SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(7), 603–622 (JULY 1994)

Signature-check Based Unification Filter

ALFREDO M. MAEDA, JUN-ICHI AOE AND HIDETO TOMABECHI
Department of Information Science and Intelligent Systems, University of Tokushima, 2-1 Minami Josanjima Cho, Tokushima-Shi 770, Japan (email: maeda@j-aoe.is.tokushima-u.ac.jp)

SUMMARY

Among the different processes that entail unification-based grammar parsing, the unification of feature structures is by far the most expensive one in terms of execution time. Unification of the feature structures of a given sentence typically takes between 85 and 98 per cent of the total elapsed time during parsing, thus the need to develop faster unification methods. The approach presented in this paper is based on the fact that, in general, between 60 and 85 per cent of unifications attempted in a typical parse result in failure. Our claim is that the efficient treatment of such unification failures reduces unification time significantly. In this paper we present what we call a unification filter or U-filter, that preprocesses the feature structures to be unified. If the U-filter succeeds, unification is then skipped because the attempt to unify the involved structures would result in failure. On the other hand, when the U-filter does not succeed it is not possible to determine at that moment whether or not the structures unify, so unification is performed. The U-filter stops around 87 per cent of unification failures, and speeds up unification time by an average of around 29 per cent over quasi-destructive graph unification, the fastest unification method known so far.

KEY WORDS Graph algorithms Natural language processing and understanding Parsing Pattern matching Unification Unification filtering

INTRODUCTION

The most expensive operation, in terms of time consumption, in unification-based grammar parsing systems is, by far, the unification of feature structures. By using well-known parsing algorithms, such as Earley's[1] or Tomita's,[2] it is rather common to have unification operations take as much as 95 per cent of the total time required to parse natural language phrases. In the case of the JPSG[3]—the Japanese phrase structure grammar—based on HPSG[4,5]—the spoken Japanese analysis system developed at ATR—unification operations reached up to 98 per cent of the total parsing time for some sentences.[6] This number is also affected by the fact that ATR's system is designed to cover a large number of linguistic phenomena, thus requiring more unifications to be performed. This brings about what is better known as the unification paradox. Namely, for a given natural language (NL) system, as the coverage of linguistic phenomena increases and the NL system gets larger, the number of unifications to be performed increases so rapidly that the overall performance of the system degrades drastically. This justifies the urgent necessity to develop more efficient and faster unification schemes.

The degree of sophistication and efficiency of unification algorithms has become

CCC 0038–0644/94/070603–20

Received 8 December 1993
Revised 8 March 1994

notably high and they are also used for code optimization,[7] especially because of the treatment efficiency of copy propagation and function-preservation. Nevertheless, unification remains the bottleneck of unification-based grammar parsing systems. This is true basically because of the tremendous number of graphs to be tested for unification, especially if the grammar is designed to cover a wide range of linguistic phenomena. Since typically between 60 and 85 per cent of the unifications performed in a typical parse result in failure, the attempt to unify them only generates overload to the unification process. Hence the need for a mechanism able to deal with unification failures efficiently. The main claim here is precisely that, empirically, the efficient treatment of unification failures should result in a significant speed-up of unification time. The resulting scheme based on this observation is the unification filter (henceforth U-filter).

UNIFICATION SCHEMES

Vast research on the solution of the graph unification bottleneck has been done since the development of the unification formalism itself. An early efficient unification method called structure-sharing[8] consists of sharing the information, in the directed graphs, that remains unmodified after unification. Such sharing of structures replaces the time consuming operation of graph copying by the dynamic creation of the resulting graph whenever needed. Since graph creation occurs each time the graph is needed, an undesirable log(d) overhead exists for all node accesses to update information in the graph being unified. Reversible unification[9] consists of saving away the original graph prior to a destructive change, allowing the 'undo' of destructive changes. This approach, in spite of avoiding the log(d) overload inherent in structure sharing, conveys a double cost, namely the saving of structures prior to unification and the reversing of unification. Both operations are proportionate to the size of the input graph.

Incremental copying, a scheme called non-destructive graph unification,[10] consists of the incremental creation of copies as such need arises. This scheme consists of the combination of both a destructive and a non-destructive unification algorithm in which copies are created incrementally. Non-destructive unification eliminates two important costs, namely the fixed cost overhead log(d) for node access and the cost for reserving destructive changes through reseting the copy field by means of invalidation. Non-destructive graph unification, however fast, has the disadvantage of generating over-copy of structures. Over-copy occurs when two graphs are copied in order to create a new graph.

Lazy incremental copy[6] avoids the copying of unchanged subgraphs by using dependency pointers, and needed copies are created from the lower subgraph. This scheme allows the copying of nodes whose subgraph was never modified to be avoided, but conveys two main disadvantages. It requires the copy dependency pointers in every node of the entire graph to be maintained, and requires bidirectionality in the entire directed graph to double-traverse modified graphs, i.e., once to unify and once to copy the graphs. Lazy non-redundant copying[11] is a combination of structure sharing and incremental copying. Incremental copying takes place in a way similar to non-destructive graph unification,[10] however, the copying of nodes is delayed until the moment when a destructive change is about to happen. Such a delay is controlled by 'chronological dereference chains',[11] which is a data structure independent of the actual unification algorithm, making it possible for implementation into any other unification scheme. Unfortunately lazy non-redundant copy adopts the disadvantages of its schemes base, namely incremental copying and structure sharing overload.

A fast variation of non-destructive graph unification, called quasi-destructive graph unification[12,13] (QD) is based upon the notion of time-sensitive quasi-destruction of node structures. QD eliminates both early-copying and over-copying of graphs, and runs in about half the time of non-destructive graph unification. In early-copying copies are created prior to the failure of unification, so that copies created since the beginning of the unification up to the point of failure are wasted. Quasi-destructive graph unification with structure sharing[13,14] (QDS) is basically a QD scheme with structure sharing included. QDS eliminates redundant copying of shared subgraphs while avoiding both over-copy and early-copy. The result is an extremely fast algorithm that runs in about 23 per cent of the time compared with non-destructive graph unification, making QDS the fastest unification algorithm known so far. As will be shown later, the inclusion of our U-filter into unification-based grammar parsing systems as a pre-unification process allows unification operations to be processed significantly faster.

