Natural Language Engineering **9** (1): 87–99. © 2003 Cambridge University Press
DOI: 10.1017/S1351324903003085 Printed in the United Kingdom

Lenient morphological analysis

KEMAL OFLAZER

Faculty of Engineering and Natural Sciences, Sabancı University, Orhanlı 81474, Tuzla, İstanbul, Turkey
e-mail: oflazer@sabanciuniv.edu

(*Received 6 April 2002; revised 30 October 2002*)

Abstract

This paper presents a scheme that allows one to relax the all-or-none nature of two-level constraints in two-level morphology in a controlled manner, so that word forms with violations of some of the two-level constraints can be analyzed and ranked. The problem has been motivated by a recent phenomenon in Turkish with imported words that violate a fundamental assumption of Turkish that pronunciation and orthography have almost a one-to-one correspondence, and by a problem in Basque words with differing amounts of competence errors. We present the formulation of our proposal, and provide details of implementations for both problems using the XRCE Finite State Toolkit.

1 Introduction

Two-level morphological analyzers (Koskenniemi 1983) have two components: the rule component handles morphographemic changes between the surface and lexical representations a word, while the lexicon component is populated with the lexical items comprising free and bound morphemes, and captures the morphotactics. Current implementations of two-level analyzers use the now well-established and well-understood finite state technology (Karttunen and Beesley 1992; Karttunen 1993; Kaplan and Kay 1994). The morphographemic rule component consists of two-levels rules that describe parallel constraints between lexical and surface representations. For a (*lexical-string, surface-string*) pair to be accepted (and hence for the surface string to be analyzed into the lexical string), *none* of the constraints imposed by the rules should be violated as there is no notion of a partial consensus on or acceptance of string pairings. There are, however, some interesting phenomena that could benefit from a notion of partial acceptance.

Consider the following: The orthography of Turkish follows pronunciation in an almost one-to-one way. Thus, morphophonological phenomena such as vowel harmony, can be handled by constraints over graphemic representations. However, recently, with rapid adoption of technical terms from English mainly in information technology, the lexical landscape has started to change, and the following phenomenon that we will exemplify with the word *serverlar* (servers) has started popping up. This word consists of the English root word *server* followed by the

Turkish plural suffix *+lar*. The interesting point here is that the vowel *a* (/A/) in the suffix harmonizes with the last vowel (/1/) in the *English* pronunciation of *server*.[1,2] Thus, the choice of the suffix vowel is felicitous as the morphophonological (but not the morphographemic) process is properly handled. Unless one builds a preliminary stage of an English text-to-speech system, the English pronunciation of *server* is not available to the morphological analyzer! So, the morphological analyzer rejects this form, since as written, vowel harmony is violated.[3] We could however have analyzed this form if the morphological analyzer was a bit forgiving about vowel harmony violations across the boundary and the first suffix.

Consider also this example from Basque: Basque speakers occasionally make competence errors in orthography due to certain effects of Spanish. One such error is that occasionally a lexical *k* appears as a surface *c*. A second competence error causes the deletion of a root-final *a* on the surface.[4]

Then, assuming there are two lemmas in the lexicon, *kale* and *kala*, the Basque form *caletik* would get analysed into two possible lexical strings:

Lexical:	k a l e + E t i k	**Lexical:**	k a l a + E t i k
Surface:	c a l e 0 0 t i k	**Surface:**	c a l 0 0 e t i k

where competence errors are underlined.[5] The first analysis has one competence error while the second one has two. It would be desirable to rank the analyses based on the number of competence errors, if any, and eliminate analyses with higher number of competence errors.

The reason why these cannot be handled in two-level systems is that the rules for mediating between the surface and lexical forms do not allow (controlled) violation of one or more rules, and there are no mechanisms for ranking and accepting based on the count and nature of these violations. Note that, we are not attempting spelling correction. In fact, the orthographically correct (but not necessarily phonologically palatable) forms for some of the Turkish cases can be quite far away from the input word in terms of the edit distance measure used in spelling correction (Oflazer 1996), due to chain effect of vowel harmony constraints, for instance.

