Revising Rules to Capture Requirements Traceability Relations: A Machine Learning Approach

G. Spanoudakis, A. d'Avila Garcez, A. Zisman

Department of Computing
City University
Northampton Square,
London EC1V 0HB
United Kingdom
{gespan | aag | a.zisman}@soi.city.ac.uk

Abstract

In this paper we present a machine learning approach for generating requirements traceability relations. This approach is based on a new learning algorithm that produces traceability rules which are able to capture traceability relations between requirement statements specified in natural language and object models. The creation of these traceability rules is informed by examples of traceability relations which are provided by the user and is based on a generalisation of other existing traceability rules.

Keywords: requirements traceability, machine learning.

1. Introduction

Requirements traceability – that is the ability to relate requirements specifications with other artefacts created in the development life-cycle of a software system – has widely and long been recognised as a significant factor for efficient software project management and software systems quality [13][14]. Research into requirements traceability has been concerned with the study and definition of different types of traceability relations [2][4][12] (e.g. satisfiability and dependency relations) and the provision of support for generating and maintaining them in requirements engineering tools [5][9][18].

However, as empirical studies of the traceability needs and practices of industrial organisations have indicated, this support is not always satisfactory [14] as most of the existing approaches and tools assume that traceability relations are created and maintained manually [13][14][10]. This makes the establishment of traceability an error-prone and time consuming task and, as a consequence, traceability is rarely established.

To address this problem, Zisman et al [20] have developed a system that automatically generates and maintains traceability relations of different types between textual requirement statements and object models (expressed in UML [8]) based on *traceability rules*. These rules specify ways of matching syntactically related words in the requirements statements with related elements in an object model (e.g. classes, attributes, operations), and create traceability relations of different types between these artefacts when a match is found. The syntactic relations used in the rules are defined in terms of patterns of words which have specific grammatical roles in a piece of text (e.g. noun, verb, adjective etc.). These grammatical roles are identified using probabilistic grammatical tagging techniques [3].

Initial experiments with this traceability system have indicated that, although it is capable of generating traceability relations at reasonable *precision* and *recall* rates, there is scope for improving its performance with respect to both these performance criteria [16][17]. In particular with respect to recall (i.e., the ratio of traceability relations between artefacts that the system can capture), we have identified the need to provide support for generating new traceability rules to capture relations that existing rules fail to generate. This support becomes particularly important when trying to deploy the traceability system in contexts where the involved textual requirement artefacts are specified using varying specification styles and practices.

In this paper, we present an approach that addresses this problem. Our approach is based on a new machine learning algorithm which, given examples of undetected traceability relations indicated by the user, tries to generate new traceability rules capable of capturing these relations and maintaining them once correctly generated. The new traceability rules are generated by transforming existing rules in order to make them match with the given examples of traceability relations.

The rest of this paper is structured as follows. In Section 2, we present an overview of the rule-based requirements traceability system and give examples of using it to generate traceability relations. In Section 3, we

overview our machine learning approach for generating new traceability rules. In Section 4, we present the learning algorithm underpinning this approach. In Section 5, we give examples of generating traceability rules. In Section 6, we overview related work, and in Section 7 we summarise our approach and plans for future work on it.

2. Rule-based traceability

The rules which are deployed by the traceability system in [20] express heuristics that predict how words, which have specific grammatical roles in a textual requirement statement and appear in specific syntactic patterns, may have been modelled in an object model. The grammatical role of a word in a textual requirement statement is signified by a *part-of-speech tag* (POS-tag) that is generated by a general-purpose probabilistic grammatical tagger called CLAWS [3].

Figure 1 shows an example of a tagged requirement statement extracted from the specification of a university course management system (the POS-tags are highlighted in the figure). According to the statement, the system should allow specific types of users to record student coursework marks. The word "coursework" in this statement has been identified as a singular common noun as signified by the tag NN1 that is assigned to it. The word "record" in the same statement has been identified as a verb in the infinitive form as signified by the tag VVI that is assigned to it.

