A PRODUCTION SYSTEM FOR INFORMATION EXTRACTION BASED ON COMPLETE SYNTACTIC-SEMANTIC ANALYSIS

Starostin A. S. (astarostin@abbyy.com),
Smurov I. M. (ismurov@abbyy.com),
Stepanova M. E. (mstepanova@abbyy.com)

ABBYY, Moscow, Russia

The article presents a mechanism for information extraction from unstructured natural language data. The key feature of this mechanism is that it relies on deep syntactic and semantic analysis of the text. The system takes a collection of syntactic-semantic dependency trees as input and, after processing them, outputs an RDF graph consistent with certain domain ontology.

The mechanism was implemented within a deployable information extraction system, which is a part of ABBYY Compreno technology—a powerful tool for a broad range of NLP-tasks that include machine translation, semantic search and text categorization. The description of the extraction algorithm and the results of the system performance evaluation are available in the article.

Evaluation tests were conducted on the MUC-6 corpus. The overall F-measure we achieved using Compreno technology was 0.83, which is lower than the best results claimed by the researchers using machine learning approaches. Our system is still in development at the moment and we hope to improve its performance in the future. One of the advantages of Compreno technology is that, unlike many statistical approaches, it does not show an abrupt performance drop if the test corpus is changed. Thus Compreno demonstrates little dependence on the exact textual data it receives and therefore might be seen as a more universal and less domain-dependent solution. Our tests on the CoNLL corpus yielded an F-measure of 0.75 with no prior adjustments made.

Key words: information extraction, named entity recognition, syntactic analysis, anaphora and coreference resolution, production rule systems

Introduction

The article describes an information extraction method which is the core of the data mining system that has been in development by ABBYY over the last three years. This system is an integral part of a more universal text analysis technology known as ABBYY Compreno. Its key feature is the ability to perform complete syntactic-semantic analysis of the input text.

At the first stage a given input is analyzed by the Compreno parser [1]. The result is a collection of syntactic-semantic dependency-based parse trees (one tree per sentence). Nodes and edges of each tree are augmented with diverse grammatical and semantic information. The parse tree forest is then used as input for a production system of information extraction rules. The application of the rules results in the formation of an RDF graph consistent with a certain domain ontology.

In the first section of the article we provide a detailed description of the information extraction mechanism. We describe the input data, the method used to represent extracted information, the structure of the extraction rules and the algorithm of their execution.

The approach we propose demonstrates two significant advantages. Firstly, the availability of syntactic and semantic structure allows us to extract facts1 as well as entities. Fact extraction rules that rely on the structure of syntactic-semantic trees tend to be laconic yet highly efficient, easily covering most natural language expressions. Secondly, the system shows little dependence on a particular language. Since our parse trees contain language-independent data (like semantic roles or universal semantic classes), many extraction rules are universal and can be used for different languages.

Despite the fact that we use declarative rules in our system, our approach to information extraction cannot be described as a rule-based one, because the syntactic and semantic analysis that precedes the extraction is not based on a set of rules. The sort of analysis performed by the Compreno parser can be defined as model-based: it rests upon a multilevel model of natural language created by linguists and then corpus-trained. Thus it is possible to consider our method hybrid, it being model-based at the first (preparatory) stage and rule-based at the second.

In the second part of the article we provide the results of the tests we conducted to evaluate our system's performance. We used the MUC-6 corpus to run the tests and chose a standard set of information objects (Person, Organization, Location, Time and Date) for evaluation.

1. Information extraction mechanism

This section describes the logic behind the information extraction mechanism. In the first place, we introduce the basic concepts necessary for the description. Secondly, we give a description of the implemented algorithm.

1.1. Input

The input accepted by the information extraction mechanism is a sequence of syntactic-semantic trees (one tree per sentence). These trees are generated by the Compreno parser during the analysis. Each tree is projective and its nodes in most cases correspond to the words of the respective sentence, although there are some

[1] Currently the authors of this article are working on a new paper dedicated to fact extraction. That paper will describe the benefits our approach brings to this field of research.

null-nodes with no surface realization. Nodes and edges of a tree are augmented with grammatical and semantic information.

Detailed description of the linguistic model implemented within the Compreno parser is beyond the scope of this paper. More information about the system can be found in [14]. We will limit ourselves to a brief outline of the features that are most important in the context of information extraction.