[1] We use the SAMPA phonetic encoding. See `http://www.phon.ucl.ac.uk/home/sampa/home.htm`

[2] There is an implicit assumption here in the mind of the (Turkish) writer of such words that the (Turkish) reader of such words knows their English pronunciations!

[3] This has become a rather common phenomenon in magazines and papers covering IT and internet topics, accounting for 3 to 5% of the word forms in such text. The main reason is that the number of new concepts being introduced vastly exceeds anybody's capability to propose native words for them, and that most writers of such text know English and so import these words in a wholesale manner!

[4] Iñaki Alegria (personal communication) indicates that in "standard texts" (books, newpapers, magazines) the competence errors range from 0.7% to 5% but this number is much higher in casual texts such as e-mails. The deletion of the root final *a*, can occur in as much as 10% of the word forms with dialectal or competence error variations.

[5] Other morphographemic phenomena in these examples are mediated by other two-level constraints.

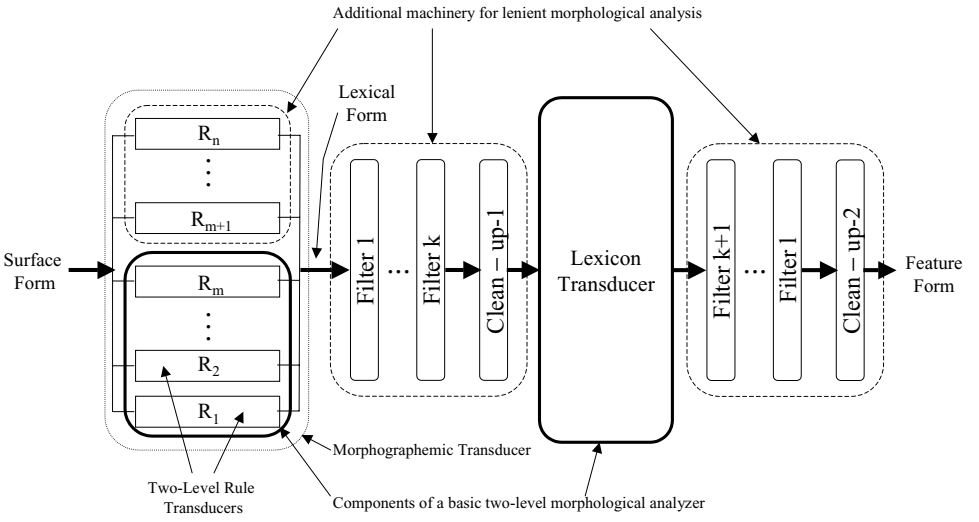

Fig. 1. The general architecture of a lenient morphological analyser.

2 Lenient morphology

Our proposal for handling these problems brings together two-level morphology and finite state optimality-theoretic lenient filters (Karttunen 1998). The approach does not require any additional machinery other than what the two-level formalism and finite state transducers provide. It basically relies on (i) augmenting two-level rules that can be violated, by additional rules and feasible pairs of symbols to recognize and mark any violations, and (ii) filtering lexical forms with any such violation markers using a sequence of finite state (lenient) filters that select among possible lexical forms, based on the number and the nature of violations.

The architecture of an augmented lenient two-level morphological analyser is shown in Figure 1.[6] The components of the basic two-level morphological analyzer are depicted with bold lines. The morphographemic transducer is the intersection of the finite state transducers for each of the two-level rules. The lexicon transducer (which is not of much interest for the purpose of this work) maps between lexical forms and feature forms which are the ultimate outputs of the morphological analyser.

The augmented architecture has additional rules (and hence their transducers) and a series of finite state transducers that manipulate lexical representations of analysed word forms with any violation markers. The additional machinery used also makes sure that any changes in representation are minimal so that the interface to the lexicon transducer (which is where bulk of the lexical content resides) is essentially the same.