SIGNATURES

A signature is a coding approach commonly used for text retrieval. Word signatures hash each word of a document into a fixed-length bit pattern, i.e. a word signature.[15] The resulting patterns are then concatenated to form the document's signature. Searching operations are performed by extraction of a particular signature and its comparison with all the document's signatures. One of the most common signature schemes is superimposed coding.[16,17] The underlying idea in this scheme is to map attributes into random m-bit codes in an n-bit field. The codes are then superimposed for each attribute that is present in a record. Other approaches for text retrieval include finite state pattern matching machines[18] and the computation of longest common subsequences.[19]

The incorporation of signature-checking methods in unification is rather cumbersome. In general, signature checking and other approaches to text search are concerned only with the finding of occurrences of a sequence within a text, independently of the context or the logical position of such sequence within the text. On the other hand, the logical position and the associated value of the different features in a graph are of primary importance in unification, mainly because a given feature may appear more than once within the same graph. This difference makes signature schemes uneasy to implement as part of the unification process.

PROPOSED APPROACH

The underlying idea in the U-filter scheme consists of preprocessing, i.e. U-filtering, the F-structures, or graphs, to be unified prior to unification itself. If the U-filter succeeds, unification is then skipped because the attempt to unify the involved structures would result in failure. On the other hand, when the U-filter does not succeed it is not possible to determine at that moment whether or not the structures unify, so unification is performed. The U-filter introduced here consists of two main parts, namely the creation and inclusion of unification signatures (henceforth U-signs) into the parsing system's grammar, and the unification filtering itself.

Unification signature data structure

Well-known text search approaches are not directly applicable to unification, basically because these do not take into account the logical position, or context, of the string being searched. However, since signature-check approaches to text-retrieval are highly efficient, to adapt such approach into unification is attractive.

To make U-filtering possible, the addition of a signature slot into the graph's data structure is required. Since directed graphs are normally used for graphic as well as computational representation of feature structures (henceforth f-structures) the terms graph and f-structure are used indiscriminately throughout this paper. An f-structure is composed of three basic types of nodes: *atomic* nodes, *complex* nodes and *top*, or variable, nodes. *Atomic* type nodes are f-structures with constant atomic symbol value. *Complex* type nodes are f-structures that contain f-structures, i.e. *atomic, complex* or *top* nodes. *Top* nodes are special *f*-structures with empty domains, i.e. variables. In Figure 1, for example, nonc and moshimoshi-hello are atomic nodes, the nestings of f-structures under substructures such as head or prag are representations of complex nodes, and variable nodes are represented as '[ ]'. Additional information on f-structures is shown in Figure 1. Re-entrancy, or multiple path

```
X01 [1  X02 [head   X03 [cform  X04 senf ]                      ]]
            [            [ctype  X05 nonc ]                      ]
            [            [pos    X06 v    ]                      ]
            [subcat  X07 end                                     ]
            [sem    X08 [recp   X09[]                  ]         ]
            [            [agen   X10[]                  ]         ]
            [            [reln   X11 moshimoshi-hello   ]         ]
            [prag   X12 [speaker  X10 ]                          ]
            [            [hearer   X09 ]                          ]
            [            [restr    X13[] ]                        ]
            [slash  X14[]                                        ]
    2  X15 [prag   X16 [restr    X17[] ]  ]
            [            [hearer   X09   ]  ]
            [            [speaker  X10   ]  ]
            [sem  X18[]                     ]
            [subcat  X19 [first  X02   ]    ]
            [             [rest   X20[] ]    ]
            [head  X21[]                    ]
    0  X22 [prag   X23 [restr  X24 [rest   X13 ]]]
            [            [           [first  X17 ]]]
            [            [hearer   X09            ]]
            [            [speaker  X10            ]]
            [sem  X18                              ]
            [slash  X14                            ]
            [subcat  X20                           ]
            [head  X21                             ]
```

Figure 1. An f-structure with atomic, complex, and top nodes

sharing, is represented as X_{nn}, where $nn \geqslant 0$. The indexes 0, 1, ..., identify the f-structure for the different constituents of linguistic situations in accordance with the HPSG formalism.[5] For example, for the situation $S \rightarrow$ NP VP, the constituent structure X0 → X1 X2 is indexed as 0 → 1 2, where X1 is the head constituent.

After analysis of the nature and structure of f-structures, the use of a very simple, but powerful, unification signature for U-filtering was decided. The U-sign of a given f-structure consists of a list that encloses all the atomic and complex nodes of the f-structure's head constituent, while ignoring all occurrences of top nodes. The U-sign is defined as follows:

Definition 1

A unification signature is an ordered structure composed of all the atomic and complex nodes of the head constituent of an f-structure, keeping their content and context within the f-structure.

List structures are easy to operate, and their embedding is simple, allowing a straightforward representation of f-structures. U-signs are alphabetically ordered to allow a simpler and faster scanning during the filtering process. Top nodes are ignored because they are dynamic and vary in content during unification. Re-entrancy is ignored because it is also dynamic and, in addition, because it deals with the f-structure of all the constituents of the situation to be unified, and not with the head constituent only. Consequently, the filter only ensures that the structures are compatible according to content and context within the head constituent of the f-features to be unified. This compatibility check process is actually the same as that performed during unification. However, in contrast to unification, the U-filter does not perform the update of top nodes nor the expansion of structures that takes place only if content and context do not fail. Analysis of the atomic and complex nodes of head constituents is enough for the purpose of the U-filter. For example, the U-sign shown in Figure 2 corresponds to the f-structure in Figure 1.

Note that the signature is an alphabetically ordered list, includes only information from the head constituent of the corresponding linguistic situation, and contains neither top nodes nor re-entrancy specifiers. Since U-signs are rather small and conform only an extra slot in the graph's data structure, the amount of extra storage needed is minimum; besides, no time is wasted because the grammar U-signs are

```
((head (cform senf)
       (ctype nonc)
       (pos v))
 (prag (hearer)
       (restr)
       (speaker))
 (sem (agen)
      (recp)
      (reln moshimoshi-hello))
 (slash)
 (subcat end))
```

Figure 2. Unification signature (U-sign) of the f-structure shown in Figure 1

created and stored beforehand as part of the grammar. The overload for creation and sorting of the input sentence is immediately compensated for by the U-filtering timing itself.

Unification signature creation algorithm

The U-sign creation algorithm consists of a main function, create-sign( ), two complementary functions to conform the signature, get-sign1( ) and get-sign2 ( ), and a function that sorts alphabetically the signature, sort-signature( ). A COMMON LISP version of the U-sign creation algorithm is shown in Appendix I. These functions are described in detail next.

