bidirectional Algorithm when possible.
- The algorithm should be easy to understand and simple to implement.
- The algorithm should not require any form of normalization.
- The algorithm should minimize impact to the current DNS architecture.
- Maximize the readability of multilingual labels.

As we have seen Unicode's algorithm is inappropriate, because different inputs give the same output and assumptions about syntax and punctuation are inappropriate for domain names.

Our algorithm is divided into two phases, inferencing and reordering. Inferencing resolves the direction of indeterminate characters (full stop, hyphen-minus, Arabic numeral, and European numeral). During this phase each character is assigned a strong direction, either left or right. The reordering phase takes the fully resolved characters and generates a display ordering for them.

The inferencing phase is accomplished in several passes. Implementers may wish to optimize this phase. In the first pass Arabic and Hebrew letters are assigned the right-to-left direction, while full stops and other alphabetic characters are assigned the left-to-right direction. The next set of passes resolves the directions of digits.

There are two rules for resolving the direction of Arabic and European numerals. All Arabic numerals are assigned the right-to-left direction. European numerals are assigned the left-to-right direction, unless the European numeral is surrounded by right-to-left characters (Arabic or Hebrew letters), in which case it takes the right-to-left direction. This is accomplished in two passes — a forward pass and a reverse pass. The final set of passes resolves the directions of hyphen-minus characters.

There are two rules for the resolution of hyphen-minus characters. All hyphen-minus characters become left-to-right, unless the hyphen-minus is surrounded by characters whose direction is right-to-left in which case the hyphen-minus becomes right-to-left. This is the same resolution as digits, but occurs after digit resolution. At this point each character in the domain name has a strong direction.

The reordering of the resolved characters makes use of a few simple data structures:

- The digit accumulator — holds a sequence of European or Arabic numerals that have a right-to-left direction.
- The character stack — holds Arabic letters, Hebrew letters, and sequences of digits.
- The mode variable — keeps track of the current direction.

The algorithm makes use of a few simple operations on these data structures:

- The clear operation — outputs each digit from the accumulator, then outputs each character from the character stack, and finally outputs the current character. After this operation the digit accumulator and the character stack are empty.
- The empty operation — outputs each character from the character stack, then outputs each digit from the accumulator, and finally outputs the current character. After this operation the digit accumulator and the character stack are empty. Empty is like clear, but the order of operations is reversed.
- The push operation — Pushes the contents of the digit accumulator onto the character stack, and then pushes the current character onto the stack. After this operation the accumulator is empty.
- The accumulate operation — appends the current character onto the digit accumulator.

The algorithm for reordering is:

- Current character direction is left-to-right (includes European numerals with a left-to-right direction).
  a. If mode is left-to-right, then "empty" else "clear".
  b. Set mode to left-to-right.
- Current character direction is right-to-left and character is not a digit.
  a. Perform "push".
  b. Set mode to right-to-left.
- Current character is a numeral (European and Arabic) with a right-to-left direction.
  a. Perform "accumulate".
  b. Set mode to-right-to-left.

At the end of the input stream if the mode is left-to-right, then "empty" else "clear".

The bidirectional domain name display algorithm converts a string of characters in logical order to a string of the same length in display order. In fact the algorithm is its own inverse, in other words $A(A(x))=x$. Hence A is a one-to-one function.

To see that this is the case we make the following argument. First, it is obvious that the function A loses no characters, so the output is a string of the same length and a permutation of the original characters. Second, all left-to-right runs (including full stop and certain hyphen-minus characters), are preserved in exactly their original positions. Third, all right-to-left runs are permuted within their own run. No characters "leak", "flop" or move to another run and the right-to-left runs are preserved in their same order. Finally, the right-to-left runs are reversed (approximately).

The nature of reversing right-to-left runs requires further explanation as the numerals (Arabic and European) complicate the matter. Consider the logical right-to-left run on line 1 in Figure 6 and its corresponding display on line 2 in Figure 6. The output on line 2 is a string reversal treating digits as units.

