Error Detection, Recovery, and Repair in the Translation of Inorganic Nomenclatures. 3. An Error Handler[§]

I. Luque Ruiz,*[,‡] J. L. Cruz Soto,[†] and M. A. Gómez-Nieto[‡]

E.U.P., Avda, Menéndez Pidal s/n, and Science Faculty, Avda, S. Alberto Magno s/n, Córdoba University, 14071-Córdoba, Spain

Received August 18, 1995[⊗]

The development of a system for the repair of lexical errors detected in the process of recognizing inorganic chemical names is described. Repairs are based on the calculation of similarity between strings, using the model and approximate matching algorithm detailed in the previous article of this series. A hierarchical data structure is proposed for the storage of information generated during the repair process. This structure comprises two hierarchically-related abstract data types, to permit the storage of partial and total repairs generated by the process. The great amount of information generated by the system might recommend its use in applications requiring explanatory subsystems to report on the success of recognition of user-input information. Problems in the repair process arising from inorganic grammar structures are also examined, and solutions are proposed for cases such as that of the delimiters between significant parts of the inorganic substance name.

1. INTRODUCTION

The repair of lexical errors appearing in inorganic chemical names poses a series of problems due to the syntax of the languages (the nomenclatures) used to write these names and also to the very nature of the problem of repair, which necessarily involves the quantitative determination of the "*similarity between* strings.[1−5]

The previous two articles in this series[4,5] examined these problems and proposed a model for lexical error repair; the present article describes its practical application to such problems.

Three main factors intervene in the repair process: *similarity threshold*, *length of the error string*, and *syntactic compatibility of the repair*. A general approach to deal with these factors has been proposed.[4,5] However, since syntactic compatibility of the repair depends on the grammer of the language involved, a different solution is required for each language. This article describes how the proposed model was applied to the repair of lexical errors detected during the translation of inorganic names to an abstract data structure by a translation system described in previous studies (see Figure 1).[6,7]

In line with the proposed model[5] and due to the syntax of inorganic names, the repair process has to be carried out in successive stages in which repairs are obtained for all or part of the error string. This process gives rise to a great quantity of information and will require dynamic data structures for storage. These data structures, in addition to the architecture of the repair system, are described herein.

2. THE ERROR HANDLER SUBSYSTEM

Figure 1 shows how the error handling system is called by the lexical analyzer once its analysis process has been concluded. The system is incorporated into the inorganic nomenclature translator system,[6] and operates as follows:

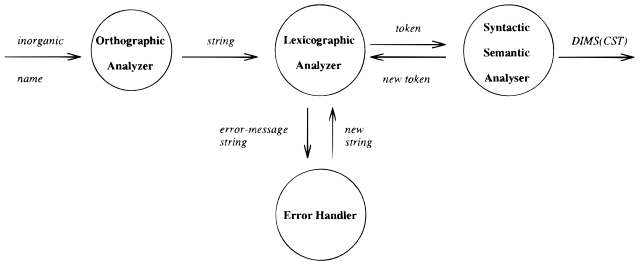

Figure 1. Error handler subsystem (DIMS (dynamic internal representation of molecular structures) and CST (compound structure table)).

- The error handler attempts to repair errors discovered by the lexical analyzer at each successive step, Scan-1, Scan-2, and Scan-3.[6]
- An input string is considered to contain incorrect substrings and is consequently sent to the error handler for repair if the errors detected are of Type II.1.[4] In other words, the error handler subsystem is called by the lexical analyzer once the input string has been completely analyzed but only on the detection of substrings which do not coincide with any of the terminal symbols of the grammar, and only if these error substrings might occasion a loss in the knowledge represented by the input string.
- The input to the error handler is composed of the following:

1. The output string built by the lexical analyzer. This string is made up of a collection of separator-delimited tokens and the error substring(s).[6]
2. The collection of error exceptions detected up until the syntactic analysis phase, that is, all exceptions or errors detected by the orthographic/lexicographic analyzer.

- The output of the error handler is a set of new strings composed of valid morphemes and delimited by separators, where

[†] Science Faculty, Avda. S. Alberto Magno s/n. e-mail: jlcruz@sun630.uco.es.

[‡] E.U.P., Avda. Menéndez Pidal s/n. e-mail: malgonim@lucano.uco.es :: mallurui@sun630.uco.es.

[§] Key words: error handlers, repair of lexical errors in inorganic names, approximate matching algorithm, abstract data structures.

[⊗] Abstract published in *Advance ACS Abstracts,* March 1, 1996.

1. Every error substring has been substituted by one or more morphemes.
2. The substitution or repair carried out on the error string must satisfy the following conditions:
   (a) The repair must be orthographically correct.
   (b) The repair must be lexicographically correct.
   (c) The repair must be syntactically correct.

However, the repair need to be semantically correct.

•By correct repair it is understood that the process of substitution of an error substring by a new substring must mean that no new errors are produced, even if the newly-generated string is not that which the user wished to input.

•One of the current limitations of the error handler is that the repair does not always produce a new string which is semantically correct, since the semantic aspect of nomenclatures is not dealt with at this particular juncture.