1. Compreno technology is based on a universal semantic hierarchy. The hierarchy is essentially a tree of IsA-relations with universal semantic classes as non-terminal interior nodes and language-dependent lexical classes as terminal leaf-nodes. Each node bears a set of semantic restrictions and syntactic constraints. These constraints as well as other properties can be inherited by hierarchy elements from their ancestor nodes. All the features that are present in an ancestor node are also present in its offspring unless explicitly stated otherwise.
2. Compreno morphological model exists outside the semantic hierarchy. For each language there is a list of lexemes[2] and their paradigms. Within the hierarchy each lexeme can be attached to one or more lexical classes. A lexical class usually binds together several lexemes, for example "redeem" and "redemption".
3. Each node of a parse tree is attached to a certain lexical class of the hierarchy, which implies that the parser performs word sense disambiguation in the course of analysis. The disambiguation algorithms exploit restrictions of the semantic hierarchy as well as co-occurrence statistics from parallel corpora.
4. Each node also stores grammatical and semantic information that defines its current role in the text (a set of grammemes and semantemes).
5. Each arc of a tree stores a surface slot (i.e. syntactic function of the dependent node like $Subject or $Object_Direct) and a deep slot[3] (i.e. semantic role of the dependent node like Agent or Experiencer). The set of deep slots is universal and language-independent, while sets of surface slots differ between particular languages.
6. Apart from the syntactic-semantic trees the Compreno parser provides the information about non-tree links between the nodes, such as coreference. For a sample sentence *John stood up and shouted* the parser will insert the omitted subject for the second verb and connect this reinstated node to the *John* node via a non-tree link. There are several other types of non-tree links aside from coreference. In some cases such links can connect nodes of different sentences—this is especially typical of pronominal anaphora, where antecedent is often found at a considerable distance from its anaphor. The importance of non-tree links for information extraction is demonstrated in the next section of the article.

[2] The term lexeme here is viewed as in traditional morphology. A lexeme combines different word forms of a word (or its paradigm). Different meanings of a word are not taken into account.

[3] The closest analogues of deep slots in the Western linguistics are Charles Fillmore's 'deep cases' [4].

It is important to note that the ultimate goal of Compreno is to convert text into a language-independent structure based entirely on universal elements of meaning, since the technology was initially developed (and is currently used) for machine translation[4]. Moreover, when we approach this particular task of information extraction, we also try to use the deep universal structure of text. This approach liberates our extraction rules from dependence on a particular language and gives them a broader range. However, in some situations we have to use surface syntactic structure in order to extract information efficiently. In both cases Compreno parse trees provide us with all the necessary information about syntax and semantics of the text.

Finally, there is a possibility for the information extraction module to address the input text directly regardless of the syntactic-semantic trees. For instance, if the input text had been marked with a predefined set of tags, the extraction system can take these tags into account, and there is a special operator for dealing with such tags in the extraction rules' syntax.

1.2. Extracted information

The output of the extraction mechanism is an RDF graph. The idea of RDF (Resource Definition Framework, [12]) is to assign each individual information object a unique identifier and store the information about it in the form of SPO triples. S stands for subject and contains the identifier of an object, P stands for predicate and identifies some property of an object, O stands for object and stores the value of that property. This value can be either a primitive data type (string, number, Boolean value) or an identifier of another object.

All the RDF data is consistent with an OWL-DL[5] ontology [10] which is predefined and static. Information about situations and events is stored in a way that is ideologically similar to that proposed by W3C consortium for defining N-ary relations [3]. The consistency of the extracted information with the domain model is a built-in feature of the system. It is secured, firstly, by the extraction rules syntax and, secondly, by validation procedures that prevent generation of ontologically inconsistent data.

In addition to RDF graph, extraction mechanism generates annotations, i.e. the information that links extracted entities to the respective parts of the original text. The combination of an RDF graph and annotation links will hereinafter be called an annotated RDF graph.

An annotated RDF graph is generated at the very final stage of the information extraction process. Until that we use a more general structure to store extracted information during the process. This structure can be described as a set of noncontradictory statements about information objects and their properties. Further on we will often call that a "bag of statements". The number of statement types is finite, and

[4] Theoretical prerequisites for the partition between the surface level and the deep level of syntactic structure representation can be found in [2] and [9].

[5] The OWL DL language subset that we use is similar to OWL Lite, but we also exploit DisjointWith axiom.

we describe them below. Running a few steps forward, we have to note that all the statements are generated by information extraction and identification rules.

However, the final annotated RDF graph can also be viewed as a bag of statements, if each SPO triple and each link from an object to a segment of text is considered a statement about that object. Therefore it is important to point out the difference between our temporary information storage structure (the inner structure) and the final output in the form of an RDF graph.

The main distinction is that the statements from the inner structure can be used to create functional dependencies. For instance, we can state that a set of values of a certain object`s property should always contain a set of values of some other property of a different object. If the set of values of the second object is changed, the first object's property changes as well. We hereinafter refer to such statements (that use functional dependencies) as *dynamic* statements. Another difference of the inner structure is that it may contain some auxiliary statements that do not comply with the final annotated RDF graph structure and are used only during the extraction process.

The bag of statements has several important properties:

1. **Cumulativity**. Statements can be added to but not removed from the bag.
2. **Consistency**. All the statements in the bag are non-contradictory to each other[6].
3. **Consistency with ontology**. The bag of statement can anytime be converted into an annotated RDF graph consistent with certain ontology.
4. **Transactionality**. Statements are added in groups, and if any statement of a group contradicts other statements from the bag, the addition of the whole group is cancelled.

Here is the list of statement types:

1. Existence statements

Existence statements proclaim the existence of information objects and assign unique identifiers to them.

2. Class membership statements

Statements that attribute objects to classes in the ontology. OWL allows us to attribute a single object to several classes, so there can be more than one class membership statement in the bag. The only restriction is that the classes should be consistent with each other, i.e. there should not be a DisjointWith statement blocking the combination of classes. The system checks for disjoint every time statements are added to the bag and prevents inconsistencies.