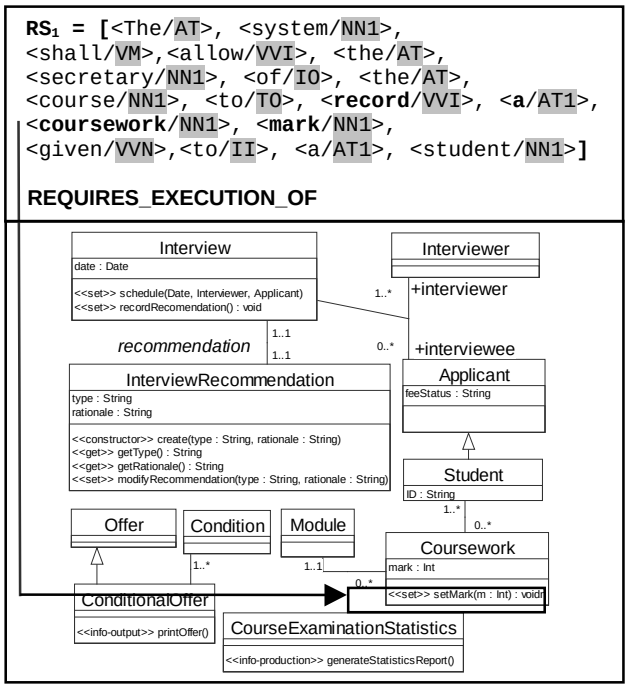

Figure 1. Example of a traceability relation

The general syntactic form of a *requirement–to–object model* (RTOM) traceability rule is:

RTOM_RULE RuleId
EXISTS
SEQUENCE(S_1, S_2, …, S_n) **in** α;
$<e_1/t^O_1>$, …, $<e_m/t^O_m>$ **in** OM_y
SUCH THAT
$p_1(f_1(\bullet), f_{L+1}(\bullet)) \circ \ldots \circ p_L(f_L(\bullet), f_{L+L}(\bullet))$
ACTION GENERATE ($r_w(\alpha.\sigma,e_j)$)
RTOM_RULE_END

In this syntactic form,

- *SEQUENCE(S_1, S_2, …, S_n) in α* specifies the syntactic pattern that the rule expects to find in a requirement statement *a*. This pattern is defined in terms of the syntactic items S_i (i=1,…,n) which may have one of the following two forms:

  $S_i = <v_i/T_i>:q$, or S_i = SEQUENCE(S_k, …, S_w): q
  where
  – v_i is a variable that can be matched with a word in α,
  – T_i is a set of possible tags that the words which can be matched with v_i should have, and
  – q is a qualifier that denotes whether a single (if q = "1") or multiple occurences (if q = "*") of the item S_i are expected;
- $e_1,\ldots,e_m$ are variables that take as values elements of an object model OM_y which are of types $t^O_1,\ldots,t^O_m$, respectively;
- $f_u(\bullet)$ (u=1,…, 2L) are functions and $p_1,\ldots, p_L$ are predicates which are used to specify conditions which must be satisfied by the words matched with the variables v_i and the object model elements matched with the variables e_i for them to be linked with a traceability relation (examples of these functions and predicates are shown in Figure 2);
- ° is the logical "and" or "or" operator; and
- $r_w(\alpha.\sigma,e_j)$ is a traceability relation of type r_w that can be created by the rule (σ is a sequence of words in *a* that matches with *SEQUENCE(S_1, S_2, …, S_n)*)

An example of a traceability rule is given below:

```
RTOM_RULE Rule-1
EXISTS
SEQUENCE(<x1/{VVI}>:1, <x2/{AT1}>:1,
<x3/{NN1}>:1, <x4/{NN1}>:*, <x5/{NN1}>:1) in A1;
<x6/CLASS>, <x7/OPERATION> in OM
SUCH THAT
OPERATION_OF(<x7>,<x6>) and
MEMBER_OF(<x1>,SYNONYMS(STEREOTYPE(<x7>)) and
CONTAINS(NAME(<x7>),<x5>) and
CONTAINS(NAME(<x6>),<x3>) and
(CONTAINS(NAME(<x6>),<x4>) or
CONTAINS(NAME(<x7>),<x4>))
ACTION GENERATE REQUIRES_EXECUTION_OF(A1.σ,<x7>)
RTOM_RULE_END
```

Rule-1 can establish a REQUIRES_EXECUTION_OF relation between a sequence of words in a requirement statement and an operation in an object model. A relation of this type is created if the operationalisation of what is described by the word sequence (typically an event) requires the execution of the operation [17].

EXAMPLES OF RTOM RULE PREDICATES

ATTRIBUTE_OF(Id_1,Id_2): This predicate becomes true if Id_1 is an attribute is defined in or inherited by the class Id_2.

ASSOCIATION_OF(Id_1,Id_2): This predicate becomes true if Id_1 is an association defined in or inherited by the class Id_2.

OPERATION_OF(Id_1,Id_2): This predicate becomes true if Id_1 is an operation defined in or inherited by the class Id_2.

CONTAINS($String_1$, $String_2$): This predicate becomes true if $String_2$ is a sub-string of $String_1$.

EQUAL_TO($String_1$, $String_2$): This predicate becomes true if $String_2$ is equal to $String_1$.