•More than one repair will be possible for an error string. Thus, in principle, given c error substrings in an input string, for each substring c_i there will be r_{i_j} possible repairs, so for each string input into the error handler there is a possible total of $R = \prod_{i=1}^{i=c} r_{i_i}$ repairs, not all of which will be correct.

•Of the total of R repairs, only the subset R^* of syntactically correct repairs for the input string, $R^* \subset R$, will form the output of the error handler.[5]

2.1. Morpheme Incompatibility. Of all the factors contributing to lexical error repair, only *morpheme incompatibility* depends on the source language and needs therefore to be studied separately.

In previous studies the authors,[4,5,8] basing their findings on the work of other researches,[9−13] concluded that inorganic nomenclatures could be effectively represented by a context-free grammar whose semantic rules included grammar rules to deal with the context-dependent aspect of the language to be represented. The syntactic rules of the grammar must include the definition of how morpheme types may be grouped together to represent the name of an inorganic substance. At the same time, these grammatical rules define which arrangements of morphemes are not permitted by the grammar—the syntactic incompatibility set. This syntactic structure for inorganic names may be roughly defined by the following expression:[6,7,14]

inorganic-compound = inorganic-substance
+ [phonetic-string]
+ [keyword] (1)

inorganic-substance = ([multiplier] + radical)
| ([multiplier] + (element-name
| (root-symbol + name-remainder))
+ [oxidation-number])
| {([multiplier] + [prefix])
| ([prefix] | [multiplier])}
+ (root-symbol +
[root-remainder] + suffix
+ [oxidation-number])
| ([multiplier] + synonym)+
+ [phonetic-string]
+ [inorganic-substance]

where | represents selection, + means concatenation, [,] means enclose options, and {,} and (,) represent aggregation.

Table 1. Incompatibility between Morpheme Types

	SU	PR	MU	RS	RR	EN	NR	ON	RA	SY	KW	SP
SU	1	0	0	0	1	0	1	0	0	0	0	0
PR	1	1	0	0	1	1	1	1	1	1	1	1
MU	1	0	0	0	1	0	1	1	0	0	1	1
RS	0	1	1	1	0	1	0	1	1	1	0	1
RR	0	1	1	1	1	1	1	1	1	1	1	1
EN	1	0	0	0	1	0	1	0	0	0	0	0
NR	1	0	0	0	1	0	1	0	0	0	1	0
ON	1	0	0	0	1	0	1	1	0	0	0	0
RA	1	0	0	0	1	1	1	1	0	0	1	1
SY	1	0	0	0	1	0	1	1	0	0	0	0
KW	1	1	1	1	1	1	1	1	1	1	1	0
SP	1	0	0	0	1	0	1	1	0	0	0	1

Table 2. Compatible Root-Symbol/Root-Remainder/Suffix Combinations

string	root-symbol	root-remainder	suffix
sulfate	sulf		ate
arsenious	arsen	i	ous
sulfatephosphite	sulf, phosph		ate, ite

In the process of repairing lexical errors, new repair strings are built. It is the goal (usually, but not always attained) that such new strings not introduce new lexical or syntactical errors. If such new errors were to be introduced, that particular repair must be discarded and a different one examined. The process will be continued until, hopefully, a satisfactory new string is formulated. In this process it is desirable that only one subset of all morphemes in the vocabulary be used for the repair of a given error substring.

Analysis of the syntactic structure of context-free grammars defined for the inorganic nomenclatures IUPAC, Stock, and Conventional[6] (see eq 1) enables us to define an incompatibility table for tokens; this is shown in Table 1 and uses the following abbreviations to represent morpheme types: **SU**ffix, **PR**efix, **MU**ltiplier, **R**oot-**S**ymbol, **R**oot-**R**emainder, **E**lement-**N**ame, **N**ame-**R**emainder, **O**xidation-**N**umber, **RA**dical, **SY**nonym, **K**ey**W**ord, and **SP**ace.

For the sake of clarification, let us examine a few examples of the information which might be represented in this table.

If we look at the Root-Symbol, Suffix, and Root-Remainder tokens, we can see from Table 1 that Root-Symbol tokens can only be followed by Root-Remainder or Suffix tokens; Suffix tokens cannot be followed by Root-Remainder tokens, but they could be followed by Root-Symbol tokens, as illustrated by the strings shown in Table 2.

Additionally, it can be seen that the Root-Symbol tokens may be preceded by the Suffix token but not by Root-Remainder tokens such as would be the case for *sodium sulfatephosphite*, as shown in Table 2.

The definition of incompatibilities between tokens enables us to limit the number of matching operations needed for the repair of an error substring. The error handler will then work in line with the following model:

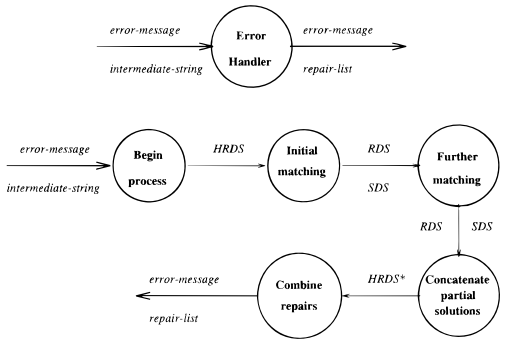

Figure 2. Context diagram of the error handler.