Class membership statements can be dynamic: we can state that an object is attributed to the same set of classes as some other object.

[6] Some examples of unallowable contradictions can be found further in the article

3. Property statements

Statements that define properties of objects. With a property statement we can assert that a set of values of an object`s property includes some particular value. To comply with the RDF standard it can be either an identifier of a different object or a primitive data type (a string, a number or a Boolean value). In our system we also use parse tree node identifiers as property values (an additional primitive data type). Properties of this sort are only used during the extraction process but do not appear in the final RDF graph.

Property statements can be dynamic. The complexity of functions that calculate values of objects from certain properties of other objects can vary. The simplest example is a function that copies values (i.e. it makes a statement that a set of values of some property of an object includes all the values of some other property of a different object). A complex example is a function that generates a normalized string from a set of parse tree nodes. This function relies on Compreno text generation module.

Together several statements of some property of an object can create ontological inconsistencies. For instance, the number of values may exceed the maximal cardinality of that property. Our module prevents such inconsistencies by rejecting any group of statements that provokes contradiction.

4. Annotation statements

Annotation statements connect information objects to parts of the original input text. Annotation coordinates are calculated from the bounds of syntactic-semantic tree nodes. Annotation can cover either a single node (i.e. a word), or a full subtree of that node.

The bag of statements can contain a number of annotation statements. This means that an annotation of an object can consist of more than one segment (i.e. be discontinuous)[7].

Annotation statements can be dynamic. For instance, an annotation can be copied from a different object or be generated from a set of values of a certain property if these values contain links to parse tree nodes[8].

Annotation statements cannot create any contradictions.

5. Anchor statements

Anchor statements are a very important part of our information extraction mechanism. Statements of this type link information objects to parse tree nodes, which enables us to access these objects later during the extraction process. The term ‘anchor’ was coined when the system was in development so that the links between objects and

[7] The information extraction module has a special built-in algorithm that optimizes the set of segments linked to an information object immediately before the final annotated RDF graph is generated. This algorithm deletes embedded segments and in some cases merges contiguous segments into one.

[8] The second capability considerably simplifies operations with object annotations since it allows us to store parse tree nodes as values of auxiliary properties while different rules extract different parts of the same object. We do not have to intentionally expand the annotation in any of the rules—it extends automatically.

tree nodes could be easily referred to. One object can be anchored to a set of nodes via a number of anchor statements.

The interpreter of the information extraction rules (which we describe later in the article) deals with these anchors in a special way: the left-hand side (or condition side) of a rule in our system can contain the so-called **object conditions**, which imply that an information object of a certain type must be assigned (anchored) to a certain parse tree node for the successful application of the rule. If such an object is found it can be accessed and modified during the application of the right-hand side of the rule.

Object conditions are most widely used in the rules that extract facts, but they are quite useful with named entities as well, since they make it possible to break the extraction of entities of particular type down to several simple stages. For instance, one rule might only create an unspecified Person entity, while the following ones add properties like first name, surname, middle name and alike. It has also become quite common to create auxiliary objects which serve as dynamic labels of parse tree nodes. First some rules create these auxiliary objects and anchor them to certain nodes, and then other rules check for the presence of these objects with the help of object conditions in their left-hand sides.

An anchor statement can attach an anchor not only to the explicitly indicated node, but also to all its coreferring nodes (via non-tree links of syntactic-semantic trees). This possibility is crucially important for the recall of fact extraction, since the extracted information objects are automatically linked to coreferents. As a result the object appears simultaneously in several contexts and can be used by fact extraction rules.

Anchor statements cannot create any contradictions.

6. Identification statements

During the extraction process it is often possible to recognize objects which actually refer to a single real-life entity and should therefore be merged. One obvious example is when a person appears several times in a text. At the first stage each mention of that person is extracted as a separate information object, but we can merge them subsequently if their surnames and names match.

Two objects can be merged into one via identification statements. After these statements are added to the bag, all statements about the merged objects are reassigned to this newly created composite object.

Identification statements can contradict to other types of statements. For example, classes of two objects can be incompatible with each other or a value of some property might exceed its maximum cardinality restriction that is set in the ontology. There is also a possibility of other, more complex inconsistencies.

7. Functional restrictions

In some cases it is convenient to impose a restriction upon a group of objects. We can add a function that accepts identifiers of information objects and some constant values (e.g. identifiers of parse tree nodes) as input and returns a Boolean value. A function must be true when it is added to the bag. After it has been added no statement that would make the function false can enter the bag.

Figure 1 contains schematic diagrams of all statements types available in our system.