MEMBER_OF($String_1$, Set of <String>): This predicate becomes true if $String_1$ is a member of the set of strings Set of <String>.

EXAMPLES OF RTOM RULE FUNCTIONS

Name(Id_1): String – This function takes as input a UML model element identifier and returns the name of this model element.

Synonym($String_1$): Set of <String> – This function takes as input a string that is the name of a UML operation stereotype and returns the list of verbs that may be used in a piece of text to signify the primary function of the operations that have been classified under this stereotype.

Stereotype(Id_1): Set of <String> – This function takes as input the identifier of a UML object model element and if this identifier identifies an operation in the model, it returns a set of strings which are the names of the stereotypes of the this operation.

Figure 2. Predicates and functions used in RTOM rules

To generate a REQUIRES_EXECUTION_OF relation, *Rule-1* attempts to match a verb phrase that consists of a verb in the infinitive form (<x1/{VVI}>:1)) followed by an article (<x2/{AT1}>:1) and a noun (<x5/{NN1}:1>), which is qualified by one (<x3/{NN1}>:1) or more other nouns (<x4/{NN1}>:*), with an operation in an object model.

The matching succeeds if: (i) the verb is a member of the set of synonyms which are associated with the stereotype of the operation (see condition MEMBER_OF(<x1>, SYNONYMS(STEREOTYPE(<x7>))), (ii) the noun that is the object of the verb-phrase (x5) is contained in the name of the operation (see condition CONTAINS(NAME(<x7>), <x5>)), (iii) the first noun that qualifies the object of the verb-phrase (x3) is contained in the name of the class that defines the operation (see condition CONTAINS(NAME(<x6>), <x3>)**),** and (iv) the additional nouns which qualify the object of the verb-phrase (if any) are contained in the name of either the class that defines the operation or the operation itself (see condition (CONTAINS(NAME(<x6>),<x4>) or CONTAINS(NAME(<x7>),<x4>)).

The satisfaction of the first of these conditions indicates that the action denoted by the verb is compliant with the functional role of the operation which is indicated by the stereotype of it. This is because in UML [8], a stereotype is used to signify the main functional role of an operation (for example, operations, which set or modify the state of class instances are stereotyped as <<set>> operations). The satisfaction of the other three conditions increases the possibility of establishing a matching with an operation that is applied to the feature signified by the object of the verb-phrase.

Rule-1 succeeds in the case of the sequence of words <record /VVI>, <a /AT1>, <coursework /NN1>, <mark /NN1> which appear in the requirement statement RS_1 and the operation setMark(m: Int):void of the class Coursework in Figure 1. This is because: (a) the word variables of *Rule-1* are matched with the words in the above sequence as follows: record[x1], a[x2], coursework[x3], mark[x5], and (b) assuming that the verb "record" in RS_1 belongs to the set of synonyms of the stereotype of the operation setMark(m: Int):void (i.e., <<save>>) the SUCH THAT conditions of *Rule-1* are satisfied. The satisfaction of *Rule-1* in this case creates a REQUIRES_EXECUTION_OF relation between the above sequence of words in RS_1 and operation which is shown in Figure 1.

3. Generation of new RTOM rules

The machine learning approach that we describe in this paper supports the generation of new RTOM rules to capture traceability relations which are indicated by the users of the system but existing rules have failed to generate. New traceability rules can be created as generalisations of existing rules using a new machine learning algorithm that we have developed for this purpose (i.e., the algorithm GenerateRules in Figure 3). This algorithm gets as input a traceability relation of the form $r_w(\alpha.*,e_j)$ that has been identified by the user (we use $a.*$ in $r_w(\alpha.*,e_j)$ as the user is not expected to identify the exact sequence of words within the requirement statement a that gives rise to the relation) and tries to generate variants of existing traceability rules that could capture this relation by generalising their syntactic patterns.