The function create-sign( ), listed in Figure 3, has as parameter the data structure, i.e. a list of lists, that corresponds to the head constituent of the graph whose U-sign is to be created. The components of the head constituent are sent, one at a time, to get-sign1( ) in order to build the corresponding portion of the U-sign. Each portion is received back by create-sign( ) to be processed accordingly and to be built up into one single list. Such list processing consists basically of analysing the data type of the head and tail* of the sublists in order to build a sound unification signature. Once the head constituent has been completely analysed, the resulting list is sorted alphabetically to obtain then the U-sign for that particular graph.

The functions get-sign1( ) and get-sign2( ) work together in the selection of the different elements of the sublist received from create-sign( ) to build a portion of U-sign. Get-sign1( ) is responsible for the treatment of atomic and top nodes, whereas get-sign2( ) is responsible for the treatment of complex nodes.

```
FUNCTION create-sign (dg);
   sign ← nil;
   signature ← nil;
   IF empty structure THEN
     return (nil);
   ELSE FOR EACH subdg IN dg DO
           sign ← get-sign1((subdg tail); nil);
           IF (sign type = list) AND (sign head ≠ atom) THEN
             sign ← append ((list (subdg head)), sign);
           ELSE IF (sign type = list) THEN
                   sign ← append ((list (subdg head)),
                                  (list sign));
           signature ← append signature, sign;
   signature ← sort-signature(signature);
END
```

Figure 3. Main function to create a signature: create-sign( )

* Lists and structures, analysed as lists, are divided into a *head* and a *tail*. The *head* is the first element of the list, and the *tail*, which is also a list, contains the rest of the elements of the list.

In get-sign1( ), listed in Figure 4, atomic nodes are taken as such and incorporated later into the corresponding sublist. Top nodes are simply ignored, i.e. not included into the U-sign. Complex nodes are divided into sublists and analysed accordingly. If the sublist is again a complex node then it is sent to get-sign2( ) for processing and the results are handled again in get-sign1( ) accordingly. If the sublist is not a complex node then it is sent recursively to get-sign1( ) for further processing.

In get-sign2( ), listed in Figure 5, the sublist is first sent back to get-sign1( ) to be processed as an independent list, and the returned list is then incorporated into a subsignature. The resulting subsignature is sent back to get-sign1( ) to be included in the U-sign. Note that the U-sign is built recursively from the innermost nesting to the uppermost nesting of the head constituent.

The resulting list from get-sign1( ) is finally sent to sort-signature( ), listed in Figure 6, where it is recursively scanned to be sorted alphabetically. If the sublist to be sorted consists of an attribute–value pair where the value is atomic, then the sorting is not performed because doing so would alter the signature, making it not correspond to the original head constituent.

Observe that in the U-sign creation algorithm it is assumed that the f-structure's head constituent is the first element of the list that conforms the graph. Therefore

```
FUNCTION get-sign1(dg; sign);
   subsign ← nil;
   IF (dg type = atom) THEN
     sign ← dg;
   ELSE IF (dg type = top) THEN
           sign ← nil;
   ELSE FOR EACH subdg IN dg DO
           IF (subdg tail type = complex) THEN
             subsign ← get-sign2(subdg);
             IF null sign THEN
               sign ← (list subsign);
             ELSE sign ← append sign, (list subsign);
           ELSE subsign ← get-sign1((subdg tail); sign);
             IF null subsign THEN
               IF null sign THEN
                 sign ← (list (subdg head));
               ELSE sign ← append  sign, (list (subdg head));
             ELSE IF (subsign type = list) THEN
                     sign ← append sign, (list (subdg head)),
                                         (list subsign);
             ELSE sign ← append sign,
                                     (list append (list (subdg head)),
                                     (list subsign));
END
```

Figure 4. First complementary function to create a signature: get-sign1( )

```
FUNCTION get-sign2(dg);
   sign ← nil;
   IF null dg THEN
     nil;
   ELSE sign ← get-sign1(dg tail; nil);
     IF (sign type = list) AND (sign head type ≠ atom) THEN
       sign ← append (list (dg head)), sign;
     ELSE sign ← append (list (dg head)), (list sign);
END
```

Figure 5. Second complementary function to create a signature: get-sign2( )

```
FUNCTION sort-signature (signature);
   IF (signature length > 1) THEN
     signature ← sort signature alphabetically according to
                 upper level features;
   FOR EACH subsign IN signature DO
     IF (subsign length = 1) AND
         (subsign head type = atom) THEN
       subsign;
     ELSE IF (subsign length = 2) AND
               (subsign tail head type = atom) THEN
             subsign;
     ELSE subsign ← sort-signature(tail subsign);
END
```

Figure 6. Function to sort a unification signature: sort-signature( )

it is important to make sure that the builder of the grammar keeps this in mind at unification-based grammar creation time. Signature creation may seem to be an overload to the system's performance. However, since signatures are stored within a slot of the graph's data structure, the signatures for the entire unification-based grammar parsing system can be stored permanently in the grammar beforehand. The only signature to be built at run time is the signature that corresponds to the input sentence. The signature for a given input is rebuilt whenever a different f-structure for the same input has to be analysed, such as when ambiguities occur. This overload is small enough as to not affect the overall operation of the U-filter.

Unification filtering

The U-filtering process itself is straightforward. It consists of pattern-matching the two signatures that correspond to the graphs to be unified, ensuring that the content and context of features are compatible in both signatures. The U-filter always sends back to the parser a 'filter-decide parameter with a Boolean value. A 'filter-decide parameter with value true is returned to the parser if and only if incompatibility between the signatures was found. This means that, since the head constituent of the f-structures to be unified does not match in the contents of their atomic and complex nodes, regardless of the existent top nodes, any attempt to unify them would undoubtedly result in failure. Consequently, the unification that was about to be performed is simply skipped and the U-filter is applied to the next unification process, if any. On the other hand, a 'filter-decide parameter with value nil is returned to the parser if and only if the signatures are fully compatible. This value indicates that the U-filter cannot determine whether or not unification of the corresponding graphs succeeds. That is, no guarantee exists then that unification over the corresponding graphs will indeed succeed, but there is a high possibility of success.* As result, unification over the graphs is performed by the paser.