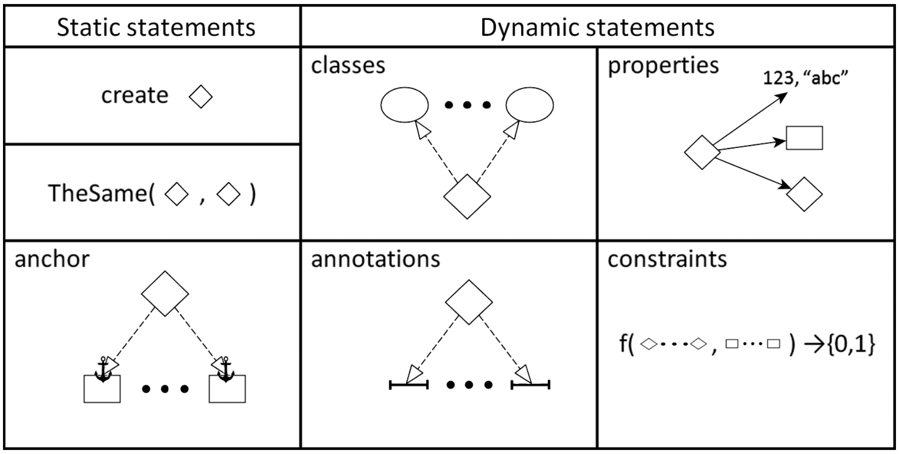


Fig. 1. Types of statements used in the information extraction process. Diamonds represent information objects (individuals), ellipses represent classes (or concepts) and rectangular boxes represent parse tree nodes

As mentioned above, our statements can be dynamic, i.e. they can depend on other statements. It is important to note that this feature can lead to contradictions caused by the dependent statements rather than the statement being added at the moment. That fact posed certain difficulties to the realization of an algorithm that emulates the bag of statements. However, all these issues were subsequently addressed.

Most of the consistency checking is performed when statements are added to the bag. However some tests can only be conducted after the information extraction process is over. For instance, we cannot know if some property meets minimum cardinality requirement until all the rules are executed. After the extraction process is complete and before the bag of statements is converted into an annotated RDF graph we also filter some auxiliary information (e.g. auxiliary objects or properties).

Now that the reader has a relatively complete picture of the way information is stored during the extraction process, we proceed to the description of the mechanism that implements the extraction rules and produces statements on information objects.

1.3. Information extraction rules

Information extraction process is controlled by the production rule system. There are two types of rules in the system: parse subtree interpretation rules (or simply interpretation rules) and identification rules. Both types of rules are described further in the article. Since interpretation rules are much more frequent, whenever we do not specify the exact type of a rule the reader should assume it is an interpretation one.

During the development of the extraction mechanism several goals were pursued. In the first place, our intention was to exploit such advantages of the production rule systems as modularity [11] and separation of knowledge from the procedure.

We particularly wanted to relieve the developers from the necessity to order the rules[9]. Secondly, we intended to implement an efficient deterministic inference model. Speaking in terms of traditional production systems [5] we can define parse tree forest and the bag of statements as our knowledge base, while the extraction process itself can be described as a forward chaining inference process. Generally speaking, there is no guarantee that the rule execution will not loop[10]. However, if the cycle occurs in the real rule system that definitely means that there is a logical mistake in some rule. Usually, it can be easily found and corrected, since there is a built-in heuristics in the algorithm allowing us to detect rules which caused cycles.

Before we proceed to the detailed discussion of extraction rules we have to point out that a full description of the extraction rule syntax is well beyond the scope of this article. We will limit ourselves to a schematic outline and examples.

1.3.1. Parse tree interpretation rules

Interpretation rules enable us to specify fragments of parse trees, which must be discovered for certain logical statements to become true. A rule is basically a production with syntactic-semantic tree patterns in its left-hand side and some expressions that make statements on information objects in the right-hand side.

Parse tree templates (hereinafter tree templates) are formulas each individual element of which checks some property of a tree node (e.g. presence of a certain grammeme or semanteme, belonging to a certain semantic or lexical class, occupation of a certain deep or surface slot and many other properties available from the parsing results). Apart from the basic logical operators (conjunction, disjunction, negation) tree templates allow us to check relative position of nodes within a tree. For instance, we can check if a node is in a subtree of another node.