To do so, GenerateRules firstly identifies all the rules that can generate traceability relations of the same type as the relation given by the user. Then, it generates transformations of the syntactic patterns of these rules that would make them match with any sequence of tagged words within the requirement statement α of the given relation (see first while-loop of the algorithm in Figure 3). These transformations are generated by the algorithm FindTrs which is discussed Section 4.

```
Algorithm    GenerateRules(TR, NR)
Input Variables:
TR;  // A traceability relation r_w(α.*,e_j)
     //not generated by existing rules
Output Variables:
NR;  // Ordered list of new rules that can capture TR
Let RULES be the set of rules that can generate a relation of the
   same type as TR;
TRULES := {};
For all R in RULES Do
   TRS_R := {}; SQ := R.SEQUENCE(S_1,S_2, …, S_n);
   Let SN be a linked list with the tagged terms in the
     artefact α;
   Let T be the first tagged term in SN; MaxGen_R := 0;
   While (T != NIL) Do
     T_init := T; Let V be the first syntactic item in SQ;
     FindTrs(R, SQ, SN, V, T, VM, Found_Tr, CS, Trans_R);
     // Trans_R is the set of transformations that
     // could be applied to R to match SN
     If Found_Tr = True then
       For all τ in Trans_R Do
         Generate a new rule R' by replacing the syntactic pattern of
              R by τ;
         //check satisfiability of new rule R'
         If R' is satisfied by SN and e_i then
           TRULES := TRULES ∪ (R, R');
         end if
       end for
       T := T.next;
     end if
   end while
end for
// order the new rules in descending generality order
NR := empty;
For all (R, R') in TRULES Do
   Gen_R' = gen (R');
   // gen(R') is the number of relations captured by R plus 1
   If NR is empty then insert R' into NR;
   else
     Let CR be the first rule in NR;
     While Gen_R' < gen(CR) and CR != NIL  Do
       CR := rule next to CR in NR;
     end while
     insert R' before CR in NR;
   end if
end for
```

Figure 3. Traceability rule generation algorithm

From these potential transformations of the syntactic conditions of the rules, `GenerateRules` considers only the ones that transform the original rules into forms which are satisfied by the artefacts α and e_j of the given traceability relation and, as a consequence, can generate this relation. To establish this, `GenerateRules` applies each transformation and then checks if the conditions in the SUCH THAT part of the transformed traceability rule are satisfied by α and e_j. The new rules are guaranteed to be able to capture the traceability relations captured by the rules they are created from and the new relation that these rules failed to generate.

Finally, `GenerateRules` ranks the rules that it creates in descending order of their generality. The generality of a new rule R' is measured by the number of traceability relations that the rule that R' was created from had captured plus one (for the new relation). The assumption underpinning this approach is that the rules that have captured more traceability relations are the ones with the wider applicability to the already seen relations and, therefore, they are more likely to capture relations which have not be seen yet (as in *reinforcement learning* [6]).

4. Identification of rule transformations

The identification of alternative transformations of existing traceability rules is carried out by the algorithm `FindTrs` that is specified in Figure 4. This algorithm generates all the possible transformations of the syntactic pattern *SEQUENCE(S_1,S_2, …, S_n)* of a rule R that can make it match with any sub-sequence of tagged words in a requirement statement.

These transformations are generated as generalisations of the syntactic pattern of R which can take one of the following three forms:

(i) the introduction of a new POS-tag in the set of possible tags T_i of the word variable v_i of a syntactic item S_i (S_i = <v_i/T_i>:q) in SEQUENCE(S_1,S_2,…,S_n);

(ii) the modification of an existing compulsory word variable into a non compulsory one; and

(iii) the introduction of a new non compulsory word variable in SEQUENCE(S_1,S_2,…,S_n).

As shown in Figure 4, these transformations are attempted in the order in which they appear above. The conditions under which each of them may be carried out are discussed in the following.

4.1 Introduction of a new POS-tag in the set of tags of an existing word variable

This form of generalisation may be allowed if `FindTrs` encounters a word in a requirement statement whose tag WT is not a member of the set of possible tags (i.e., TAGS) of the current word variable in the syntactic pattern of a rule. The new tag is inserted in TAGS only if it can be substituted for at least one of the tags which are already in it. `FindTrs` checks if this is the case by consulting the relation *can_substitute*.

can_substitute is binary symmetric relation that we have defined over the set of tags of CLAWS (see [3]) to represent possible substitutions between these tags in a piece of text. For example, *can_substitute(NN1,NN2)* in the extension of this relation means that a singular common noun (NN1) can be found in the position of a plural common noun (NN2) in a piece of text, and *can_substitute(NN1,JJ)* means that an adjective (JJ) can be found in the position of a singular common noun (NN1).

The substitution expressed by a *can_substitute* relation is valid only within specific syntactic patterns.

can_substitute(NN1,JJ), for instance, is valid in the context of a syntactic pattern where the singular noun precedes another noun and acts as a qualifier of it. This is because an adjective may also be found before a noun in a sentence and in this case it also acts as a qualifier of the noun (in the requirement statement of Figure 1, for instance, the adjective "provisional" could replace the noun "coursework" and act as a qualifier of the noun "mark"). The same substitution, however, is not valid in cases where the singular noun appears in the end of a sentence since an adjective cannot be found in such a position.