The importance of the U-filter lies in that it runs significantly faster than unification itself. Therefore each time the U-filter detects that the graphs do not unify a time saving is obtained; namely the time span that would be taken to actually try to unify the graphs. On the other hand, whenever the U-filter is unable to determine if unification would fail, a time overload is obtained. That is, the total unification time for the corresponding graphs would be the U-filtering time plus the unification time. This overload is inevitable whenever unification succeeds, and is, so to speak the 'price to be paid' for the benefit of being able to stop the majority of non-unifiable f-structures. Obviously the worst case occurs when the U-filter does not determine unification failure and unification actually fails because then two wasted time spans have to be added to parsing time, namely unsuccessful filtering and unsuccessful unification. Although U-filtering implies an extra consumption of time in parsing, existing unification schemes run significantly faster with the U-filter as a preprocess of them than without it.

It is important to note that U-filtering always takes place before unification. Furthermore, since the U-filter is independent of unification, it is possible to implement it as a process prior to any known unification scheme, significantly decreasing the total unification time in virtually any unification scheme. Note that the U-filter does not perform graph unification, it only performs pattern-matching of the signatures that correspond to the graphs to be unified.

Unification filter algorithm

The U-filter algorithm consists of a main function, unification-filter( ), two functions that perform the filtering of signatures, u-filter1( ) and u-filter2( ), and a function dedicated to finding common occurrences of elements in the signatures, get-next-match( ). The corresponding COMMON LISP program is shown in Appendix II.

The function unification-filter( ), listed in Figure 7, performs two simple actions. First, set up a *catch–throw* construct by assignment to a tag, in this case 'filter-decide, that immediately stops execution whenever a value is assigned to it in an invoked function, and return that value to unification-filter( ). Secondly, call the function that scans the U-sign by recursively taking one sublist of each U-sign at a time, namely u-filter1( ). Note that unification-filter( ) always returns a Boolean value. A returned value true means that the U-filter detected inconsistency, i.e. the attempt to unify

* Around 87 per cent in the implementation discussed in this paper.

```
FUNCTION unification-filter (signature1; signature2);
   result ← catch with tag 'filter-decide
             calling u-filter1(signature1; signature2);
   return (result);
END
```

Figure 7. Main function for U-filtering: unification-filter( )

the graphs would result in failure, so unification is unnecessary. On the other hand, a returned value nil means that the U-filter found no inconsistencies, i.e. cannot determine whether or not unification fails, so unification becomes necessary.

U-filter1( ), listed in Figure 8, first checks the head of the first sublist of the U-signs to be filtered. If the U-signs' heads are equal and contain no tails, and no tails in the U-signs exists, then return 'filter-decide with value nil, i.e. unification is needed. However, if the U-signs have tails, then recursively call u-filter1( ) with the U-signs' tails as parameters. If any of the head sublists have a tail then a catch–throw construct with the tag 'u-filter2-stop is created and the process is forwarded to u-filter2( ). If both U-signs have no tail, then return 'filter-decide with value nil and finish the u-filter1( ) process at the corresponding level of recursion. On the other hand, if

```
FUNCTION u-filter1(signature1; signature2);
   sign1h ← (signature1 head);
   sign2h ← (signature2 head);
   sign1t ← (signature1 tail);
   sign2t ← (signature2 tail);
   IF (sign1h head) = (sign2h head) THEN
     IF NOT ((null (sign1h tail)) OR (null (sign2h tail))) THEN
       result ← catch with tag 'u-filter2-stop
                 calling u-filter2((sign1h tail); (sign2h tail));
     IF (null sign1t) OR (null sign2t) THEN
       'filter-decide ← nil; throw with keyword 'filter-decide;
     ELSE u-filter1((sign1t); (sign2t));
   ELSE IF (null sign1t) AND (null sign2t) THEN
          'filter-decide ← nil; throw with keyword 'filter-decide;
        ELSE signature1; signature2 " get-next-match(signature1;
                                                     signature2);
          IF (null signature1) OR (null signature2) THEN
            'filter-decide ← nil;
                                throw with keyword 'filter-decide;
          ELSE u-filter1(signature1; signature2);
END
```

Figure 8. First complementary function for U-filtering: u-filter1( )

tails exist then call get-next-match( ) to obtain the next common sublist and, if any was found, recursively call u-filter1( ); otherwise return 'filter-decide with value nil and finish the process at the current level of recursion.

The actual decision upon when the U-filter succeeds, i.e. when unification is not needed, takes place within u-filter2( ), shown in Figure 9. First, if the U-subsigns are atomic and equal then return 'u-filter2-stop with the value true, which means that U-filtering of the subsigns succeeded, and finish the process at the current level of recursion. On the other hand, if the U-subsigns are different, return 'filter-decide with the value true, meaning that unification is not necessary, and finish the process at the corresponding level of recursion. If no third sublist exists in both U-signs, i.e. no further U-subsigns exist, then do the following. If the U-signs have no tail then return 'u-filter2-stop with the value true and finish the process at the current level of recursion. If tails exist then check if the U-signs contain only heads with equal content; if so return 'filter-decide with the value true, and terminate the process at that level of recursion; otherwise recursively call u-filter2( ) with the U-signs' tails as parameters. However, if further U-subsigns exist do the following. If the U-

```
FUNCTION u-filter2(signature1; signature2);
   sign1 ← (signature1 head);
   sign2 ← (signature2 head);
   IF (sign1 type = atom) OR (sign2 type = atom) THEN
     IF sign1 = sign2 THEN
       'u-filter2-stop ← true; throw with keyword 'u-filter2-stop;
     ELSE 'filter-decide ← true;
                                throw with keyword 'filter-decide;
   ELSE IF (null (sign1 third)) AND (null (sign2 third)) THEN
           IF (null (sign1 tail)) OR (null (sign2 tail)) THEN
             'u-filter2-stop ← true;
                                   throw with keyword 'u-filter2-stop;
           ELSE IF (sign1 head) = (sign2 head) AND
                     NOT (sign1 second) = (sign2 second) THEN
                   'filter-decide ← true;
                                     throw with keyword 'filter-decide;
                 ELSE u-filter2((sign1 tail); (sign2 tail));
   ELSE IF NOT (sign1 head) = (sign2 head) THEN
           'u-filter2-stop ← true;
                                throw with keyword 'u-filter2-stop;
         ELSE IF (sign1 head type = atom) THEN
                 u-filter1((sign1 tail); (sign2 tail));
         ELSE u-filter1(sign1; sign2);
           u-filter2((sign1 tail); (sign2 tail));
END
```

Figure 9. Second complementary function for U-filtering: u-filter2( )

subsigns' heads are different then return 'u-filter2-stop with the value true and finish recursion at that level. If the U-subsigns' heads are different but both are atomic then recursively call u-filter1( ) with the U-signs' tails as parameters. Otherwise recursively call both u-filter1( ) and u-filter2( ), in that order, with the U-signs and their tails as parameters, respectively, and finish the process at the current level of recursion.