Hence, this sort of reversal is its own inverse. Therefore, the whole algorithm is its own inverse.

Figure 6: String reversal

AB12CDE678FGHI (1)

IHGF678EDC12BA (2)

This algorithm can be used to accommodate two different groups of domain name creators. One group know what they want to register, but are unsure how it will be displayed. On the other hand, there are creators who know what they want to see displayed, but are unsure what logical sequence of characters should be registered. This single universal algorithm addresses both of these situations. This eliminates the need for specialized individual algorithms (logical to display and display to logical).

V. Display Order

In this section we provide sample input (logical order) along with corresponding output (display order) of domain name labels. This set of input represents some of the numerous ways in which domain name labels can be created. In the following tables we do not use entire domain names such as *www.label.label*.com. This is unnecessary, as each label is rendered independently of the others.

The output was tested for readability by a number of native Arabic readers (at Florida Tech). These readers are from various countries (Saudi Arabia, Libya, Egypt, and Lebanon) and represent a wide audience of potential domain name creators.

The sample test cases are divided into two classes. One class contains sequences of letters and letters with hyphens. The other class contains sequences of letters, digits, and hyphens. All of the sample input tables contain three columns. The logical column contains the logical sequence of typed characters, the display column contains the output from the domain name display algorithm, and the comment column indicates the type of the logical input sequence. An explanation of these types can be found in Table1. Table3 and Table 4 contain labels form the first and second input classes respectively.

We anticipate that most Arabic domain name creators will construct labels that are comprised of Arabic letters, Arabic numerals, European numerals, and hyphen-minus characters. On the other hand, we foresee Hebrew domain name creators constructing labels that are solely comprised of Hebrew letters, European numerals, and hyphen-minus characters. Naturally, we do not expect Latin based labels to dramatically change. We consider the correct output in these cases to be essential.

There are other inputs that are “contrived” or “artificial”. These contrived inputs are cases where it is difficult to determine an appropriate display. Such situations include the intermixing of right-to-left and left-to-right characters with Arabic and European numerals. These cases are discussed here to illustrate the behavior of the algorithm, and also to examine trade-offs made during the construction of the algorithm.

Certainly, the algorithm must correctly display both legacy domain names and single script domain names. Legacy conformance is illustrated in test case 1 in Table3 and test case 1 in Table4. In the case of single script (Arabic or Hebrew) label, it is essential that their display be consistent with user expectations. This is shown in test cases 2 and 3 in Table3.

Looking at the contrived cases, in particular test case 5 in Table3. This test case consists of Arabic letters (right-to-left) followed by left-to-right characters. Normally, a reader would expect the Arabic characters to appear in the display at the right most end of the label. This is the output that Unicode’s Bidirectional Display Algorithm yields. Our algorithm does not generate this display order as it requires the adoption of rules that cause the algorithm to no longer remain one-to-one. This problem was explored earlier. See Figure 5. Our algorithm always generates a display order that has an implicit left-to-right embedding. The authors claim that this is an acceptable trade-off given the goal of creating a one-to-one algorithm. Additionally, this policy is consistent with the overall left-to-right reading of a domain name.

The next example examines the algorithm’s treatment of digits bordering conflicting directional boundaries. For example, when the Unicode Bidirectional Algorithm with the embedding level fixed to left-to-right is applied to test cases 5 and 6 in Table4 the output (12BA) is the same. This occurs, because digits bind with the last logical strong directional run. See rules W2 and W7 in Unicode Standard Annex #9 [5]. In test case 5 in Table4 the “12” binds with the “AB”, while in test case 6 in Table4 the “12” binds with the left-to-right embedding. Our algorithm, however always breaks directional ties by examining the types of the digits. In other words, European numerals always bind with a left-to-right run, while Arabic numerals always bind with a right-to-left run. The authors argue that this trade-off is acceptable as this rule is easier to comprehend and implement despite the discrepancy with Uni-