In our approach, such contextual validity conditions are not explicitly expressed in the *can_substitute* relation. They are, however, checked by the algorithm `FindTrs`. The substitution of an adjective for a singular noun, for instance, is accepted only if `FindTrs` encounters an example of a traceability relation that involves a requirement statement which has a sequence of words demonstrating this possibility and matching with the entire syntactic pattern of a traceability rule. The assumption underpinning this approach to ensuring the validity of the accepted tag set modifications is that both the requirement statements which are given as input to the algorithm `FindTrs` and the syntactic patterns of the rules which are transformed by this algorithm are grammatically correct.

The second example in Section 5 demonstrates this form of generalisation.

4.2 Making a word variable non-compulsory

`FindTrs` may modify the qualifier *q* of a compulsory word variable in a rule and make it non compulsory. This happens when the algorithm encounters a word in a requirement statement that should be matched with a variable but whose tag is neither in the set of possible tags of this variable nor it can substitute for any of these tags. In such cases, the variable itself should also not be used in the SUCH THAT conditions of the rule.

The assumption underpinning this form of generalisation is that word variables which are not used in the SUCH THAT conditions of a rule are going to be matched with semantically insignificant words in requirement statements and, therefore, their presence is not important from a semantic point of view.

Clearly, the presence of these variables may be necessary for the correctness of the syntactic pattern expected by the rule. To ensure that this is not the case for a word variable that is to become non-compulsory, `FindTrs` accepts this form of generalisation as part of a rule transformation only if the transformation in its final form (i.e., the form that it takes when the algorithm terminates) makes the rule match with a sequence of words in the requirement statement that prompted the change.

An example of this form of rule generalisation is given in Section 5 (see Example 1 in this section).

4.3 Introduction of new non-compulsory variable in the syntactic pattern of a rule

This form of generalisation may be allowed when `FindTrs` encounters a word in a requirement statement whose tag WT: (a) does not belong to the set of tags of the current word variable of a rule, and (b) cannot be substituted for any of these tags.

In such cases, if WT is a tag that can be associated only with words which cannot be used in the SUCH THAT conditions of traceability rules, `FindTrs` inserts a new non-compulsory word variable in the transformation that is currently under development and puts WT in the set of possible tags of this variable (the new variable is placed before the current word variable of the rule that is being transformed).

The tags which cannot be associated with words that may appear in the SUCH THAT conditions of rules are those which signify: articles (e.g. *the*/AT, *a*/AT1); various forms of determiners (e.g. *much*/DA1, *all*/DB); various forms of syntactic conjunctions (e.g., *and*/CS, *but*/CCB, *unless*/CS); prepositions (e.g., *with*/IW, *of*/IO); comparative, superlative and catenative adjectives (e.g., *better*/JJR, *best*/JJT, *willing*/JK); various forms of numbers; pronouns (e.g., *his*/APPGE, *he*/PPHS1); and adverbs (e.g., *very*/RG, *why*/RRQ). These tags are defined as elements of a set called `NOT_SEMANTIC_TAGS` which is made available to `FindTrs` as background knowledge (see Figure 4).

An example of a rule transformation that includes a new non-compulsory word variable is given in Section 5 (see Example 1 in that section).

5. Examples of rule transformations

In this section, we present two examples of rule transformations which are generated by the algorithms `GenerateRules` and `FindTrs`.

Example 1: Our first example demonstrates the change of a compulsory variable in the syntactic pattern of a rule into a non-compulsory one, and the introduction of a new non-compulsory variable to a rule.