•All morphemes or terminal symbols of the defined grammar (those corresponding to the IUPAC, Stock, and Conventional nomenclatures and to the trivial names defined in the Translator[6]) have been classified based on the category or syntactical type they belong to.

•Where the error handler receives an input string of the type $C \equiv \{t_1t_2et_3t_4\}$, the morphemes to be submitted to the error substring repair matching process will be those which satisfy

$$I(T_{t_2},T_r) = I(T_r,T_{t_3}) = 0$$

where T_{t_i} and T_r represent the type to which the respective tokens t_i and r belong. It should be borne in mind that if a string C', obtained from an input string C repaired by the error handler, is compatible, that is where $GI(C') = 0$, this does not necessarily mean that the string C' will not contain errors which may subsequently be detected by the syntactic analyzer. What is does mean is that the syntactic structure of each of the significant parts of the inorganic name is syntactically correct. The reason for this should be fairly clear: the string C may contain syntactic errors which remain undetected at this stage in the translation process, since it is the lexical analyzer that calls up the error handler, and, at that point in the process, syntactic analysis has not yet been performed.

3. THE REPAIR PROCESS

The repair process involves a series of stages, at each of which the possible repairs obtained from the matching process are evaluated. Figure 2 contains a context diagram[15] to illustrate the steps taken by the repair process when dealing with an incorrect name for an inorganic substance. To clarify, let us take a closer look at each of these steps.

3.1. Step 1. Initialization. When the error handler is called, a data structure named the *Hierarchical Repair Data Structure (HRDS)* is initialized. This structure stores the string which has been input to the error handler, along with a string containing pointers to another data structure named the *Repair Data Structure (RDS)*, whose function is to store all the error substrings which the lexical analyzer has detected and which needs to be repaired. The *RDS* data structure comprises the following:

Repair Data Structure (RDS)	
error:	error substring
start:	starting point in the input string
end:	end point in the input string
leading-token:	type of token preceding the error substring
trailing-token:	type of token following the error substring
solutions:	total number of possible repairs
repair:	↑ list of possible repairs
next:	↑ next error substring (↑ RDS)

3.2. Step 2. Initial Matching. Matching is performed on each of the error substrings or RDS elements, following these steps.

Step 2.1. Calculation of Incompatibilities. A set of compatible token types whose instances or morphemes would be compatible with the repair is determined. This process is based on the value of the *RDS.leading-token* attribute stored in each element of the *RDS* and also on the table of token incompatibility.

Step 2.2. Calculation of Repair Length. Based on the previously-established error factor (f),[5] the maximum acceptable repair length is calculated; all longer morphemes, even though compatible, are discarded. This step makes use of the *RDS.start* and *RDS.end* attributes, which define the length of the error string. The error factor is a working variable in the error handler; its value (which lies within the interval 0 to 1) may be set according to the expected nature of the inorganic name strings to be input (e.g., maximum percent error per character, the user's ability, etc.).

Step 2.3. Calculation of Similarity. For each permited token, every possible match is performed and similarity is calculated. All viable repairs are ordered according to their S value, ruling out any repair whose value falls below that of the preestablished threshold. This process follows the model and algorithm proposed in the previous article of this series.[5]

Step 2.4. Storage of the Repair. For each of the repairs established in the previous step, a new data structure is built, named the *Solution Data Structure (SDS)*; this is linked to its corresponding *RDS* instance via the *RDS.repair* attribute and is defined as follows:

Solution Data Structure (SDS)	
token:	repair morpheme or set of morphemes
length:	length of repair string
type:	type or sequence of types of tokens included in the repair string
start:	initial position for repair of the string
end:	final position for repair of the string
image:	the Image data structure,[5] storing the result of the matching process
similarity:	value obtained for similarity
left-error:	pointer to new error string lying to the left of the repair (↑ RDS)
right-error:	pointer to new error string lying to the right of the repair (↑ RDS)
next:	pointer to the next repair (↑ SDS)

Step 2.5. Iteration. This process is repated for each of the error substrings, that is, for each instance of the *RDS*.

It can be shown that the syntactic compatibility of the repair must depend entirely on the preceding or leading token. Testing for left ($I(t,r)$) and right ($I(r,t)$) incompatibility would give rise to inconsistent or erroneous results. Let us look at an example. Take the case of the string *periadatu-*

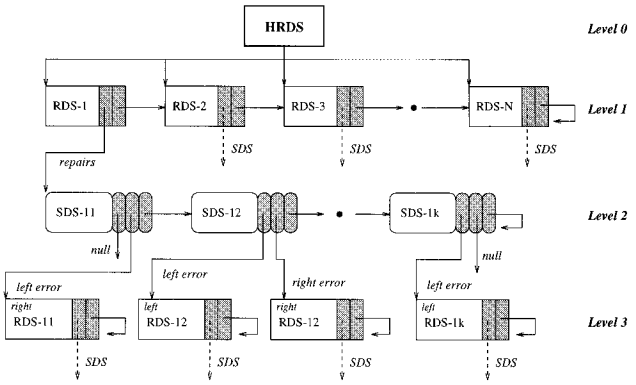

Figure 3. Outline of the repair process.