Table 3: Labels with letters and hyphens

Test Case #	Logical	Display	Comment
1	abc	abc	L,L,L
2	ABC	CBA	AL,AL,AL
3	NOP	PON	R,R,R
4	abDE	abED	L,L,AL,AL
5	DEab	EDab	AL,AL,L,L
6	abNO	abON	L,L,R,R
7	NOab	ONab	R,R,L,L
8	abDEgh	abEDgh	L,L,AL,AL,L,L
9	ABdeGH	BAdeHG	AL,AL,L,L,AL,AL
10	ABNOde	ONBAde	AL,AL,R,R,L,L
11	ab-de	ab-de	L,L,ET,L,L
12	AB-DE	ED-BA	AL,AL,ET,AL,AL
13	NO--QR	RQ--ON	R,R,ET,ET,R,R
14	ab-DE--NO	ab-ON--ED	L,L,ET,AL,AL,ET,ET,R,R
15	AB--de-NO	BA--de-ON	AL,AL,ET,ET,L,L,ET,R,R

code's output. Most significantly, it enables the algorithm to remain one-to-one.

Table 4: Labels with letters, digits, and hyphens

Test Case #	Logical	Display	Comment
1	ab12	ab12	L,L,EN,EN
2	56-ab	56-ab	AN,AN,ET,L,L
3	56-AB	BA-56	AN,AN,ET,AL,AL
4	56--NO	ON--56	AN,AN,ET,ET,R,R
5	AB12	BA12	AL,AL,EN,EN
6	12AB	12BA	EN,EN,AL,AL
7	12-34-AB	12-34-BA	EN,EN,ET,EN,EN,ET, AL,AL
8	12NO	12ON	EN,EN,R,R
9	1256AB	12BA56	EN,EN,AN,AN,AL,AL
10	5612AB	5612BA	AN,AN,EN,EN,AL,AL
11	AB-56-78	78-56-BA	AL,AL,ET,AN,AN,ET,AN ,AN
12	AB-12-34	BA-12-34	AL,AL,ET,EN,EN,ET, EN,EN
13	AB-12-34-CD	DC-34-12-BA	AL,AL,ET,EN,EN,ET, EN,EN,ET,AL,AL
14	AB-56-78-CD	DC-78-56-BA	AL,AL,ET,AN,AN,ET, AN,AN,ET,AL,AL
15	NO-12-34-AB	BA-34-12-ON	R,R,ET,EN,EN,ET,EN, EN,ET,AL,AL
16	ab-56-78-cd	ab-78-56-cd	L,L,ET,AN,AN,ET,AN, AN,ET,L,L
17	ab-12-56-CD	ab-12-DC-56	L,L,ET,EN,EN,ET,AN,AN , ET,AL,AL
18	ab-56-12-CD	ab-56-12-DC	L,L,ET,AN,AN,ET,EN,EN , ET,AL,AL
19	NO1256PQ	QP1256ON	R,R,EN,EN,AN,AN,R,R
20	NO5612ab	56ON12ab	R,R,AN,AN,EN,EN,L,L
21	NO1256ab	ON1256ab	R,R,EN,EN,AN,AN,L,L
22	12-34	12-34	EN,EN,ET,EN,EN
23	56-78	78-56	AN,AN,ET,AN,AN

VI. Conclusion

The contributions of this paper are:

- Exposes the essence of bidirectional reordering.
- An illustration of separating inferencing from reordering.
- An argument for the importance of a one-to-one algorithm for domain names.
- A Proposal for multilingual domain names and their display: honors legacy, is one-to-one, and is simple.

When domain names are interspersed within natural language text the problem of displaying the text and domain names becomes rather complex. This complexity, however can be managed if the problem is broken into separate and distinct phases. The problem with simply modifying the Unicode Bidirectional Algorithm to accommodate domain names is it makes an already complex algorithm even more difficult to manage.