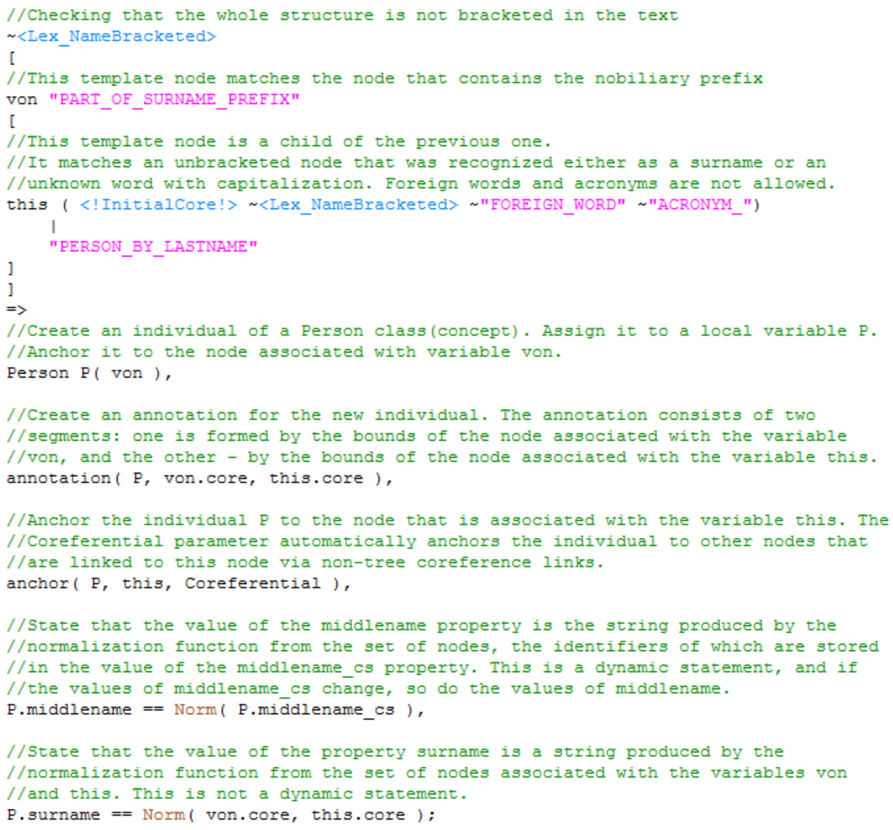
In most cases tree templates describe the interconnected segments of syntactic-semantic trees (i.e. subtrees). The only exception is a special anaphoric condition. This condition allows us to search nodes in the left context of a certain node completely ignoring tree structure and surpassing boundaries of a single sentence. Such rules are used for coreference resolution, especially in cases of nominal anaphora.

Tree templates can contain conditions that require an information object to be anchored to a certain node of a parse tree. We call such requirements positive object conditions. Our rules also support negative object conditions that require a node not to have an object of a certain type attached to it. We already mentioned object conditions in the part about anchor statements.

[9] One particular example of quasi-production language that does not comply with this requirement is Jape ([8]). Jape requires setting the order in which groups of production rules (or phrases) are executed explicitly. During their execution rules within a group do not have the access to each other's results. In the process of development of such rules it often occurs that the rules which create an object of a certain type are executed after the rules which accept such an object as their input. Moreover, it is not possible to reorder the rules because rules from the first group might also use some objects created by the second group. The only solution to this problem is to launch the same groups of rules several times. However, this solution is far from being ultimate since it artificially limits the number of recursion steps.

[10] It is possible to create a set of rules that will loop infinitely.

When we add a statement to the right-hand side of a production it is often necessary to refer to the nodes of the subtree that matches the template in the left-hand side and sometimes to the information objects attached to these nodes. For that purpose we introduce names (or variables) for separate parts of tree templates. If a certain subtree matches a template, its nodes can be accessed via the variables assigned to the template parts. These variables can be used in the right-hand side of a production to make statements on objects. In some cases they can also be accessed in the left-hand side (in order to create a complex condition that checks for certain dependence between several tree nodes). A variable can be either a set variable or a unique one. A set variable can be associated with several nodes, while a unique variable can only hold one node[11] as its value.



```
//Checking that the whole structure is not bracketed in the text
~<Lex_NameBracketed>
[
//This template node matches the node that contains the nobiliary prefix
von "PART_OF_SURNAME_PREFIX"
[
//This template node is a child of the previous one.
//It matches an unbracketed node that was recognized either as a surname or an
//unknown word with capitalization. Foreign words and acronyms are not allowed.
this ( <!InitialCore!> ~<Lex_NameBracketed> ~"FOREIGN_WORD" ~"ACRONYM_")
    |
    "PERSON_BY_LASTNAME"
]
]
=>
//Create an individual of a Person class(concept). Assign it to a local variable P.
//Anchor it to the node associated with variable von.
Person P( von ),

//Create an annotation for the new individual. The annotation consists of two
//segments: one is formed by the bounds of the node associated with the variable
//von, and the other - by the bounds of the node associated with the variable this.
annotation( P, von.core, this.core ),

//Anchor the individual P to the node that is associated with the variable this. The
//Coreferential parameter automatically anchors the individual to other nodes that
//are linked to this node via non-tree coreference links.
anchor( P, this, Coreferential ),

//State that the value of the middlename property is the string produced by the
//normalization function from the set of nodes, the identifiers of which are stored
//in the value of the middlename_cs property. This is a dynamic statement, and if
//the values of middlename_cs change, so do the values of middlename.
P.middlename == Norm( P.middlename_cs ),

//State that the value of the property surname is a string produced by the
//normalization function from the set of nodes associated with the variables von
//and this. This is not a dynamic statement.
P.surname == Norm( von.core, this.core );
```

Fig. 2. An example of a rule performing Person extraction. The rule deals with the case where a person is mentioned with a nobiliary prefix

[11] There are parts of tree templates that can never match more than one node of a parse tree. These parts can be determined during the template compilation. Variables attached to these parts are declared unique.

The second example (Figure 3) shows a fragment of a rule (only part of its right-hand side is shown), in which a positive object condition is used. This rule demonstrates a way to add more information to the properties of already existing (i.e. previously created) objects. The rule deals with cases where a person has middle names (Albus Percival Wulfric Brian Dumbledore).

To access an information object that matched a positive object condition we use a special notation "*X.o*" where *X* is the name of the unique variable assigned to the node at which the condition was introduced. The variable *X* has to be unique since each time we process the object-condition during the interpretation of the rule we need to know the exact tree node the information object must be anchored to.