(V) ion, containing the error substring *iadatu* which is preceded by the string *per* (prefix) and followed by the string *(V)* (oxidation-number). If we analyze the incompatibility between tokens, we have the following:

•The string *(V)* may only be preceded by suffix and element-name tokens.

•The string *per* may only be followed by root-symbol and multiplier tokens.

Now if we calculate the intersection of the two sets: {*suffix*, *element-name*} ∩{*root-symtol*, *multiplier*} = {0}, we get an empty set, which would mean that three were no compatible types of tokens for use in the repair. Furthermore, if we calculate the union of both sets: {*suffix*, *element-name*} ∪ {*root-symbol*, *multiplier*} = {*suffix*, *element-name*, *root-symbol*, *multiplier*}, we get a set of token types which are incompatible with the leading or trailing tokens; the repair(s) obtained will thus be incorrect, such as would be the case in the concatenation of *prefix* + *suffix*.

Accordingly, to calculate token incompatibility we only test for leftwise incompatibility ($I(t,r)$), using the *RDS.leading* attribute, and, when the repairs have been obtained, their rightwise compatibility ($I(r,t)$) is checked and incompatible repairs are discarded. In this way, the repair process finds solutions that repair the error string from left to right, as we shall see further on.

The possible repairs found in *step 2.5* are stored in an *SDS* data structure, thus the following.

Figure 3 outlines how the various repairs obtained for each of the error substrings are stored in the input string. These solutions may be considered as either total or partial. A solution is classed as total when it produces no further error strings to the left or right of the repair string. For every such total solution, a check is made for incompatibility with the trailing token of the input string. This process is performed using the *RDS.trailing* attribute of the *RDS* instance corresponding to that particular repair. Any incompatible total solutions are discarded.

For each partial solution, two new (possibly empty) error strings are created, located to the left and/or right of the repair string. These new strings are stored in *RDS* data structures in the same way as the error strings received by the error handler. Figure 3 illustrates how a hierarchical structure, recursive with respect to data types, is built in order to store the solutions proposed by the repair process. The first level stores the error substrings contained in the string that is input into the error handler. This level *(level 1)* branches down *(level 2)* to all the possible repairs or total solutions for the substring corresponding to each *RDS* node. Each partial repair—the *SDS* nodes at *level 2*—will then lead down to the new error substrings generated by the repair process; that is, strings situated to the left and/or right of the repair string *(level 3)*. Once again, the *RDS* nodes in *level 3* will branch down into repairs for these new substrings and so on.

All repairs (instances of the *SDS*) produced in the previous step for each error substring (each instance of the *RDS*) are ordered according to the following criteria:

1. according to whether the repair is total or partial, in other words, depending on the length of the error string under repair

2. partial repairs: ordered according to their position in the error substring, starting from the left

3. according to their similarity value, as derived from the matching process.

3.3. Step 3. Successive Matching. For every error substring generated by each of the partial solutions obtained in step 2, a new matching process is performed, the same as in step 2, and where the following are applied.

1. Firstly, repairs are sought for the error substring located to the left of a partial repair, and if the process generates another error substring on the left, the process is once again applied to this string. This procedure is repeated until no new error substrings are produced on the left. The error substring on the right-hand side of the partial repair is then dealt with in the same way.

If we visualize the solutions to the error repair process as forming a tree (Figure 3), then repairs are obtained by moving round the *prefixes* of the tree, always attempting to repair the leftmost child node (an input error substring or one generated by a partial repair).

2. The repair process checks for incompatibility of types and the size of the new repair set is dictated by the length of the new error string, as described above. Checking of types is always performed by the calculation of left-side incompatibility ($I(t,r)$), where t is a token recognized by the lexical analyzer (nodes to the left) or a previously obtained partial repair (nodes to the right).

3. This procedure is performed for each of the partial repairs p_{ij} obtained during step 2 for each error substring e_i found in the error handler input string.

3.4. Step 4. Concatenation of Partial Solutions. Once the above procedure has been concluded, concatenation of partial solutions takes place. In this process, the *HRDS* data structure is examined so that any partial repair generated for the left-hand side of an error substring is concatenated to the left of the repair set, and the same procedure is applied to the right.

In the concatenation process, the following occurs:

•A compatibility check is made of morphemes to be concatenated. Where concatenation would result in incompatibility, the partial repair in question is eliminated from the *SDS* structure.

In the process of finding repairs for an error substring generated on the left-hand side by a partial repair p_{ij}, morpheme compatibility is checked against the morpheme preceding e_i in the original input string, namely against $I(t_k, r^I_{i_{j1}})$, and not against the previous partial solution $I(r^I_{i_{j1}}, p_{ij})$. Partial repairs for error substrings generated on the right, however, are compared with the previous partial repair.

•For each repair—the product of concatenation—a similarity value is obtained. The fact that our model takes into account the possibility of overlapping morphemes in the error string means we have to recalculate similarity values during the process of concatenating successive partial solutions.