The essence of the Unicode Bidirectional Algorithm is first to perform contextual analysis on the text and then determine where the boundaries of the directional runs are. The general problem with this strategy is that as technology continues to expand greater demands will be placed upon the bidirectional algorithm to always correctly render any and all textual data causing the algorithm to be in a constant state of flux.

When the Unicode Bidirectional Algorithm performs contextual analysis on text it overrides the static proprieties assigned to some of the characters. Specifically, this occurs during the processing of weak and neutral types. Separating this portion of the algorithm from resolving implicit levels and reordering levels greatly extends the applicability of the algorithm. Ideally the analysis of the text should be distinct from the actual determination of directional boundaries.

During the analysis phase domain names, mathematical expressions, phone numbers, and other higher order data elements are detected. Nevertheless, it is impossible to create an algorithm that can always correctly identify such elements. The real issue is whether or not it is possible to create an algorithm that identifies such elements within some reasonable range of error and under a set of acceptable constraints for the elements themselves.

The determination as to whether a stream contains a domain name is rather straightforward if the domain name is preceded by some special identifier. Specifically, "http://", "ftp://", or "telnet://". When these identifiers are not present, however the ability to recognize a domain name becomes more challenging. The authors believe it is unreasonable to force every domain name to be preceded by some special signal. There are many cases where it is inappropriate to specify the protocol. For example, consider the case where a domain name appears in a printed advertisement on a bus. The authors therefore recommend that there be a clear separation between natural language element detection and the rendering of those elements. In the future we plan to examine such issues.

VII. References

[1] Atkin, Steven and Stansifer, Ryan "Implementations of Bidirectional Reordering Algorithms." 18th International Unicode Conference, April 2001.

[2] Ayna. Multilingual Domain Names.

[3] Mockapetris, P. "RFC 1034 — Domain Names Concepts and Facilities."

[4] Unicode Consortium, The. *The Unicode Standard, Version 3.0*. Addison-Wesley. 2000.

[5] Unicode Consortium, The. "Unicode Standard Annex #9 - The Bidirectional Algorithm." Available: *http://www.unicode.org/unicode/reports/tr9*. Retrieved: June 15, 2001.

[6] Unicode Consortium, The. "Unicode Standard Annex #15 - Unicode Normalization Forms." Available: *http://www.unicode.org/unicode.reports/tr15*. Retrieved: June 15, 2001.