[6] Note that this picture depicts a two-level morphological analyzer in a different layout compared to what is typically used in the literature: the morphographemics transducer is on the left side while the lexicon transducer is on the right side.

2.1 Augmenting two-level rules

Two-level morphology provides three distinct types of rules and another one which is the combination of two of the rule types. We now describe these and how they are augmented for lenient analysis. In these rules, a:b denotes a lexical-surface feasible pair,[7] and LC_i and RC_i denote arbitrary regular expressions over the set of feasible pairs, constraining the left and right contexts of the feasible pair on the left hand side of the rule, respectively.

1. The *Context Restriction* rule a:b => LC_1 _ RC_1 ; . . . ;LC_k _ RC_k ; states that a lexical a can paired with a surface b only in one of contexts specified on the right side. A violation of this rule is an occurrence of the feasible pair a:b in a context other than those specified. To mark such violations
   (a) We add the feasible pair X:b to the set of feasible pairs.[8]
   (b) We add the context prohibition rule X:b /<= LC_1 _ RC_1 ; . . . ;LC_k _ RC_k ; (see below), to the set of rules so that the X:b pair never appears in a legitimate context. Any overgeneration of the X:b pair will be filtered by filters and the lexicon transducer later.
2. The *Surface Coercion* rule a:b <= LC_1 _ RC_1 ; . . . ;LC_k _ RC_k ; states that a lexical a *must* be paired with a surface b in all the contexts specified. A violation of this rule is an occurrence of a feasible pair a:w (w≠b) in one of these contexts. To mark such violations
   (a) We introduce feasible pairs X:w to the set of feasible pairs, for each w≠b such that a:w is a feasible pair.
   (b) We add a rule X:w => LC_1 _ RC_1 ; . . . ;LC_k _ RC_k ; to the set of rules for each such w≠b. These rules limit the occurrence of the violation markers only to the relevant contexts.
3. The *Context Prohibition* rule a:b /<= LC_1 _ RC_1 ; . . . ;LC_k _ RC_k ; states that the pair a:b can *not* occur in any of the contexts specified. A violation of this rule is an occurrence of a:b in one of the contexts specified. To mark such violations
   (a) We add the feasible pair X:b to the set of feasible pairs.
   (b) We add the rule X:b => LC_1 _ RC_1 ; . . . ;LC_k _ RC_k ; to the set of rules so that X:b is allowed to appear in contexts where a:b is prohibited.
4. The *Combination* rule a:b <=> LC_1 _ RC_1 ; . . . ;LC_k _ RC_k ; is a combination the first two rules. It can be augmented in two steps by handling the => and <= components separately.

We also make the following changes in the contexts of all rules due to the new feasible pairs introduced. For context restriction and prohibition rules, if feasible-pair symbols a:b or a: (denoting all feasible pairs with the lexical symbol a) occur

[7] We assume single symbols appear on the left-hand side of the rules, not the more general arbitrary regular relations.

[8] We use X as a generic placeholder for violation markers on the lexical string side. Each rule will have different violation marker symbol.

in the regular expressions describing the contexts in some rules, then we replace those occurrences with the union of these symbols with all new feasible pairs `X:b` generated from `a:b`, so that `X`'s behaves just like `a` as far as contexts are considered. For surface coercion rules, this has to be done for any feasible pairs `a:w`, `w`$\neq$`b`, unioning its occurrence with `X:w`.

The following example shows how two simplified rules for a certain vowel harmony process in Turkish will need to be augmented. The set of feasible pairs, Σ, consists of `A:a` and `A:e` (plus possibly others) where `A` is a lexical meta-symbol that represents a unrounded back vowel with unspecified height feature.

Original set of rules			$\Longrightarrow$	**Augmented set of rules**						
Σ	=	{A:a,A:e,...}		Σ	=	{A:a,A:e,Xa:e,Xe:a,...}				
A:a	<=	LCa _ ;		A:a	<=	LCa _ ;	A:e	<=	LCe _ ;	
A:e	<=	LCe _ ;		Xa:e	=>	LCa _ ;	Xe:a	=>	LCe _ ;	

The augmented set of rules has two additional feasible pairs `Xa:e` and `Xe:a`. The first one, if it appears in a lexical form, indicates that even though the context required an `A:a` pairing, an `A:e` pairing was found. The original set of rules would have rejected an `A:e` pairing in this context, but the augmented set of rules would pair `Xa` with the surface `e`. The second is for the symmetric `Xe:a` pair.