•A new *HRDS** structure is built (see Figure 2); here, every instance of the *RDS* structure (the error substring(s) contained in the input string) is assigned as many instances of *SDS* structures (solutions found by the repair process) as have been produced from

- −the total repairs obtained during step 2 for each error substring e_i
- −the total repairs obtained through concatenation (Step 4) of successive partial solutions for each partial repair obtained in Step 2.

•Since the process of deriving total solutions through the concatenation of successive partial solutions may lead to the appearance of redundant repairs, this redundancy is eliminated by only recognizing repairs derived from a single set of repair morphemes.

•The instances of *SDS* for each *RDS* are ordered according to their stored similarity value.

3.5. Step 5. Combination of Repairs. The new *HRDS** structure built in Step 4 now contains the set of repairs whose similarity is greater than that of the predefined threshold and which will achieve total repair of each of the error substrings. At this point, all possible repairs for each error substring e_i may be considered to be possible corrections, and we can generate the set of repairs which will achieve global repair of the entire string *C*—the original string fed to the error handler and composed of a number *n* of e_i error substrings.

The set of total repairs for the string *C* is formed by the product of the repair sets for each error substring e_i, thus

$$R^* = R_1^* \times R_2^* \times R_3^* \times \ldots \times R_n^*$$

where *R** is the set formed by new strings *C** that repair the error string *C*—the input to the error handler—and $R_i^*$is the set composed of the strings that repair the error substring e_i which is part of the string *C*. All the R_i^*sets are formed by globally compatible repairs, yet it should be borne in mind that syntactic errors may be present in the strings forming part of the set *R**, as discussed in the previous section.

The elements of the set *R** are stored in a sequential list, in which global repairs are ordered according to the final similarity value of the repair. This value is derived from the similarity of each repair string produced, using the following expression

$$S(r_i^*) = \min(S(r_{1_j}^*, r_{2_k}^*, r_{3_l}^*, \ldots, r_{n_p}^*) \qquad (2)$$

in which we examine the independence of two error strings e_i and e_j in a string *C*.

Thus, the similarity of a total repair for the error string *C* containing *n* error substrings e_i will be equal to the minimum value for the similarities of the repairs considered for each e_i in the new repair string *C**.

4. PROBLEMS IN THE REPAIR PROCESS

The characteristics of the various nomenclatures, together with the error repair process itself, will give rise to many and varied problems when seeking morpheme strings to be used for the syntactically-correct replacement of an error string. The next section shall attempt to examine some of these problems.

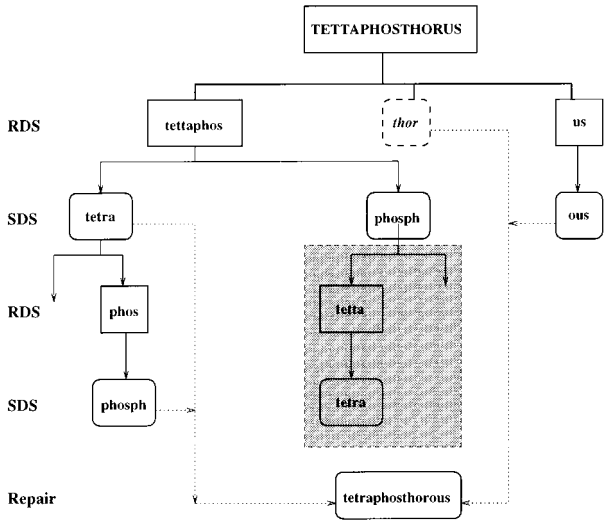

Figure 4. *RDS* for the string *tettaphosthorus*.

4.1. Mistaken Lexical Recognition. As mentioned in foregoing sections, the presence of an error, usually typographic, may go unnoticed or be hidden by other errors during lexical analysis, and only symptoms corresponding to other errors will be observed. By way of example, let us examine the following.

Suppose the string *tettaphosthorus* were input instead of the string *tetraphosphorus*. It can be seen that two errors have occurred: the character "*r*" has been substituted by "*t*" in the token *tetra*, and the "*p*" by "*t*" in the token *phosphorus*.

When the lexical analyzer examines this string, it will recognize the lexeme *thor*, the root-symbol token for the element *thorium*, and will detect two error strings: the string *tettaphos* and the string formed by the characters "*us*".

As we can see, the existence of an error (substitution of the character "*p*" by "*t*") has gone unnoticed by the lexical analyzer, which has recognized the lexeme *thor*, even though this is the consequence of an error.

When the error handler tries to find repairs for the two error strings, the following situation may arise:

•For the error string "*us*", the error handler will find a suitable solution in the string *ous*, a suffix-token which can legitimately follow the root-symbol token representing the string *thor*; other repairs such as the name-remainder morpheme *ium* would have a lower similarity value.

•For the string *tetta*, the solution *tetra* will be found: this is a multiplier token and may precede a root-symbol token such as *thor*.

•For the string *phos*, a suitable solution will be the root-symbol token *phosph*, which may follow the previously-obtained multiplier string *tetra*. Figure 4 illustrates the repair and *RDS* structure for the above example.