Get-next-match( ) is the function devoted to finding common elements, i.e. sublists or atoms, in the U-signs being U-filtered; see Figure 10. This function's process consits of the recursive comparison of the sublists' heads until a match is found. Once this has been accomplished, the matching sublists are sent back to u-filter1( ). Here it is important to have the signatures alphabetically ordered.

AN EXAMPLE OF U-FILTERING

In order to get a clear idea of the U-filter's simplicity, an example of successful and unsuccessful filtering is illustrated in Table I. Successful U-filtering takes place between U-SIGN1 and U-SIGN2, and unsuccessful U-filtering takes place between U-SIGN1 and U-SIGN2◇. Note that U-SIGN2◇ consists of a modified version of U-SIGN2 and thus only some parts of the U-filtering are shown. The actions and results of U-filtering are indicated in the corresponding 'Action/result' and 'Action/result◇' columns.

U-filtering between U-SIGN1 and U-SIGN2 takes place as follows. The first sublist is scanned in u-filter1( ). Since both sublists are atomic and contain the same head feature, head (see steps 1 and 2 in Table I), the corresponding tails are sent to u-filter2( ) to be compared recursively. Once in u-filter2( ), each sublist is analysed by comparison of heads and tails, resulting in a successful match (steps 3 to 6). Note

```
FUNCTION get-next-match(sign1; sign2);
   lis1 ← (sign1 head);
   lis2 ← (sign2 head);
   aux1 ← nil;
   aux2 ← nil;
   IF (listl head) = (((sign2 tail) head) head) THEN
     return (sign1; (sign2 tail));
   ELSE IF (((sign1 tail) head) head) = (lis2 head) THEN
           return ((sign1 tail); sign2);
   ELSE IF nul lis1 OR null lis2 THEN
           return (sign1;sign2);
   ELSE aux1; aux2 ← get-next-match(sign1; (sign2 tail));
     IF null aux2 THEN
       aux1; aux2 ← get-next-match((sign1 tail); sign2);
     return (aux1; aux2);
END
```

Figure 10. Function to get the next common elements in a U-sign: get-next-match( )

Table I. A simple example of U-filtering

Step	U-SIGN1	U-SIGN2	U-SIGN2◇	Action/result	Action/result◇
0	((head (cform senf) (pos v)) (sem (agen) (reln hello)) (subcat end))	((head (cform senf)) (prag (hearer) (speaker) (sem (agen) (reln hello)))	((head (cform)) (sem (reln goodbye)))		
1	(head (cform senf) (pos v))	(head (cform senf))	(head (cform))	get sublist	get sublist
2	head	head	head	match	match
3	((cform senf) (pos v))	(cform senf)	—	get sublist	
4	(cform senf)	(cform senf)	—	get sublist	
5	cform	cform	cform	match	match
6	senf	senf	*nil*	match	skip
7	(pos v)	*nil*	—	skip	
8	(sem (agen) (reln hello))	(prag (hearer) (speaker))	—	skip	
9	(sem (agen) (reln hello))	(sem (agen) (reln hello))	(sem (reln goodbye))	get sublist	get sublist
10	sem	sem	sem	match	match
11	((agen) (reln hello))	((agen) (reln hello))	—	get sublist	
12	(agen)	(agen)	—	match	
13	(reln hello)	(reln hello)	(reln goodbye)	get sublist	getsublist
14	reln	reln	reln	match	match
15	hello	hello	goodbye	match	U-filter=t
16	(subcat end)	*nil*	—	skip	
17	*nil*	*nil*	—	U-filter=nil	

that since the remaining tail, (pos v), does not exist in U-SIGN it is skipped in get-next-match( ) (step 7). The rest of the list is then analysed in u-filter1( ). Since the head sublists are different (step 8), a get-next-match( ) is performed and U-filtering over the extracted sublists takes place. Again, pattern-matching of sublists takes place successfully (steps 9 to 15). Finally, the get-next-match( ) for (subcat end) reaches the end of the lists (step 16), and since no inconsistency was found U-filter returns to the parser a 'filter-decide parameter with value nil to indicate that unification is needed (step 17).

In the case of U-filtering U-SIGN1 against U-SIGN2◇, the pattern-matching of the two signatures keeps going smoothly until step 14. At step 15, however, the value for the slot reln is different in the two U-signs, namely hello and goodbye. Such a discrepancy is detected by the U-filter, so that a 'filter-decide value true is returned to the parser to indicate that unification is unnecessary.

EXPERIMENTAL RESULTS

To test the U-filter an HPSG-based Japanese grammar (JPSG[3]) that covers diverse, important linguistic phenomena in conversational Japanese was used. For example, case adjunction, adjuncts, slash categories, co-ordination, control, WH-constructs, interrogatives, pragmatics (politeness, hearer relations, speaker, etc.),[20] and zero-pronouns. The grammar graphs contain 2324 nodes converted from the path equations. This grammar is similar to those developed at ATR and the Carnegie Mellon University.

The experiment used consisted of the parsing of 16 sentences taken from a telephone conversation dialogue within the domain of conference registration. These sentences vary from very short sentences, such as *moshimoshi* 'hello', to very long sentences, such as *kochirakarasochiranitourokuyoushiwoshikyuuniookuriitashimasu* '[we] will send {polite} [you] a registration form from here {polite} to there {polite}'. The sizes of the U-signs, in numbers of list elements, range between 4 and 50 elements; however, these numbers may significantly increase as the grammar used covers even wider linguistic phenomena.