2.2 Filtering forms with violations

As such, the lexical side of the augmented set of rules would produce lexical forms with 0 or more violations. Naturally, we would like to see if there are any lexical forms that do not involve any violations. If so, then they may be what we are after.[9] If however, there are no such forms, that is, all lexical forms have at least one violation, then we would like to implement a policy of choosing among these lexical forms with violations. The following constraints or their combinations are some interesting options:

- One can choose lexical forms with the smallest number of violations or with up to some small number of violations.
- One can choose lexical forms with specific kinds of violations. For instance, for Turkish, one can allow at most one vowel harmony violation or at most one consonant deletion violation, or perhaps no more than two violations total.
- One can choose lexical forms where violations are only across the first suffix boundary or in certain suffixes. This would again be useful for the Turkish example discussed earlier, since, once the first suffix is properly resolved no violations would be allowed in subsequent morphemes or later within the same morpheme. For instance, both of these words would be marked as illegal:

[9] Though, as we will see in the Basque case, these actually may not be the ones we are interested in.

— *serverlarde*, because the last vowel *e* (/e/) is not correct; it should have been an *a* (/A/) to harmonize with the previous surface vowel *a*. The last vowel to harmonize to is in the first suffix (the (intended) lexical form is *server+lAr+DA*.[10]) Thus even though there is a single violation, it is not a violation we would want to tolerate.
— *serverımuz*, because the last vowel *u* (/u/) which happens to be the second vowel in the suffix (the (intended) lexical form is *server+HmHz*[11]) should have been an *ı* (/1/) as it is conditioned by the first vowel in the suffix. Again, this is not a violation we would want to tolerate.

Some of the constraints that we would like to filter with can be hard (or merciless) constraints. For instance for the last case above, one may use a hard constraint: Any lexical form that violates this can be considered to be a wildly off case and eliminated. Some other constraints (e.g., the first two) may need to be implemented as (a cascade of) lenient filters. We can use the filters on both sides of the lexicon transducer in Figure 1 to implement the required functionality.

2.3 Lenient filters

Lenient filters are implemented with Karttunen's lenient composition operator (Karttunen 1998). It is best to think of lenient composition in the context of a *generator-filter* combination. A generator *G*, which in our case corresponds to the two-level rule transducer, produces (a set of) outputs – the lexical forms – in response to an input surface string. When such a generator *G* is composed with a filter *F*, using lenient composition, the resulting finite state transducer will pass through only those outputs of *G* that satisfy the filter constraint. If however, no output satisfies the filter constraint, then all outputs of *G* are let through, that is, the filter behaves as if it is off.

Figure 2 shows a simple use of these filters where the lexical side of two-level rules transducer (the generator) is feeding into a series of two finite state filters, along with the intermediate outputs of these filters in response to two inputs.[12] The first filter labeled *Allow 0 violations*, which is a lenient filter, will pass through only those outputs of the generator that have no violation markers, while the second filter labeled *Allow* $\leq$ *1 violations*, which is a merciless filter will pass through those outputs of the filter above that has 0 or 1 violation markers. This figure also shows a clean-up transducer which removes any violation markers from the eventual outputs.

On the left side, lexical form for the input surface form *serverlarda* (in the servers) has a single violation coming out of the rule transducer. The lenient filter finds none

[10] D is a lexical meta-symbol denoting dental consonant, one of *d* (/d/) or *t* (/t/).
[11] H is a lexical meta-symbol denoting a high vowel, one of *ı* (/1/), *i* (/i/), *u* (/u/) or *ü* (/Y/).
[12] Note that we have just shown only one of the possible lexical forms as without the lexicon, the two-level rule component is wildly overgenerating.