Figure 2 demonstrates an example of a rule that performs person extraction. This particular rule deals with the case where a person is mentioned with a nobiliary prefix (von Bismark, da Silva etc). Square brackets define a child node. The left-hand side of the rule contains two variables: *von* and *this*. The right-hand side of the rule makes statements which address these variables.

```
// Template root - a node that is a personal first name
this "PERSON_BY_FIRSTNAME"
[
//Looking through the entire subtree of the root node for a node that would
//be either a surname or a nobiliary prefix. If found, this node is assigned to
//the variable P. En exclamation mark states that the variable is disjunctive -
//if there are several nodes meeting the requirements they will be matched one
//by one in separate launches of the production. Disjunctivity of P guarantees
//that it is always unique.
//We also make sure that all the nodes between the root and the P node are
//personal first names and occupy the Classifier_Name deep slot. All the nodes
//in this path are assigned to the variable middle.
//Finally, there should be a Person object already anchored to the P node. This
//object's surname property
...( Classifier_Name: middle "PERSON_BY_FIRSTNAME" )
!P ( "PERSON_BY_LASTNAME" | "PART_OF_SURNAME_PREFIX" )
<% Person, surname ~= null, firstname == null %>
]
=>
//State that the set of values of middlename_cs property contains the set of nodes
//associated with the variable middle
P.o.middlename_cs == middle,
```

Fig. 3. An example of a rule performing Person extraction. The rule deals with the case where a person has middle names

1.3.2. Identification rules

Identification rules are used to merge (unite) a pair of objects. An identification rule is basically a production with object conditions for two objects in the left-hand side. If a pair of objects fulfils these conditions, the objects are merged into one. The right-hand side of an identification rule is omitted because it is always the same: a statement that the two objects are identical (an identification statement).

We use three types of conditions in the identification rules. Conditions of the first type describe the properties of the objects separately, while those of the second and the third allow to impose restrictions on both objects simultaneously (first and foremost, the intersection of values of certain properties). Conditions of the first type are written in the same syntax as the object conditions in the interpretation rules. Conditions of the second type are formulae with property intersection statements as basic element and conjunction and disjunction as logical operators. Such formulas can efficiently filter the number of potentially identical objects. Conditions of the third type are functions written in a JavaScript extension [7]. If such a function is present, the rule will only be applied if it returns true.

A significant difference of identification rules from interpretation rules is that the former can operate only with information objects and have no access to parse tree nodes. We assume that all the information necessary for the identification should be stored within the properties of objects (including auxiliary properties unavailable to the end user).

An example of an identification rule is shown in Figure 4.

```
<% Person %>
<% Person %>

//State that the values of the surname property of the two objects should intersect.
intersects( surname ),
//Same requirement imposed on the firstname property.
intersects( firstname )

::

//The script part of the rule. Here we compare the values of the gender property
//of the objects. o1 and o2 are the standard names (variables) to mark information
//objects in scripts. They are assigned to the first and the second object condition
//respectively.

gen_1 = o1.gender;
gen_2 = o2.gender;

//If any of the two objects does not have this property, we consider the matching
//correct, so the function returns true
if( gen_1.length == 0 || gen_2.length == 0 )
{
return true;
}
//Values of properties are always compared to each other as sets. In this particular
//case each of them is a one-element set.
return SameSets( gen_1, gen_2 );
```

Fig. 4. An example of identification rule for Person object

The sample rule above contains all three types of conditions available in the identification rules: object conditions (check that the objects belong to class Person), intersection checking (two intersection statements connected by a conjunction operator) and a simple script function.

Now that all the necessary notions have been introduced we can proceed to the description of the information extraction algorithm.

1.4. Information extraction algorithm

While describing the information extraction algorithm we use the generic term 'matching'. By this term we mean both a match of a tree template in an interpretation rule with a segment of an actual parse tree and a match of an identification rule with a certain object.

A matching of a tree template with a segment of a tree can be represented as a pair $<r, V_r>$, where r is the unique identifier of a rule and V_r is a set of mappings where

- Each set variable of a rule *r* is associated with a set of syntactic-semantic tree nodes.
- Each unique variable is associated with precisely one node.
- Each unique variable with a positive object condition holds an information object.

It is important to point out that finding a matching is a sufficient condition for the right-hand side of the rule to be converted into a set of statements.

For identification rules a matching is a triple $<r,o_1,o_2>$, where *r* is the identifier of a rule and o_1 and o_2 are the information objects. These objects correspond to the first and the second object condition respectively. As in the interpretation rules, if there is a specific matching found for an identification rule, it becomes possible to process its right-hand side, i.e. to make an identification statement about the two objects.

The information extraction algorithm has the following steps:

1. Analyze the input text with the Compreno parser to get a forest of syntactic-semantic parse trees.
2. Find all the matchings for the interpretation rules that do not have object conditions.
3. Add the matchings to the sorted match queue
4. If the queue is empty, terminate the process.
5. Get the highest priority matching from the queue
6. Convert the right-hand side of the corresponding rule into a group of statements.
7. Try to add the statements to the bag.
8. If failed, declare matching invalid and go to step 4.
9. If succeeded, initiate new matchings' search.
10. If found, add new matchings to the queue. Go to step 4.