Results of the experiments are shown in Table II. Benchmarking was performed against the QDS scheme because it is the fastest unification scheme developed so far, to our knowledge. Non-destructive graph unification and quasi-destructive graph unification are not directly compared here because proofs that there is a speed-up gain of QDS over these two schemes exist elsewhere in the literature.[12–14]

In Table II, columns whose labels start with QDS show data corresponding to the use of the QDS scheme alone; columns whose label starts with a UF show data corresponding to the U-filter alone; columns whose label starts with a QDS-UF contain data corresponding to QDS with UF included. All the UF timings consist of the unification signature creation time for the input sentence, including unification signature sorting, and the U-filtering of signatures. 'QDS unif. num.' shows the total number of top level unifications QDS performed during the parse of each sentence. Note that the number of unifications varies from just a few for simple sentences, to several hundreds for long, complex sentences. 'UF unif. num.' displays the number of QDS unifications that had to be performed when the U-filter was used; note that in total less than half the number of unifications with QDS only

Table II. Comparison of the QDS scheme performance with and without the U-filter

Sentence number	QDS unif. num.	UF unif. num.	QDS success rate	UF pass rate	UF effectiveness	QDS time (ms)	QDS-UF time (ms)	UF time (ms)	QDS-UF/ QDS ratio
1	6	3	0·50	0·50	1·00	39	38	9	0·97
2	101	39	0·34	0·39	0·87	888	595	172	0·67
3	18	7	0·22	0·38	0·58	97	77	21	0·79
4	71	44	0·55	0·62	0·89	768	501	126	0·65
5	305	156	0·37	0·51	0·73	2148	1779	470	0·82
6	59	24	0·27	0·41	0·66	329	239	74	0·72
7	6	3	0·50	0·50	1·00	39	38	9	0·97
8	81	43	0·51	0·53	0·96	948	661	175	0·69
9	480	226	0·37	0·47	0·79	4220	2460	746	0·58
10	555	283	0·41	0·51	0·80	4584	3532	763	0·77
11	109	57	0·45	0·52	0·87	948	723	201	0·76
12	428	175	0·33	0·41	0·80	4036	2199	660	0·54
13	559	268	0·39	0·48	0·81	5963	4747	1251	0·79
14	52	21	0·38	0·41	0·93	437	322	86	0·73
15	77	45	0·55	0·58	0·95	687	597	148	0·86
16	77	44	0·55	0·57	0·96	667	603	141	0·90
Average			0·42	0·48	0·87				
Total	2984	1438				26,798	19,111	5052	
Percentage of Total	100	48·2				100	71·3		

were needed. Thus, empirically, the gain in speed by using the U-filter increases as the percentage of unification failures in parses increases. 'QDS success rate' shows the ratio of successful top level unifications, which has an average of 0·42, by QDS only. This average of success is rather high, considering that, in general, natural language parses reach unification success rates of around 0·15. 'UF pass rate' shows the fraction of graphs that the U-filter could not stop, and which therefore had to be unified. The small difference between these two ratios indicates that only a few of the graphs that pass the U-filter fail unification. 'UF effectiveness' shows the ratio of successful unifications with respect to U-filter, which has an average of 0·87. This high average means that only a small number of the graphs sent to unification result in failure. Actually, in the experiment, the number of top level unifications QDS performed varied from between 6 and over 550 without the U-filter, and between 6 and over 280 with the U-filter.

Also in Table II, 'QDS time' shows the total unification time for each sentence by QDS only. 'QDS-UF time' shows the time it took to unify the corresponding sentences when QDS was preceded by the U-filter. 'UF time' shows the amount of time consumed by the U-filter, including U-sign creation for the input sentence.* QDS-UF was executed in around 71 per cent of the execution time for QDS alone. The last column in Table II, 'QDS-UF/QDS ratio,' shows the timing ratio between 'QDS time' and 'QDS-UF time' for each sentence, which ranged between 0·54 in the worst case and up to 0·97 in the best case. Provided that the average success rate in the experiment was around 42 per cent, it is foreseeable that if the failure rate and the number of unifications increase significantly, with small change in the U-filter effectiveness, then the resulting speed-up would be higher.

A time comparison chart of QDS with and without the U-filter for the 16-sentence telephone conversation is displayed in Figure 11 (note the logarithmic scale on the vertical axis). 'QDS' depicts the unification time by QDS only, 'QDS-UF' depicts the unification time it takes to unify the conversation when the U-filter is used, and 'UF' depicts the U-filtering time only, including signature creation and sorting. The positions of the symbols for each timing line (diamonds, stars and thetas) represent the location for each one of the 16 sentences analysed, displayed according to the number of unifications performed by QDS only. Two sentences required only six unification operations each by QDS only, namely *moshimoshi*, 'hello' and *iie*, 'no': whereas one sentence required 559 unification operations, namely *wakaranaitengago-zaimashitarawa takushidoumoniitsudemookikikudasai*, 'If there is {polite} anything [you] do not understand, please {polite} ask me {polite} anytime'. As the number of unifications performed increases (horizontal axis) execution time increases (vertical axis) linearly. Note also that as the number of unifications increases, the gap between QDS and QDS-UF increases. This behaviour takes place simply because, typically, as the number of unifications to be performed increases, the number of unification failures also increases; and since the U-filter stops the majority of unification failures in a short time span, a higher speed-up is obtained. UF alone is significantly faster than QDS and QDS-UF, showing that all graphs that do not pass U-filtering are discarded faster than through traditional unification failure. Within UF, creation and sorting of the U-sign for the input sentences in the experiment consumed between 3 and 12 ms, with an average of around 5 ms.

* Includes also signature sorting.