Suppose that the user indicates the existence of a `REQUIRES_EXECUTION_OF` relation between the requirement statement:

```
RS2 = [<The/AT> <system/NN1> <shall/VM>
      <allow/VVI> <an/AT1> <interviewer/NN1>
      <to/TO> <change/VVI> <his/her/APPGE>
      <interview/NN1> <recommendation/NN1>
      <for/IF> <an/AT1> <applicant/NN1>]
```

```
Algorithm FindTrs (R, SQ, RS, V, T, VM, M, CS, S)
Input Variables:
R:   // The traceability rule to be transformed R;
SQ:  // The sequence of syntactic items of a rule R;
RS:  // A req. statement represented as a list
     // of tagged words;
V:   // Current syntactic item in SQ, V = <x/TAGS>:q
T:   // Current tagged word in RS, T = <t/WT>
VM:  // A variable indicating if a match has
     // already been found for the syntactic term V
Output Variables:
M:   // Variable that indicates if a match is found
S:   // Set of transformations that may be applied to
     // SQ to match RS
CS:  // Current substitution
WT := T.Tags;  TAGS := V.Tags;  q:= V.q;
If (T != NIL) and (V != NIL) and (M != False) then
If (WT ∉ TAGS) then
  If q = "*" then // x of V is a multi-occurrence var
    If  VM = False then //x is a not matched yet
      // create transformations with x not matched
      S1 := CS ∪ {<x/TAGS>:q};
      VM1 := False;
      M1 := True;
      FindTrs (R, SQ, RS, V.next, T, VM1, M1, S1, S);
      If exists VT in TAGS and can_substitute (VT,WT)
      then
        // create transformations with x matched
        TAGS' := TAGS ∪ {WT};
        CS := CS ∪ {<x/TAGS'>:q};
        VM := True;
        M2 := True;
        FindTrs (R, SQ, RS, V, T.next, VM, M2, CS, S);
      else //WT cannot substitute for a tag in TAGS
        If (WT ∈ NOT_SEMANTIC_TAGS) then
          // WT is not a tag of items that can be used in
          // SUCH  THAT rule conditions so
          // introduce a new non compulsory var
          CS := CS ∪ {<y/WT>*};
          VM := True;
          FindTrs (R, SQ, RS, V, T.next, VM, M, CS, S);
        else // WT may be used in SUCH THAT conds
          M := False;
        end if
      end if
      If (M1 != True) and (M2 != True) then
        M :=False
      end if
    else // x of V is a matched non-compulsory var
      // create transformations without matching x
      S1 := CS; VM1 := False; M1 := True;
      FindTrs (R, SQ, RS, V.next, T, VM1, M1, S1, S);
      // create transformations by re-matching x
      If exists Y in CS such that Y.Var = V.Var then
        TAGS := Y.Tags;
      end if
      If (WT ∉ TAGS) then
    If exists VT in TAGS and can_substitute (VT, WT)
    then
        TAGS' := TAGS ∪ {WT};
        If exists Y in CS such that Y.Var = V.Var then
          replace Y in CS with <x/TAGS'>:q;
        else
          CS := CS ∪ {<x/TAGS'>:q};
        end if
        M2 := True;
        FindTrs (R, SQ, RS, V, T.next, VM, M2, CS, S);
      else //WT cannot substitute for a tag in TAGS
        if (WT ∈ NOT_SEMANTIC_TAGS) then
          // introduce a new non compulsory var
          CS := CS U {<y/WT>*};
        M2 := True;
        FindTrs (R, SQ, RS, V, T.next, VM, M2, CS, S);
      end if
    end if
  else // WT ∈ TAGS
    M2 := True;
    FindTrs (R, SQ, RS, V, T.next, VM, M2, CS, S);
  end if
    If (M1 != True) and (M2 != True) then
      M :=False
    end if
  end if
  else // x of V is a single-occurrence variable (q = 1)
    If exists VT in TAGS and can_substitute (VT, WT)
    then
      TAGS' := TAGS ∪ {WT};
      CS := CS ∪ {<x/TAGS'>:q};
      VM := False;
      FindTrs (R, SQ, RS, V.next, T.next, VM, M, CS, S);
    else
      if x is not used in R's SUCH THAT conditions then
          // make x a non compulsory var
          CS := CS ∪ {<x/TAGS>*};
          FindTrs (R, SQ, RS, V.next, T, VM, M, CS, S);
      else
        If (WT ∈ NOT_SEMANTIC_TAGS) then
          // WT is not a tag of items that can be used in
          // SUCH  THAT rule conditions so
          // introduce a new non compulsory var
          CS := CS ∪ {<y/WT>*};
          VM := True;
          FindTrs (R, SQ, RS, V, T.next, VM, M, CS, S);
        else
          M := False;
        end if
      end if
    end if
  end if
else // WT ∈ TAGS
  If q = "*" then // x of V is a multi-occurrence var
    // create transformations with x not matched
    S1 := CS; VM1 := False; M1 := True;
    FindTrs (R, SQ, RS, V.next, T, VM1, M1, S1, S);
    // create transformations with x matched
    If there is no Y in CS such that Y.Var = V.Var then
      CS := CS ∪ {<x/TAGS'>:q};
    end if
    VM2 := True; M2 := True;
    FindTrs (R, SQ, RS, V, T.next, VM2, M2, CS, S);
  else // x of V is a single-occurrence variable
    CS := CS ∪ {<V/Tags>:q};
    FindTrs (R, SQ, RS, V.next, T.next, VM, M, CS, S);
  end if
 end if
else // (T=NIL) or (V = NIL) or (M = False)
 If (V = NIL) and (M = True) then
   S = S ∪ CS;
 else
   M := False;
 end if
end if
```

Figure 4. Algorithm for finding rule transformations

and the operation `modifyRecommendation(type: String, rationale: String)` of the class `InterviewRecommendation` in Figure 1. *Rule-1* fails to capture this relation as its syntactic pattern does not match with any sequence of words in RS_2. The application of the algorithm `FindTrs` in this case generates the transformation:

CS_1 = { <x1/{VVI}>:1, <x2/{AT1}>:*,
<x8/{APPGE}>:*, <x3/{NN1}>:1,
<x4/{NN1}>:*, <x5/{NN1}>:1 }

According to CS_1, *Rule-1* can be transformed by: (i) making the variable x2 that can be matched to an article a non-compulsory variable, and (ii) introducing a new non-compulsory variable x8 that may be matched with a possessive pronoun (as signified by the tag APPGE) between the variables x2 and x3.