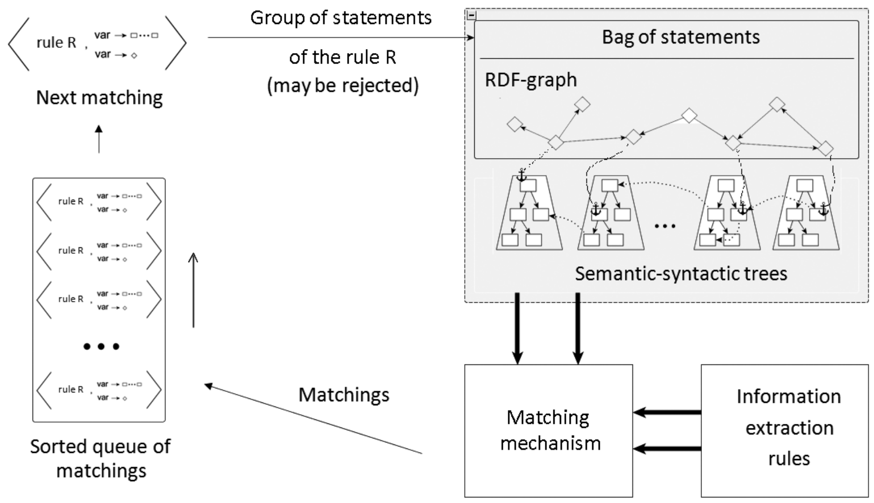


Fig. 5. Schematic representation of the information extraction process

Some parts of the above algorithm need to be described more thoroughly. Steps 2 and 9 are performed with the help of a special matching mechanism. This mechanism can retrieve all the matchings for the rules without object conditions. It also constantly monitors the contents of the bag of statements. Every time step 7 is performed successfully and new statements get into the bag, the mechanism takes them into account and, if necessary, generates new matchings for the rules that do contain object conditions. These new matchings can be created both for the rules that have already been matched before and for those which remained unmatched until that moment. The former occurs when an object condition of a certain rule is matched by more than one object. In this case each object is matched in a separate matching.

The implementation of the matching mechanism is relatively complex. For instance, it has a built-in bytecode interpreter for the compiled rules, a system of indexes for the syntactic-semantic trees, a module for tracking changes in the bag of statements and several other features. Full-length description of this mechanism is beyond the scope of the paper.

It is also important to explain the way the queue of matchings is sorted at the third step. In some cases developers can set the order of rules, i.e. there is partial order over the whole set of rules. Of any two rules one can be given priority over the other. It means that if both rules are ready to be applied, the rule with the higher priority should go first. For convenience reasons we also support group ordering of rules. If group A was given priority over group B, then each rule belonging to group A has higher priority than one belonging to group B. Partial order relation is transitive. Correctness of partial order is checked every time a system of rules is compiled. If loops are detected, compilation fails and the user receives an error message. The order of matchings in the queue is always consistent with the partial order set within a system of rules.

This approach differs significantly from those with consecutive execution of rules, since partial order only determines the priority of rules and does not prevent repeated execution.

It is easy to see that the described algorithm does not consider alternatives. If a group of statements derived from some matching is inconsistent with the bag of statements in its current state, the matching is simply dismissed. We can afford to use this 'greedy' principle because our parser performs word-sense disambiguation, so we rarely ever have to hypothesize about a node. There are some exceptions like words unknown to the parser, but for such cases we have special methods of dealing with these words and incorporating them in our model.

2. Evaluation

We tested our system on the texts that were manually annotated with name entities for the 6th Message Understanding Conference (MUC-6) held in November 1995 [6]. Today the MUC-6 data set is considered one of the main evaluation benchmarks for named entity recognition.

The MUC-6 corpus contains 318 manually annotated Wall Street Journal articles dating from January 1993 to June 1994. Selection of the articles was not random since they were also used for the Scenario Template task which had three specific scenarios: 'aircraft order', 'labor negotiations' and 'management succession' [13].

These texts were annotated with information objects of standard types: named entities (persons, locations and organizations), temporal expressions (mentions of specific time or dates) and numerical expressions (money and percentages). For the purposes of the conference all the articles were divided into training and test sub-corpora. The final test corpus used for the named entity recognition task contained 30 articles.

Compreno does not need a training corpus, and therefore all 318 articles were used as a test corpus.

2.1. Entity types used for evaluation

During the evaluation we tested the extraction of the following types of information objects: Person, Location, Organization, Time, Date, Money. There are about 1500 productions in our system created to extract these kinds of entities.

The basic principles of annotation for Persons, Locations, Organizations, Dates and Money used in the MUC-6 corpus are very similar to those used in our model. Some exceptions are shown further in the article.

Annotation standards for Time in the MUC-6 corpus differ greatly from these in our model. Nevertheless, we decided to try and compare our results for Time objects with MUC gold standard anyway.

However, we did not do any tests for Percentage entity, since within our model it is not viewed as a distinct type.