Now, while these solutions may at first appear compatible, the subsequent global solution will highlight a compatibility problem, since the string *phosph* cannot be followed by the string *thor*, as both are root-symbol morphemes and thus incompatible (*I*(*root-symbol*, *root-symbol*) = 1).

In this kind of situation, the Translator[6] would initially be unable to recognize the input string, thus giving rise to an error at the output stage.

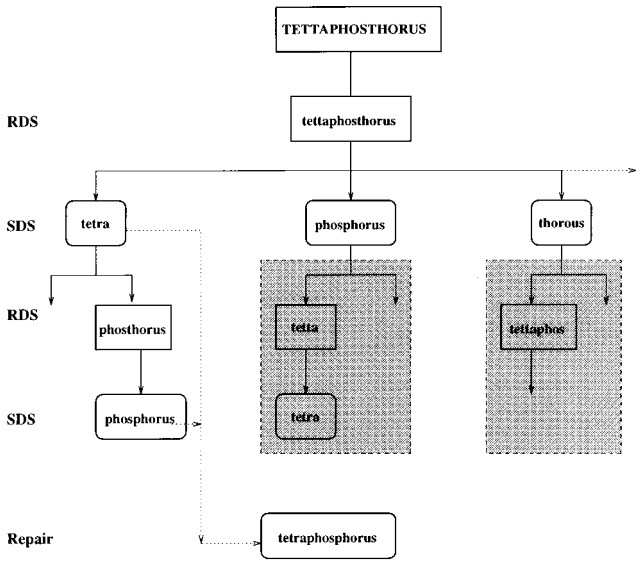

Figure 5. *RDS* for the string *tettaphosthorus*, using backtracking.

Two possible methods for the solution of this kind of situation are now described; note that these are not necessarily mutually exclusive:

1. *Delete the substring(s) responsible for this incompatibility from the syntactically-incorrect repair string*. In the above example, it would be enough to delete the root-symbol string *phosph*, which here accounts for incompatibility. Subsequently, the string *tetrathorous* would be proposed as a repair; while this string is not correct (i.e., it is not what the user meant to enter), it is syntactically valid.

2. *Perform lexical backtracking*. If, however, we bear in mind that repair incompatibility may be due to mistaken identification of morphemes by the lexical analyzer, we may consider strings recognized by the lexical analyzer as error strings and go on to repair them as if they were any other error string.

In the most general of cases, we would have to consider the input string to the error handler as an error string, and the error handler would then be responsible for its identification. In fact, the error handler and the lexical analyzer perform similar duties. Though it would do so much less efficiently, the error handler may carry out lexical analysis, since both processes are basically matching processes. One, however, demands exact matching, while the matching of the other is only approximate.

For the proposed example, suppose that the string *tettaphosthorus* is a mistake; a simplified verion of its repair is shown in Figure 5, where the string *tetraphosphorus* would be the repair with the highest similarity value.

Naturally, it would be rather optimistic to hope that backtracking would produce repairs as simply as in the above example. Backtracking involves supplying a set of possible repairs instead of merely reporting an error with no possibility of repair.

Instead of saying that the entire string is wrong, which would lead to an excessively costly repair process due to the great number of matchings that might have to be performed, the process may be simplified and thus the following:

1. perform lexical backtracking only on the leading or trailing token, with respect to the error

2. carry on with the above-described repair process and

(a) If syntactically-correct repairs are found, end the process.

(b) Otherwise, go back to the previous point and examine new tokens to be included in the error string.

This method will obviously take up less processing time if the error is corrected in the first phase (as would be usual in the present example, where the error is due to overlapping of two morphemes, e.g., the transposition of a morpheme's terminal character with the first character of the following morpheme). This does, however, lead to more complicated programming.

The error handler is not an expert system. It is a purely procedural system and contains no knowledge base through which it could explain an absence of compatible repairs. This lack of repairs may be due to an error factor close to zero, an unintelligible input string such as *psiu pmnajntx*, a very high similarity threshold, mistaken lexical recognition by the lexicographic analyzer, or other factors not considered by the proposed model. The use of lexical backtracking on the input string will increase the probability of generating "correct" repairs if mistaken lexical recognition has taken place, although it will result in increased computing time. The user may also modify the system variables (i.e., by increasing the error factor and/or reducing the similarity threshold) in an effort to find compatible solutions.

4.2. Semantic Inconsistency. As mentioned beforehand in the introduction, the error handler will supply syntactically-correct repairs, but these may not be semantically correct. This means that the error handler will produce a list of possible repairs for an input string containing error substrings; the list will include repairs which will produce further errors during semantic analysis. Let us look more closely at one such example:

Suppose we have an input string *sodium sulfire*, containing the error substring *ire*. The error handler will supply the following strings, each of which has the same similarity value: *ide*, *ine*, and *ite*; others with a lower similarity value will also be supplied, such as *ate*, *ane*, *ino*, etc. An incompatibility error will be produced in the semantic analyzer due to the strings *sodium sulfide* or *sodium sulfine*, whereas the string *sodium sulfite* will be correctly identified.