VIII. Appendix

```
1.  // DomainName.java version 1.0
2.  // Converts domain names in logical and display order.
3.  // Steven Atkin
4.  // 6/15/01
5.
6.  import java.io.BufferedReader;
7.  import java.io.InputStreamReader;
8.  import java.io.IOException;
9.  import java.util.LinkedList;
10. import java.util.Stack;
11.
12. public class DomainName {
13.
14.     private class AttributedCharacter {
15.         private char character;
16.         private byte direction;
17.         private boolean digit;
18.
19.         public AttributedCharacter (char ch, byte type) {
20.             character = ch;
21.             digit = false;
22.             direction = type;
23.             // set all full stop characters to left
24.             if (type == CS)
25.                 direction = L;
26.             else if (type == EN || type == AN)
27.                 digit = true;
28.         }
29.         public byte getDir () { return direction; }
30.         public void setDir (byte dir) { direction = dir; }
31.         public boolean isDigit() { return digit; }
32.         public char getCharacter() { return character; }
33.     }
34.
35.     private static final byte L = 0;
36.     private static final byte R = 1;
37.     private static final byte AL = 2;
38.     private static final byte EN = 3;
39.     private static final byte ES = 4;
40.     private static final byte ET = 5;
41.     private static final byte AN = 6;
42.     private static final byte CS = 7;
43.     private static final byte BN = 8;
44.     private static final byte B = 9;
45.     private static final byte S = 10;
46.     private static final byte WS = 11;
47.     private static final byte ON = 12;
48.
49.
50.
51.
52.
53.
54.
55.
```

```
56.     // character mappings for 0-127
57.     private static final byte[] mixedMap = {
58.             BN,  BN,  BN,  BN,  BN,  BN,  BN,  BN,
59.             BN,   S,   B,   S,   WS,   B,  BN,  BN,
60.             BN,  BN,  BN,  BN,  BN,  BN,  BN,  BN,
61.             BN,  BN,  BN,  BN,   B,   B,   B,   S,
62.             WS,  ON,  ON,  ET,  ET,  ET,  ON,  ON,
63.             ON,  ON,  ON,  ET,  CS,  ET,  CS,  ES,
64.             EN,  EN,  EN,  EN,  EN,  AN,  AN,  AN,
65.             AN,  AN,  CS,  ON,  ON,  ON,  ON,  ON,
66.             ON,  AL,  AL,  AL,  AL,  AL,  AL,  AL,
67.             AL,   AL,  AL,  AL,  AL,  AL,  R,  R,
68.             R,   R,  R,  R,  R,  R,  R,  R,
69.             R,   R,  R,  R,  R,  R,  R,   S,
70.             ON,   L,   L,   L,   L,   L,   L,   L,
71.             L,   L,   L,   L,   L,   L,   L,   L,
72.             L,   L,   L,   L,   L,   L,   L,   L,
73.             L,   L,   L,  ON,  ON,  ON,  ON,  BN
74.         };
75.
76.     private byte[] activeMap = mixedMap;
77.
78.     public DomainName () {
79.         activeMap = mixedMap;
80.     }
81.
82.     // Convert a logical or display domain name
83.     public String convert (String domainName) {
84.         LinkedList attribs = assignAttributes(domainName);
85.
86.         resolveDigits(attribs);
87.         resolveHyphenMinus(attribs);
88.         return reorderStrong(attribs);
89.     }
90.
91.     // Use the character map to get the character attrributes
92.     private LinkedList assignAttributes (String label) {
93.         LinkedList list = new LinkedList();
94.
95.         for (int i = 0; i < label.length(); ++i) {
96.             final char character = label.charAt(i);
97.             final byte type = activeMap[character];
98.             list.add(new AttributedCharacter(character, type));
99.         }
100.        return list;
101.    }
102.
103.
104.
105.
106.
107.
108.
109.
110.
111.
112.
```

```
113. private String emptyStack(Stack stack) {
114.         StringBuffer result = new StringBuffer();
115.         while(!stack.empty())
116.             result.append(stack.pop());
117.         return result.toString();
118.     }
119.
120.
121. // Resolve numerals
122.     private void resolveDigits (LinkedList label) {
123.         byte lastStrong = L;
124.         boolean remaining = false;
125.         int len = label.size();
126.
127.         for (int i = 0; i < len; ++i ) {
128.             final byte type = ((AttributedCharacter) label.get(i)).getDir();
129.             if (type == L || type == AL || type == R)
130.                 lastStrong = type;
131.             else if (type == EN && lastStrong == L)
132.                 ((AttributedCharacter) label.get(i)).setDir(L);
133.             else if (type == EN)
134.                 remaining = true;
135.             else if (type == AN)
136.                 ((AttributedCharacter) label.get(i)).setDir(AL);
137.         }
138.         // If there are any unresolved European numerals, make the second pass.
139.         if (remaining) {
140.             lastStrong = L;
141.             for (int i = len-1; i >= 0; --i) {