The first of these generalisations is allowed as the variable x2 is not used in the SUCH THAT conditions of the rule and therefore it is not considered to be a semantically significant variable. The second generalisation is allowed as the tag of the term "his/her" in RS_2 (i.e., APPGE) signifies a possessive pronoun which cannot be used in the SUCH THAT conditions of a traceability rule (APPGE is an element of the set `NOT_SEMANTIC_TAGS`).

The replacement of the syntactic pattern of *Rule-1* by CS_1 transforms this rule into a form that matches with the sequence of words: `change`[x1] `his/her`[x8] `interview`[x3] `recommendation`[x5].

Assuming that the verb "change" is in the set of synonyms of the stereotype of the operation `modifyREcommendation(type: String, rationale: String)` (i.e., `<<set>>`), this matching satisfies the SUCH THAT conditions of the new form of *Rule-1* (they are the same as the SUCH THAT conditions in the original form of the rule). Thus, `GenerateRules` accepts the new form of *Rule-1*.

Example 2: Our second example demonstrates the possibility of generating more than one alternative transformations of the same rule, and the introduction of a new tag into the set of tags of an existing variable.

Suppose that the user indicates the existence of a `REQUIRES_EXECUTION_OF` relation between the requirement statement:

RS_3 = [<The/AT>, <system/NN1>, <shall/VM>,
<allow/VVI>, <the/AT>,
<secretary/NN1>, <to/TO>,
<produce/VVI>, <a/AT1>, <course/NN1>,
<examination/NN1> <statistics/NN>,
<report/NN1>]

and the operation `generateStatisticsReport()` of the class `CourseExaminationStatistics` in Figure 1. *Rule-1* fails to capture this relation because, although its syntactic pattern matches with the sequence of words "produce a course examination" in RS_4, this matching does not satisfy the SUCH THAT conditions of the rule.

The application of the algorithm `FindTrs` in this case generates the following two alternative transformations for *Rule-1*:

CS_2 = { <x1/{VVI}>:1, <x2/{AT1}>:1, <x3/{NN1}>:1,
<x4/{NN1, NN}>:*, <x5/{NN1}>:1 }
CS_3 = { <x1/{VVI}>:1, <x2/{AT1}>:1, <x3/{NN1}>:1,
<x4/{NN1}>:*, <x5/{NN1, NN}>:1}

According to CS_2, the tag NN which signifies common, but neutral for number, nouns is introduced as an alternative tag for the multi-occurrence variable x4. This introduction results as a consequence of attempting to match x4 with the word "statistics" in RS_3 (after having already matched it with the word "examination"). The replacement of the syntactic pattern of *Rule-1* by CS_2 makes the rule match with the following sequence of words in RS_3: `produce`[x1] `a`[x2] `course`[x3] `examination`[x4] `statistics`[x4] `report`[x5].

Thus, `FindTrs` accepts CS_2 as a legitimate transformation of *Rule-1*. Then, assuming that the verb "produce" is in the list of synonyms of the stereotype of the operation `generateStatisticsReport()` (i.e., the stereotype `<<info-production>>`), RS_3 and this operation satisfy the SUCH THAT conditions of the new form of *Rule-1*. Thus, `GenerateRules` accepts CS_2.

The transformation CS_3 introduces the tag NN as an alternative tag for the compulsory variable x5. This introduction results as a consequence of attempting to match x5 with the word "statistics" in RS_3. The replacement of the syntactic pattern of *Rule-1* by CS_3 makes the rule match with the following sequence of words in RS_3: `produce`[x1] `a`[x2] `course`[x3] `examination`[x4] `statistics`[x5].

Thus, CS_3 is accepted as a legitimate transformation of *Rule-1* by `FindTrs`. Subsequently, given the matching of the above sequence of words in RS_3 with the new form of *Rule-1*, RS_3 and `generateStatisticsReport()` satisfy the SUCH THAT conditions of this form and, therefore, `GenerateRules` accepts it.

Note that in this example, `GenerateRules` is not capable of establishing a preference between the two rules that can be created from *Rule-1* by applying CS_2 and CS_3 as both these rules are of the same generality.