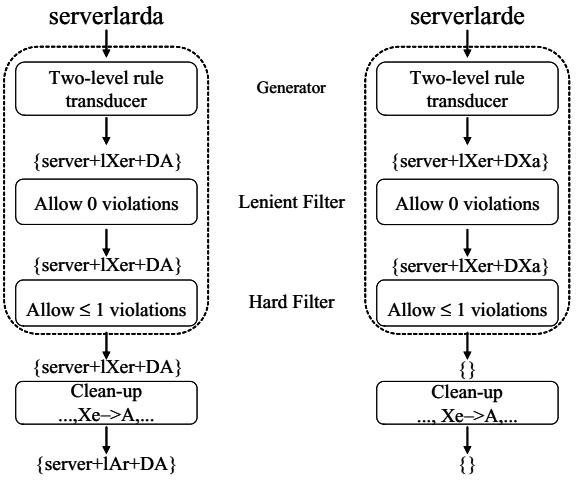

Fig. 2. Tracing the outputs of the morphographemic transducer through a series of filters.

of the inputs has 0 violations, and so passes all down to the second filter. The second filter finds a string with only 1 violation at its input and passes it through.

On the other hand, the input surface form on the right side has a lexical form with two violations. The first filter finds no inputs with 0 violations and passes all to the next stage. The second filter finds no input with 1 or less violations, and being a hard constraint, blocks the inputs producing no outputs.

3 Implementation examples

In this section we provide details of implementations to deal with the problems in Turkish and Basque discussed earlier.

3.1 Turkish

For Turkish we will only implement the vowel harmony and consonant deletion examples. First we provide the three two-level rules that deal with these cases: two rules for resolving the `A:a` vs. `A:e` vowel harmony, and one rule for deleting a morpheme initial `y` after a consonant-final stem. First we give the three (rather simplified) two-level rules that handle these phenomena:[13]

```
"A is realized as a"
     A:a <= [:BACKV] [CONS]* %+:0 [CONS: | :CONS | :0]* _ ;
"A is realized as e"
     A:e <= [:FRONTV] [CONS]* %+:0 [CONS: | :CONS | :0]* _ ;
"Morpheme initial y is deleted after a stem-final consonant"
     y:0 <= [:CONS] %+:0 _ ;
```

[13] We use the syntax of the XRCE Finite State Tools. `%` and "..." escape characters that otherwise have special uses in regular expression syntax.

The first two rules have the same left context except for the conditioning vowel: `:BACKV` in the first rule denotes all feasible pairs whose surface symbol is a back vowel (one of *a* /A/, *ı* /1/, *o* /o/, *u* /u/), `:FRONTV` in the second rule denotes all feasible pairs whose surface symbol is a front vowel (one of *e* /e/, *i* /i/, *ö* /2/, *ü* /Y/). The other parts of the left contexts are pretty much obvious and not really relevant. The relevant feasible pairs are: `A:a, A:e, A:0, y:0, y:y`. After we augment these rules we get the following additional rules:

```
"A is realized as e in a wrong context, Xa marks the violation"
    Xa:e => [:BACKV] [CONS]* %+:0 [CONS: | :CONS | :0]* _ ;
"A is realized as 0 in a wrong context, Xa marks the violation"
    Xa:0 => [:BACKV] [CONS]* %+:0 [CONS: | :CONS | :0]* _ ;
"A is realized as a in a wrong context, Xe marks the violation"
    Xe:a => [:FRONTV] [CONS]* %+:0 [CONS: | :CONS | :0]* _ ;
"A is realized as 0 in a wrong context, Xe marks the violation"
    Xe:0 => [:FRONTV] [CONS]* %+:0 [CONS: |:CONS | :0]* _ ;
"y not deleted  where it should be , Xy marks the violation"
    Xy:y => [:CONS] %+:0 _ ;
```