Now, if we input the string *sodium sulfire(VI)*, which contains the same error substring *ire*, then the repairs *ide*, *ine*, and *ite* would all produce a semantic error, owing to the presence of the oxidation-number morpheme *(VI)*, which states that the element *sulphur*—represented by the morpheme *sulf*—has a valency of +6. While the repair string *ate* would be syntactically and semantically correct, the error handler would not view this repair as having priority, since its similarity value is lower than that of *ide*, *ine*, or *ite*.

This, then, is one of the current limitations of the error handler, since the defined semantic rules are not incorporated into the parser's method of syntax production.

As long as the design and execution of semantic compatibility control procedures are the responsibility of the error handler, the output will be a set of all possible syntactically-correct repairs, whose use and management will then be used and managed by the program.

The system may present a list of all possible repairs, leaving the final decision to the user. It could also call the parser for each repair, in descending order of similarity, and

Table 3. Identification of the String *chlor_ic_acid*

morpheme	morpheme type
chlor	root-symbol
_	space
ic	suffice
_	space
acid	keyword

Table 4. Errors Caused by Absent or Excess Spaces

situation	example
case 1	copper_(II)_oxide sodium_bi_sulfate
case 2	copper(II)oxide sodiumbisulfate
case 3	sulfuric_ac_id copp_er(II)_oxide
case 4	magnesium_oxi_de nitric_anhy_dride
case 5	sodium_alumina_te bismuth_yl

present the user with any repair that has not produced an error in the recognition of the output string.

4.3. Delimiters and Spaces. Spaces are alphabetical characters belonging to the grammar of the working nomenclature and whose only function is to separate significant "parts" of the string representing an inorganic chemical name.

When a string passes through the orthographic analyzer,[6] redundant spaces are stripped, leaving only one space between characters.

However, when the redundancy of these spaces is due to their insertion between significant "parts" that cannot be separated from the chemical name, we have an error that is particularly difficult to deal with.

Suppose we have an input string such as *chlor_ic_acid*, where a space has been inserted between the root-symbol morpheme *chlor* and the suffix morpheme *ic*. (For clarification, spaces are represented by the underline character *(_)*.)

When the lexical analyzer processes this string, the substrings in the input string are correctly recognized, as seen in Table 3, but the error caused by the space inserted between the root-symbol and its suffix is also detected; in this case there could only be a root-remainder string, so the extra space is automatically eliminated and the corresponding error message is generated alongside the new repair string.

The insertion of spaces at the beginning and end of the string is also easily fixed, as these spaces will be deleted by the Translator.[6] The real problem occurs when there are situations such as those listed in the Table 4.

In cases 1 and 2, the lexical analyzer would not detect an error. The problem arises, however, at the syntactical analysis stage. ln strings such as these, there is no loss of knowledge, but the presence of extra or missing spaces leads to a structure that is linguistically incorrect in the syntactic sense. Nevertheless, a syntax analyzer whose production rules did not look for the *space* token would not detect these errors.

In fact Type I errors, which according to our classification would not produce any loss in chemical knowledge—as discussed in the first article in this series[4]—are ignored. The lexical analyzer does not transfer spaces to the parser, so the syntactic rules defined in the grammar are not concerned with syntax structure from the linguistic point of view. Input strings such as those depicted in Cases 1 and 2 would thus be recognized regardless.

The strings in Cases 3 and 4 create very different problems. In both cases, the lexical analyzer will detect errors and call up the error handler. The problem arises if two error strings separated by a space are sent to the handler, which would then have to repair these individually; this may then require backtracking in order to produce syntactically-correct repairs. In such cases, however, the use of backtracking may provide repairs that are far from the desired result. For example, for the string *magnesium_oxi_de*, which contains the error strings *oxi* and *de*, repairs such as *oxo* and *di* might be produced; these have a high similarity value but are utterly useless as far as the user is concerned. Likewise in the case of the string *nitric_anhy_dride*, which includes the error strings *anhy* and *dride*, separated by a space. Supplied repairs would be of the nature of *anion*, *hydride*, *hydric*, *etc*.

By far the most complicated of all is Case 5. The insertion of a space was created two significiant "parts", one of which is recognizable to the lexical analyzer as a lexeme pertaining to the grammar, and the other will simply produce an error. The direct repair of such cases is difficult and will only be accomplished using lexical backtracking.

For the repair of errors where excess spaces are present the handler acts as follows:

•The space is considered to be a "part" of the inorganic substance name, only in order for incompatibility to produce an entry in the incompatibility table, as shown in Table 1.

•If the error handler receives an input string containing a space immediately before or after an error substring, the space is eliminated. This may mean that two error substrings are joined together, while they were separated by a space in the input string.

Take, for example, the string *magnesium_oxi_de*, in which the two error substrings *oxi* and *de* are separated by a space; if we delete this space we are left with a single error string—the substring *oxide*—which will be repaired with a maximum similarity value by the morpheme *oxide*.

In cases such as that of Case 5, elimination of the space from the substring *bismuth_yl* will highly favor the repair process, even though the number of error strings is not reduced.