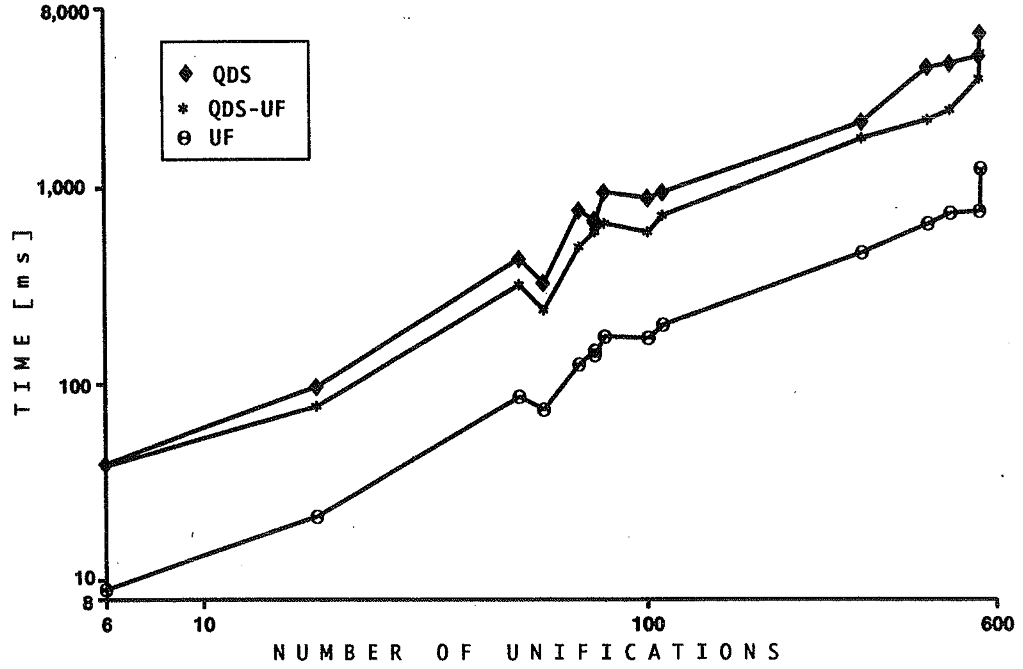

Figure 11. Time comparison chart between QDS and QDS-UF, and UF timing

CONCLUSIONS

Unification is, by far, the most costly operation in unification-based grammar parsing. A great disadvantage of unification is that the vast majority of unifications performed result in failure; a waste in processing time. In this paper, an efficient method for the treatment of unification failures has been proposed, namely the U-filter, which determines if two graphs do not unify in significantly less time than it would take actual unification methods to do so. Structures that pass U-filtering must be unified in the traditional way, and may or may not unify. Therefore, the U-filter must offer an effectiveness and processing speed that, combined with unification methods, reduces the total unification time significantly. The U-filter scheme consists of adding to the data structure of a graph, or f-structure, a slot containing a unification signature for that node; and using such a signature to determine unification failure even before unification itself takes place.

Since the U-filter is unification independent it can be implemented under any known unification scheme. In addition, its simplificity makes it easy to implement in diverse environments as well as easily adaptable to variations. The amount of extra storage needed for U-signs is very small and the overload for creation and sorting of the input sentence is immediately compensated for by the U-filtering timing itself.

The slowest part of the U-filter is U-sign sorting, so a schema that tackles this problem is needed. Although no case in which the U-filter ran slower than unification alone existed in the experiment, it is possible that, for parses with a low percentage of unification failures, less than 40 per cent, the U-filter may sometimes run slightly slower than unification alone. The U-filter proposed here works efficiently for relatively small signatures, of around 50 elements; however, an empirical observation is that efficiency may be affected for large signatures. A vector array space based unification filter may be a solution for this problem and is being considered for future development. A vector space array consists of a vector data structure, a matrix, in which the elements that conform to the signature are stored. Since the access of elements in a vector is constant, then a unification filter based on this approach should perform filtering in less time than using lists. The development of more efficient representations of signatures is desirable. A routine that puts the head constituent on top of the corresponding grammar rules in unification-based grammar parsing systems is also desirable. This would avoid grammar designers having the cumbersome task of doing so.

APPENDIX I: COMMON LISP CODE FOR U-SIGN CREATION

```
;;; CREATE-SIGN
(defun create-sign (dg)
  ; dg <- the graph to be U-Filtered
  ; auxdg <- the arc-list of dg.
  ; sign <- the resulting list from dg under the head of each sublist of auxdg.
  ; signature <- recursively the resulting sublist out of each sublistof auxdg, and the final list
  ;               that corresponds to the original dg.
  (declare (type dgnode dg))
  (let ((auxdg (DGNODE-arc-list dg)) (sign nil) (signature nil))
    (declare (type dgnode auxdg))
    (cond ((null auxdg) nil)
          (t (dolist (subdg auxdg)
               (setq sign (get-sign1 (cdr subdg)))
               (cond ((and (not (null sign)) (listp sign) (atom (car sign)))
                       (setq sign (list (append (list (car subdg)) (list sign)))))
                      ((listp sign)
                       (setq sign (list (append (list (car subdg)) sign))))
                      (t (setq sign (list (append (list (car subdg)) (list sign))))))
               (setq signature (append signature sign)))
             signature ))))
;;; GET-SIGN1
(defun get-sign1 (dg &optional sign)
  ; dg <- the graph to be U-Filtered
  ; sign <- the partial result in the generation of sign.
  ; auxdg <- the arc-list of dg, i.e., A list containing the structure under the node dg.
  ; subsign <- the partial result in the generation of sign.
  (declare (type dgnode dg) (type list sign))
  (let ((auxdg (DGNODE-arc-list dg)) (subsign nil))
    (declare (type dgnode auxdg))
    (cond ((ATOMICNODE-p dg)
            (setq sign auxdg))
          ((LEAFNODE-p dg)
            (setq sign nil))
          (t (dolist (subdg auxdg)
               (declare (type dgnode subdg))
               (let ((subdg-param (cdr subdg)))
                 (declare (type dgnode subdg-param))
                 (cond ((COMPLEXNODE-p subdg-param)
                         (setq subsign (get-sign2 subdg))
                         (cond ((null sign)
                                 (setq sign (list subsign)))
                                ((listp (car sign))
                                 (setq sign (append sign (list subsign))))
                                (t (setq sign (append (list sign) (list subsign))))))
                        (t (setq subsign (get-sign1 (cdr subdg) sign))
                           (cond ((null subsign)
                                   (cond ((null sign)
                                           (setq sign (list (car subdg))))
                                          ((listp (car sign))
                                           (setq sign (append sign (list (list (car subdg))))))
                                          (t (setq sign (append (list sign) (list (list (car subdg)))))))
                                 ((listp subsign)
                                      (setq sign (append sign (list (car subdg)) (list subsign))))
                                     (t (setq sign (append sign (list (append (list (car subdg))
                                                                              (list subsign)))))
                                       ))))))))) sign)


    ;;; GET-SIGN2
    (defun get-sign2 (dg)
      ; dg <- the graph to be U-Filtered
      ; sign <- the partial result in the generation of sign.
      (declare (type dgnode dg))
      (let ((sign nil))
        (cond ((null dg)
               nil)
              (t (setq sign (get-sign1 (cdr dg)))
                 (cond ((and (listp sign) (atom (car sign)))
                         (setq sign (append (list (car dg)) (list sign))))
                       ((listp sign)
                         (setq sign (append (list (car dg)) sign)))
                       (t (setq sign (append (list (car dg)) (list sign)))))
                 sign ))))


    ;;; SORT SIGNATURE
    (defun sort-signature (signature)
      (cond ((> (length signature) 1)
             (setf signature (sort signature #'string-lessp :key #'car))
             (dolist (subsign signature)
               (cond ((and (= (length subsign) 1) (atom (car subsign)))
                      subsign)
                     ((and (= (length subsign) 2) (atom (cadr subsign)))
                      subsign)
                     (t (setf (cdr subsign) (sort-signature (cdr subsign)))))))
            (t signature))
      signature)
```