These rules are compiled into finite state transducers and intersected by the XRCE *twolc* compiler (Karttunen and Beesley 1992) and saved as a transducer to the file `turkish.fst`. The rest of the implementation consist of a script for the XRCE finite state tool *xfst*, employing a toy lexicon to show the main points. One thing we should point out is that relaxing the rules for native Turkish root words may not be desirable. That is, we would like to localize violations to the foreign word lexicon. We would however like to do this with a very minimal change to the lexicon. First, we define the rule transducer but in the downward direction:

```
define TwoLevelRules [@"turkish.fst" ].i;
```

The root word lexicon is split into two, with the foreign root words being prefixed with the `^` symbol, and all lexical word forms being optionally prefixed with a marker `!`.

```
define ForeignRootLexicon [ {^server} | {^girl} |{^cat}|{^bordeaux}];
define TurkishRootLexicon [ {ev} | {masa} ];
define RootLexicon [ForeignRootLexicon | TurkishRootLexicon];
define MorphemeLexicon [[{+lAr} | {+DA} | {+Hm} | {+HmHz} | {+yH} ]];
define Lexicon [ ("!") RootLexicon MorphemeLexicon* ];
```

The following define symbols that mark violations and symbols which can be "violated":

```
define HarmonyVMarkers [ "Xa" | "Xe" ];
define ViolationMarkers [HarmonyVMarkers | "Xy" ];
define VSymbol [ A ];
```

The following define filters that allow inputs with at most 0, 1 and 2 violating symbols, respectively:[14]

```
define Violation0  ~  $[ ViolatingSymbols ];
define Violation1  ~[[$[ ViolatingSymbols ]]^ > 1];
define Violation2  ~[[$[ ViolatingSymbols ]]^ > 2];
```

The following filters localize violations to specific locations: The form either has no violations, or any violations appear in the first morpheme, that is, there is one morpheme boundary to the left. The next two regular expressions below force the vowel harmony violation to the first violatable symbol in the first morpheme. Finally, the last intersects the previous three into one filter which enforces all three filters.

```
define Localize1 [ ViolatingSymbols => .#. ~$["+"]  "+" ~$["+"] _ ];
define Localize2 ~[$[HarmonyVMarkers ~[$"+"] HarmonyVMarkers]];
define Localize3 ~[$[VSymbol ~[$"+" ] HarmonyVMarkers]];
define VLocalization [Localize1 & Localize2 &  Localize3];
```

`RememberViolations` optionally inserts the `"^"` marker to the left side of the form and then inserts the `"!"` marker to the beginning if there is any violation marker in the intermediate lexical form.

```
define RememberViolations [ [. .] (->)  "^" || .#. _ ] .o.
                          [ [. .]  ->   "!" || .#. _ $[VSymbol]];
```

This next filter limits the occurrence of the `"!"` marker to just before a `"^"` marker so that only words involving roots from the foreign root lexicon can involve violations.

```
define FilterTurkishRootsWithViolations ["!" => .#. _ "^" ];
```

The following are the two clean-up transducers to remove the violation markers and other temporary symbols.

```
define CleanUp1 [ [ "Xa" | "Xe" ] -> A, "Xy" ->y ];
define CleanUp2 [["!" | "^"] -> 0];
```

The lenient analyser now looks like:[15]

```
define LenientAnalyzer  TwoLevelRules .o. ViolationLocalization .O.
                        Violation0 .O. Violation1 .o. Violation2 .o.
                        RememberViolations .o.  CleanUp1 .o.
                        Lexicon  .o.
                        FilterTurkishRootsWithViolations  .o.
                        CleanUp2;
```

[14] For instance the last definition here states "it is not the case that there are more than 2 segments of the input that include a violating symbol."

[15] `.O.` denotes the lenient composition operator while `.o.` denotes the standard composition operator.

This lenient analyser would allow up to two violations provided they do not violate the localization constraints. Any lexical form with more constraints would get killed by the merciless composition involving `Violation2`. After the `CleanUp1` transducer applies, the outputs are now composed with the lexicon. Since only the foreign root words are prefixed with `^` in the lexicon, any analysis involving `^` followed by a native Turkish root word would be eliminated. Any analysis not prefixed with `^` when composed with the lexicon must have a native Turkish root. The subsequent filter `FilterTurkishRootsWithViolations` allows the `!` marker only if it is followed by the `^` marker. Thus any forms with `!` markers followed by a native Turkish root are eliminated.