6. Related Work

Since the mid-nineties, the investigation of applications of machine learning techniques to problems arising in

requirements engineering has attracted the attention of researchers in the requirements engineering and machine learning communities.

In [19] van Lamsweerde and Willemet present an *inductive goal inference* procedure which supports the elicitation of goal-oriented requirements from sets of operational scenarios provided by the user. In [1], machine learning is used in conjunction with *abduction* to support the evolution of requirements specifications using the "Analysis-Revision Cycle" framework. Learning techniques have also been used for validation of domain specific requirement models (e.g. an air traffic control model in [7]).

Finally, although not directly related to requirements engineering, work in the area of learning formal grammars [11][15] may inform the future development of the approach presented in this paper. Ideas and algorithms for representing and learning recursive productions in a formal grammar may, for example, be applicable to the problem of learning recursive traceability rules.

7. Conclusions and Future Work

In this paper we presented a machine learning approach for generating requirements traceability relations. This approach is based on a new learning algorithm that produces traceability rules which are able to capture traceability relations between requirement statements specified in natural language and object models. The creation of these traceability rules is informed by examples of such relations which are provided by the user and is based on the generalisation of existing traceability rules. Currently, we are conducting a series of experiments (along the lines reported in [20]) for validating the proposed approach. Our experiments are based on an industrial case study that includes the specification of commercial requirements and use-cases for a family of software intensive TV products.

References

[1] d'Avila Garcez A., Russo A., Nuseibeh B., Kramer J., "Combining Abductive Reasoning and Inductive Learning to Evolve Requirements Specifications", (to appear in) IEE Proceedings - Software, 2003.

[2] Dömges R., Pohl K. "Adapting Traceability Environments to Project-Specific Needs", Communications of ACM, Vol. 41, No 21, 1998.

[3] Garside R., Smith N., "A Hybrid Grammatical Tagger: CLAWS4", Corpus Annotation: Linguistic Information from Computer Text, Longman 1997.

[4] Gotel O., Finkelstein A., "Contribution Structures", Proc. of 2nd Int. Symposium on Requirements Engineering, 100-107, 1995.

[5] Integrated Chipware, RTM, www.chipware.com

[6] Kaelbling K., Littman M., Moore A., "Reinforcement Learning: A Survey", Technical Report, Carnegie Mellon University.

[7] McCluskey T., West M., "The Automated Refinement of a Requirements Domain Theory", Journal of Automated Software Engineering, 8(2): 195-218, 2001.

[8] OMG Unified Modelling Language Specification, ftp://ftp/omg.org/pub/docs.ad/99-06-08.pdf.

[9] Pinheiro F., Goguen J., "An Object-Oriented Tool for Tracing Requirements", IEEE Software, 52-64, March 1996.

[10] Pohl, K., Dömges, R., and Jarke, M. "Towards Method-Driven Trace Capture", Proc. of the 9th Int. Conference on Advanced Information Systems Engineering, 1997.

[11] Pollack J., "Recursive Distributed Representations", Artificial Intelligence 46, 1, 77-105, 1990.

[12] Ramesh B., Dhar V., "Supporting Systems Development Using Knowledge Captured During Requirements Engineering", IEEE Transactions in Software Engineering, 498-510, 1992.

[13] Ramesh B., Jarke M., "Towards Reference Models for Requirements Traceability", IEEE Transactions in Software Engineering, 27(1) , 58-93, 2001.

[14] Ramesh B., Powers T., Stubbs C. and Edwards M., "Implementing Requirements Traceability: A Case Study", Proc. of 2nd Int. Symposium on Requirements Engineering, 89-95, 1995.

[15] Tesar B., Smolensky P., "Learning Optimality-Theoretic Grammars", Lingua, 106:161-196, 1998.

[16] Spanoudakis G., "Plausible and Adaptive Requirement Traceability Structures", Proc. of 14th Int. Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, July 2002.

[17] Spanoudakis G., Zisman A., Perez-Minana E., Krause P., "Rule-Based Requirements Traceability", Technical Report, SEG-01-03, Dept. of Computing, City University, 2003.

[18] Teleologic, Teleologic DOORS, www.teleologic.com/products/doors

[19] van Lamsweerde A., Willemet L., "Inferring Declarative Requirements Specifications from Operational Scenarios", IEEE Transactions on Software Engineering, 24(12):1089-1114, 1998.

[20] Zisman A., Spanoudakis G., Perez-Minana E., Krause P., "Towards a Traceability Approach for Product Families Requirements", Proc. of 3rd ICSE Workshop on Software Product Lines, 2002