Such incompatibility in the case of spaces allows us to propose a revised output string where spaces have been deleted or inserted according to the types of tokens contained in the string. Although this process bears no influence on the subsequent syntactic-semantic analysis, it certainly produces a repair which is extremely interesting from a linguistic aspect. This process should favor the use of the error handler in a computer-aided learning system for inorganic formulation or in any other system that would make use of this knowledge.

5. DISCUSSION

This series of articles has presented a general study of the problems involved in the detection, reporting, and repair of errors occurring during the process of translating knowledge between its various representations; a model for the repair of lexical errors contained in inorganic names has been proposed,[4,5] and the present paper has presented the system designed to perform such a process. The system attempts to repair lexical errors detected by the translator system for

the conversion of inorganic nomenclatures into a data structure (DIMS) suitable for computer processing.[6]

The error handler has produced excellent results and in tests has managed to repair almost all common lexical errors and even some strings containing grossly exaggerated or unrealistic errors. The sensitivity of the matching algorithm, the model for calculation of similarities and the stepwise repair architecture would all seem to provide, in addition to a successful repair, an expansive amount of information about the process itself. This latter point, in particular, would point to the suitability of this model and system for any system demanding powerful communication between user and machine, such as computer-assisted learning; the authors are currently working on the application of the above results to the area of inorganic reactions.[16,17]

Although in the present study the proposed model has been applied to lexical errors in inorganic names, it could equally be applied to any problem in which this type of error has to be dealt with, simply by customizing the features of the language used for the human-machine interface.

REFERENCES AND NOTES

(1) Kukich, K. Techniques for Automatically Correcting Words in Text. *ACM Computing Surveys 24(4)* **1992**, 377−439.
(2) IUPAC. *Nomenclature of Inorganic Chemistry*. *Recommendations 1990*; Blackwell Scientific Publications: 1992.
(3) *Chemical Nomenclature Usage*; Lees, R., Smith, A. F., Eds.; Horwood: Chichester, UK, 1983.
(4) Luque Ruiz, I.; Cruz Soto, J. L.; Gómez-Nieto, M. A. Error Detection, Recovery and Repair in the Translation of Inorganic Nomenclatures. 1. A study of the problem. *J. Chem. Inf. Comput. Sci*. **1996**, *36*, 7−15.
(5) Luque Ruiz, I.; Cruz Soto, J. L.; Gómez-Nieto, M. A. Error Detection, Recovery and Repair in the Translation of Inorganic Nomenclatures. 2. A Proposed Strategy. *J. Chem. Inf. Comput. Sci*. **1996**, *36*, 16−24.
(6) Luque Ruiz, I.; Cruz Soto, J. L.; Gómez-Nieto, M. A. Computer Translation of Inorganic Chemical Nomenclature to a Dynamic Abstract Data Structure. *J. Chem. Inf. Comput. Sci*. **1994**, *34(3)*, 526−533.
(7) Luque Ruiz, I.; Cruz Soto, J. L.; Gómez-Nieto, M. A. Inorganic Chemical Knowledge Representation Using Dynamic Data Structures. *J. Chem. Inf. Comput. Sci*. **1993**, *33(3)*, 378−384.
(8) IUPAC. *Nomenclature of Inorganic Chemistry*. *Recommendations 1958*; Blackwell Scientific Publications: 1960.
(9) Aho, A. V.; Sethi, R.; Ullman, J. D. *Compilers: Principles, Techniques and Tools*; Addison-Wesley: Massachusetts, U.S.A., 1986.
(10) Fischer, C. N.; Leblanc, R. J. *Grafting a Compiler*; The Benjaman Cummings Publisher: 1988.
(11) Lewis, P. M.; Rosenkrantz, D. J.; Stearns, R. E. *Compiler Design Theory*; Addison-Wesley, 1976.
(12) Cooke-Fox, D. I.; Kirby, G. H.; Rayner, J. D. Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 1. Introduction and Background to a Grammar-Based Approach. *J. Chem. Inf. Comput. Sci*. **1989**, *29*, 101−105.
(13) Cooke-Fox, D. J.; Kirby, G. H.; Rayner, J. D. Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 2. Development of a Formal Grammar. *J. Chem. Inf. Comput. Sci*. **1989**, *29*, 106−112.
(14) Luque Ruiz, I.; Cruz Soto, J. L.; Gómez-Nieto, M. A. *Knowledge Representation through Objects in the Development of Expert Systems for Chemical Synthesis and Reactions*. Advanced Methods in Artificial Intelligence. Lecture Notes in Computer Science 682. Springer Verlag: 1993.
(15) Yourdon, E. *Modern Structured Analysis*; Prentice-Hall: 1989.
(16) Luque Ruiz, I.; Gómez-Nieto, M. A.; Arribas Machuca, E.; Durán Vacas, E. Design and Development of a Computer-Aided System for the Following and Adjustment of Inorganic Reactions. E. U. Politécnica, University of Córdoba, Technical Report, Project-198-1995, Córdoba, Spain, 1995.
(17) Luque Ruiz, I.; Cruz Soto, J. L.; Gómez-Nieto, M. A. A Tutorial System on Inorganic Chemical Formulation and Reactions. ACM-SAC'94, Proceedings of Symposium on Applied Computing; Phoenix Az, 1994.

CI9502257