3.2 Basque

In this section we present an implementation for handling the competence-error based ranking in the morphological analysis of the Basque words. As described earlier, Basque speakers occasionally make competence errors in orthography due to certain effects of Spanish. These errors may lead to word forms which are homographs of legitimate forms and increase spurious morphological ambiguity. It would be desirable if such errors can be detected and filtered in the presence of forms with less or no errors.

The Basque problem is different from the Turkish one is a fundamental way: A Turkish lexical form with violation markers would not normally have been analyzed by the morphographemic component of the original Turkish analyzer, that is, those forms are technically misspelled.[16] So, when the generator (the morphographemic transducer) generates forms with violation markers, *it produces no forms without violation markers*. On the other hand, in the Basque case, a surface form may give rise to lexical forms with and without violation markers. For instance, if `c:c` is a valid feasible pair, then `cale+Etik` would also be a valid lexical form for the surface form `caletik`, and this form has no violations. Lenient filtering before the lexicon would let this form pass through removing lexical forms with violation markers, even though we may not have a root word `cale` in the lexicon. It seems that we need to filter with the lexicon first, and then eliminate parses with violation markers but this needs to be done without any major changes to the lexicon.

The first part of the solution involves changes to the rule component: we add new feasible pairs for the competence errors and let violation markers appear in the relevant lexical forms. For the sample problem at hand, as there are no specific rules, we just have to add rules to force competence errors inject violation markers into the lexical forms. The following rule (along with the feasible pair `Xa:0`) allows the deletion a root final `a`, but marking it with the violation marker `Xa:`

```
"root final a deletion"
Xa:0 => ~$[%+:0] _ %+: ;
```

[16] Unless a Turkish and a foreign root word are homographs.

The second marker occurs freely with no real contextual constraints. So introducing a feasible pair `Xk:c` allows for this competence error. The following script implements the rest of the solution. The most important change is that first we count the number of competence errors and stash this away at the beginning of the lexical form and then change the violation markers back to the lexical symbols they replace so that the forms can now be composed with the lexicon. First, we define the two-level rules and the lexicon:

```
define TwoLevelRules [@"basque.fst"].i;
define RootLexicon [ {kale} | {kala}];
```

and the violation markers and count markers:

```
define VM [ "Xk" | "Xa" ];
define VCM [ "0"  | 1 | 2];
```

The lexicon is preceded by a violation count marker:

```
define Lexicon [ VCM RootLexicon [{+Etik}]];
```

The next two filters pass through inputs with exactly one and two violations respectively, and the last transducer marks each form with the number of violations (up to 2) by inserting a count marker at the beginning of the string.

```
define OnlyOneViolation  ~[$VM] VM ~[$VM];
define OnlyTwoViolations ~[$VM] VM ~[$VM] VM ~[$VM];
define CountViolations [[. .]->"0" || .#. _ ~$[VM].#. ,,
                       [. .]->"1" || .#. _  OnlyOneViolation .#. ,,
                       [. .]->"2" || .#. _  OnlyTwoViolations .#.];
```

The following filters define violation count patterns for forms that result from composing lexical forms with violations, with the lexicon transducer:

```
define NoViolations         "0" ?*;
define AtMostOneViolation  ["0" | 1 ]?*;
define AtMostTwoViolations ["0" | 1 | 2 ]?*;
```

Finally, the following define transducers for cleaning up violation markers and temporary symbols.

```
define CleanUp1 [  "Xa"  -> a, "Xk" ->k ];
define CleanUp2 [   VCM -> 0 || .#. _ ];
```

The lenient analyser now is defined as follows:

```
define LenientAnalyzer TwoLevelRules .o.
       CountViolations .o. CleanUp1 .o.
       Lexicon .O.
       NoViolations .O. AtMostOneViolation .o.AtMostTwoViolations .o.
       CleanUp2;
```

Figure 3 traces the input `caletik` through all the components of the lenient analyzer transducer. The output, as expected, is `kale+Etik`, which has only one violation.