142.                 final byte type = ((AttributedCharacter) label.get(i)).getDir();
143.                 final boolean isdigit = ((AttributedCharacter) label.get(i)).isDigit();
144.                 if ((type == L || type == AL || type == R) && !isdigit)
145.                     lastStrong = type;
146.                 else if (type == EN && (lastStrong == R || lastStrong == AL))
147.                     ((AttributedCharacter) label.get(i)).setDir(R);
148.                 else if (type == EN)
149.                     ((AttributedCharacter) label.get(i)).setDir(L);
150.             }
151.         }
152.     }
153.
154.
155. // Resolve hyphen-minus characters
156.     private void resolveHyphenMinus (LinkedList label) {
157.         byte lastStrong = L;
158.         boolean remaining = false;
159.         int len = label.size();
160.
161.         for (int i = 0; i < len; ++i ) {
162.             final byte type = ((AttributedCharacter) label.get(i)).getDir();
163.             if (type == L || type == AL || type == R)
164.                 lastStrong = type;
165.             else if (type == ET && lastStrong == L)
166.                 ((AttributedCharacter) label.get(i)).setDir(L);
167.             else if (type == ET)
168.                 remaining = true;
169.         }
```

```
170.        // If there are any hyphen-minus characters left, make the second pass.
171.        if (remaining) {
172.            lastStrong = L;
173.            for (int i = len-1; i >= 0; --i) {
174.                final byte type = ((AttributedCharacter) label.get(i)).getDir();
175.                if (type == L || type == AL || type == R)
176.                    lastStrong = type;
177.                else if (type == ET && (lastStrong == R || lastStrong == AL))
178.                    ((AttributedCharacter) label.get(i)).setDir(R);
179.                else if (type == ET)
180.                    ((AttributedCharacter) label.get(i)).setDir(L);
181.            }
182.        }
183.    }
184.
185.    // Reorder the characters once their directions have been resolved
186.    private String reorderStrong (LinkedList attribs) {
187.        byte mode = L;
188.        StringBuffer result = new StringBuffer(attribs.size());
189.        StringBuffer digits = new StringBuffer();
190.        Stack rightStack = new Stack();
191.
192.        for (int i = 0; i < attribs.size(); ++i) {
193.            final char character = ((AttributedCharacter) attribs.get(i)).getCharacter();
194.            final byte dir = ((AttributedCharacter) attribs.get(i)).getDir();
195.            final boolean isdigit = ((AttributedCharacter) attribs.get(i)).isDigit();
196.
197.            // left-to-right characters
198.            if (dir == L) {
199.                if (mode == AL || mode == R) {
200.                    result.append(digits);
201.                    result.append(emptyStack(rightStack));
202.                }
203.                else {
204.                    result.append(emptyStack(rightStack));
205.                    result.append(digits);
206.                }
207.                result.append(character);
208.                mode = L;
209.                digits = new StringBuffer();
210.            } // end if left
211.
212.            // right-to-left characters
213.            else if ((dir == AL || dir == R) && !isdigit) {
214.                rightStack.push(digits);
215.                rightStack.push(new StringBuffer().append(character));
216.                mode = AL;
217.                digits = new StringBuffer();
218.            } // end if Arabic or Hebrew
219.
220.            // Numerals
221.            else if (isdigit && (dir == AL || dir == R)) {
222.                    digits.append(character);
223.                    mode = dir;
224.            } // end if Arabic or European numeral
225.        } // end for loop
226.
```

```
227.        // cleanup
228.        if (mode == R || mode == AL) {
229.            result.append(digits);
230.            result.append(emptyStack(rightStack));
231.        }
232.        else {
233.            result.append(emptyStack(rightStack));
234.            result.append(digits);
235.        }
236.        return result.toString();
237.    }
238.
239.    public static void main (String args[]) {
240.        DomainName domain = new DomainName();
241.        String line = new String();
242.        BufferedReader in  = new BufferedReader(new InputStreamReader(System.in));
243.
244.        do {
245.            try {
246.                line = in.readLine();
247.            }
248.            catch (IOException e) {
249.                System.out.println("Error on input line");
250.            }
251.            if (line != null && !line.equals(""))
252.                System.out.println(domain.convert(line));
253.        }while (line != null && !line.equals(""));
254.    }
255. }
```