APPENDIX II: COMMON LISP CODE FOR THE U-FILTER

```
;;; UNIFICATION FILTER
(defun unification-filter (signature1 signature2)
  (catch 'FILTER-PASS (u-filter1 signature1 signature2)))


;;; U-FILTER1
(defun u-filter1 (signature1 signature2)
  (let ((signhead1 (first signature1)) (signhead2 (first signature2))
        (signtail1 (rest signature1)) (signtail2 (rest signature2)))
    (cond ((equal (first signhead1) (first signhead2))
           (cond ((not (or (null (rest signhead1)) (null (rest signhead2))))
                  (catch 'U-FILTER2-STOP (u-filter2 (rest signhead1) (rest signhead2)))))
           (cond ((or (null signtail1) (null signtail2))
                  (throw 'FILTER-DECIDE nil))                                             ;;; unification needed
                 (t (u-filter1 signtail1 signtail2))))
          (t (cond ((and (null signtail1) (null signtail2))
                    (throw 'FILTER-DECIDE nil))                                           ;;; unification needed
                   (t (multiple-value-setq (signature1 signature2) (get-next-match signature1 signature2))
                      (cond ((or (null signature1) (null signature2))
                             (throw 'FILTER-DECIDE nil))                                   ;;; unification needed
                            (t (u-filter1 signature1 signature2)))))))))
```

```
(defun u-filter2 (signature1 signature2)
  (let ((signhead1 (first signature1)) (signhead2 (first signature2)))
    (cond ((or (atom signhead1) (atom signhead2))
           (cond ((equal signhead1 signhead2)
                  (throw 'U-FILTER2-STOP t))
                 (t (throw 'FILTER-DECIDE t))))                  ;;; unification not needed
          ((and (null (third signhead1)) (null (third signhead2)))
           (cond ((or (null (rest signhead1)) (null (rest signhead2)))
                  (throw 'U-FILTER2-STOP t))
                 ((and (equal (first signhead1) (first signhead2))
                       (not (equal (second signhead1) (second signhead2))))
                  (throw 'FILTER-DECIDE t))                      ;;; unification not needed
                 (t (u-filter2 (rest signhead1) (rest signhead2)))))
          (t (cond ((not (equal (first signhead1) (first signhead2)))
                    (throw 'U-FILTER2-STOP t))
                   ((atom (first signhead1))
                    (u-filter1 (rest signhead1) (rest signhead2)))
                   (t (u-filter1 signhead1 signhead2)
                      (u-filter2 (rest signature1) (rest signature2))))))))


;;; GET-NEXT-MATCH
(defun get-next-match (sign1 sign2)
  (let ((lis1 (first sign1)) (lis2 (first sign2)) (aux1 nil) (aux2 nil))
    (cond ((equal (first lis1) (first (first (rest sign2))))
           (values sign1 (rest sign2)))
          ((equal (first (first (rest sign1))) (first lis2))
           (values (rest sign1) sign2))
          ((or (null lis1) (null lis2))
           (values sign1 sign2))
          (t (multiple-value-setq (aux1 aux2) (get-next-match sign1 (rest sign2)))
             (cond ((null aux2)
                    (multiple-value-setq (aux1 aux2) (get-next-match (rest sign1) sign2)))
                   (t t))
             (values aux1 aux2)))))
```

REFERENCES

1. J. Earley, 'An efficient context-free parsing algorithm', *Commun. ACM,* **13**(2), 94–102 (1970).
2. M. Tomita, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*, Kluwer Academic, 1986.
3. T. Gunji, *Japanese Phrase Structure Grammar*, D. Reidel, Dordrecht, 1987.
4. G. Gazdar, G. Pullum and I. Sag, *Generalized Phrase Structure Grammar*, Harvard University Press, 1985.
5. C. Pollard and I. Sag, *Information-based Syntax and Semantics, Vol. 1,* CSLI, 1987.
6. K. Kogure, 'Strategic lazy incremental copy graph unification', *Proc. COLING-90*, 1990.
7. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
8. F. Pereira, 'A structure sharing representation for unification-based grammar formalisms', *Proc. ACL-85*, 1985, pp. 137–144.
9. L. Karttunen, 'D-PATR: a development environment for unification-based grammars', *Proc. COLING-86*, 1986.
10. D. Wroblewski, 'Nondestructive graph unification', *Proc. AAAI-87*, 1987, pp. 582–587.
11. M. C. Emele, 'Unification with lazy non-redundant copying', *Proc. ACL-91*, 1991, pp. 323–330.
12. H. Tomabechi, 'Quasi-destructive graph unification', *Proc. ACL-91*, 1991, pp. 315–322.
13. H. Tomabechi, 'Efficient unification for natural language', *Ph.D. Thesis*, Carnegie Mellon University, Program in Computational Linguistics, 1993.
14. H. Tomabechi, 'Quasi-destructive graph unification with structure sharing', *Pro. COLING-92*, 1992.
15. P. A. Larson, 'A method for speeding up text retrieval', *Proc ACM SIGMOD-83*, 1983.
16. S. Christodoulakis and C. Faloutsos, 'Design considerations for a message file server', *IEEE Trans. Softw. Eng.*, **SE-10**(2), 201–210 (1984).
17. M. C. Harrison, 'Implementation of the substring test by hashing', *Comm. ACM,* **14**(12), 777–779 (1971).
18. A. V. Aho and M. J. Corasick, 'Efficient string matching: an aid to bibliographic research', *Comm. ACM* **18**(6), 333–340 (1975).

19. J. W. Hunt and T. G. Szymanski, 'A fast algorithm for computing longest common subsequences', *Comm. ACM,* **20**(5), 350–353 (1977).
20. K. Yoshimoto and K. Logure, 'Japanese sentence analysis by means of phrase structure grammar', *ATR Technical Report, TR-1-0049*, 1989.