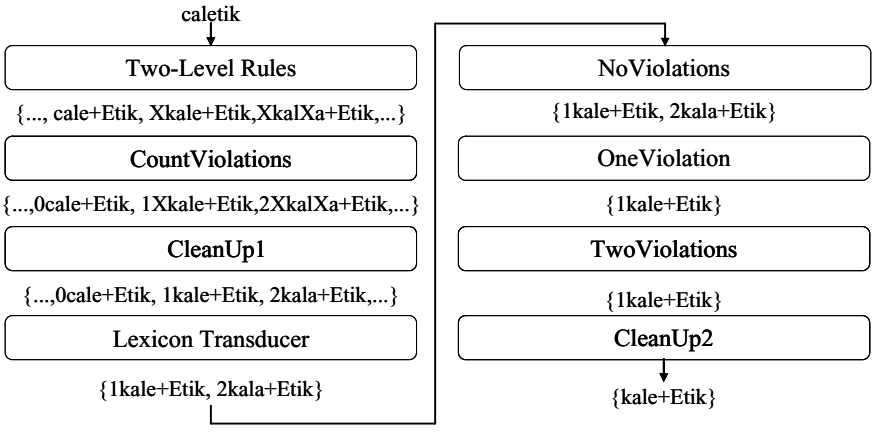

Fig. 3. Tracing `caletik` through the lenient Basque analyser.

4 Discussion and conclusions

We have presented a framework that combines two level morphological analysers with optimality-theoretic constraint filters to allow principled analyses of word forms with controlled violations of some of the two-level constraints, and demonstrated solutions to two problems from Turkish and Basque languages. The solutions require no changes to the two-level machinery and utilize combinations of Karttunen's lenient filters and representational augmentations with absolutely minimal changes to the lexicon.

Finite state morphological analysers are also expected to function as morphological generators that generate all surface forms given a lexical form. This becomes an interesting issue when we employ the approaches we have discussed above. In general, our analysis will produce a single lexical form but that lexical form may correspond to multiple surface forms with differing numbers of violations. For instance, it may be that *kalatik* also maps to *kala+Etik*, but this surface form has no competence violations. It would be quite useful if we could generate the surface form with the minimal number of violations. The recent progress in bidirectional finite state optimality theory reported by Jäger (2001) has most of the answers to this problem.

Acknowledgements

I thank Iñaki Alegria of the Department of Computer Science at the University of the Basque Country for providing me with the example problem from Basque and for numerous discussions on it. I also thank Kimmo Koskenniemi and Lauri Karttunen for inviting me to participate at the ESSLLI 2001 Special Event Panel on *20 Years of Two Level Morphology*, in Helsinki, where the main points of this paper were first presented.

References

Jäger, G. (2001) Gradient constraints in finite state OT: The unidirectional and the bidirectional case. Technical report, OTS, University of Utrecht, The Netherlands.

Kaplan, R. M. and Kay, M. (1994) Regular models of phonological rule systems. *Computational Linguistics* **20**(3): 331–378.

Karttunen, L. and Beesley, K. R. (1992) Two-level rule compiler. Technical Report, XEROX Palo Alto Research Center.

Karttunen, L. (1993) Finite-state lexicon compiler. Technical Report, XEROX Palo Alto Research Center.

Karttunen, L. (1998) The proper treatment of optimality theory in computational linguistics. In: Karttunen, L. and Oflazer, K., editors, *Proceedings International Workshop on Finite State Methods in Natural Language Processing (FSMNLP)*.

Koskenniemi, K. (1983) Two-level morphology: A general computational model for word form recognition and production. Publication No: 11, Department of General Linguistics, University of Helsinki.

Oflazer, K. (1996) Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics* **22**(1): 73–90.