It is also worth taking into consideration that MUC-6 gold standard allows alternative annotation variants. These alternatives are added to entities as their attributes *(3) "ALT" (<TIMEX TYPE="DATE" ALT="1987">all of 1987</TIMEX>.*

When we compared our results (i.e. the automatic markup) to the gold standard, this attribute was ignored.

2.2. System enhancement

To make the comparison of our automatic markup with the gold standard fair and correct we had to modify our basic system of rules. The modification had been driven by the fact that in some cases our ideas of how an entity should be annotated and where should its boundaries lie differed from those of the MUC-6 annotators.

For instance, we had to ignore anaphoric references to objects of any types in the process of comparison. We also made some separate alterations for each type of entity. All the changes were generalized and none of them had anything to do with specific individual objects or with extraction errors that occurred.

Locations. For information objects of this type we had to restrict the extraction of attributive mentions ("Japanese bank", "Boston University"). We also restricted some location types that were completely absent from MUC-6 (planets, airports etc). We also excluded some keywords from annotations since they were not annotated in the gold standard (**state of** Michigan, **city of** Hiroshima).

Person. The rules for person extraction were changed so that honorifics ("Mr", "Ms", "Sir",…) were not included in the annotation. We also forbade extracting persons on the names of the saints (*St Paul*).

Organization. The rules that extracted abstract governmental organizations (*government, police* etc) were disabled.

Money. The rules were altered so that the words meaning the approximate amount (***around*** *$1.6 billion*) were excluded from the annotation.

Time and dates. The rules that extracted relative time points and periods (this year, today, for two years) were disabled.

2.3. Evaluation results

The comparison of the test automatic markup with the gold standard showed following results:

Table 1. Evaluation results

Type of entity	Precision	Recall	F-measure
All entities	0.853	0.813	0.832
Money	0.947	0.933	0.940
Person	0.700	0.887	0.783
Location	0.936	0.806	0.866
Organization	0.767	0.639	0.697
Date	0.941	0.880	0.910
Time	0.674	0.573	0.620

These results are lower than the numbers shown by statistical systems on MUC-6 original test corpus of 30 texts (the F-measures of many systems that participated in the contest were higher than 90% and the winner reached 96.42% in F-measure). However it is worth noting that our system was not specifically trained on MUC-6 corpus texts or any other WSJ articles. We also did not make any deliberate changes in our model (apart from the technical ones described above) that could artificially improve performance on this particular set of texts. It would be correct to assume that our system was put in position of a statistical entity extractor trained on a completely different corpus.

Error analysis demonstrated that approximately 60% of errors were the errors of the Compreno parser, 20% occurred due to flaws in the extraction rules and the MUC-6 corpus inconsistencies accounted for the remaining 20%. These results show that the system has a significant potential for further development, especially since there are hopes to improve the quality of the syntactic-semantic parser.

After testing our system on the MUC-6 corpus we also conducted additional tests on the CoNLL corpus [8]. During these tests no settings were modified and no changes were made whatsoever. The resulting F-measure was 0.75. This allows us to make a preliminary conclusion that our system is more resistant to the replacement of one corpus with another than systems based on machine-learning approaches. In the near future we intend to conduct a more extensive performance evaluation on several other corpora.

We do realize that the tests we conducted are insufficient to provide complete evaluation of the system performance, especially since the spectrum of its applications is much wider than named entity recognition. This evaluation was the first and relatively easy step, which brought a lot of valuable information about the system as a whole even though we were only assessing one particular subtask. Later this year we intend to get some results that demonstrate the quality of fact extraction by our system (we are currently in the process of annotating a test corpus).

Conclusion

In this paper we described an information extraction mechanism based on a production rule system. The rules are applied to the results of full syntactic-semantic analysis performed by the Compreno parser. The output of the extraction mechanism is an RDF graph consistent with domain ontology and augmented with information about objects' annotation (markup).

We also presented the idea of storing the extracted information as a set of dynamic logical statements. We described two types of declarative extraction rules: interpretation rules that interpret subtrees of syntactic-semantic trees and identification rules that merge information objects. We gave schematic description of the information extraction algorithm.

A considerable advantage of the system we have created is that a developer of rules does not have to set the order of their execution. Rules are executed in arbitrary order if there is data that matches their left-hand sides. However, if the necessity appears, a developer can set partial rule order.

In our descriptions we attempted to make the reader familiar not only with the core structure of the mechanism, but also with particular solutions which help us address common problems of information extraction from natural language texts.

Finally, we presented the results of the evaluation tests we conducted on the MUC-6 manually annotated corpus. Our system demonstrated relatively good performance with no prior adjustments made. Additional tests on the CoNLL corpus allow us to make a preliminary conclusion that our system is not dependent on a particular corpus (like statistical ones often are) and remains efficient after the corpus is changed. To confirm this conclusion further tests are required and we plan to conduct them in the nearest future. After these tests are performed we intend to publish a new article focusing on the task of fact extraction.

References

1. *Anisimovich K. V., Druzhkin K. Ju., Minlos F. R., Petrova M. A., Selegey V. P., Zuev K. A.* (2012), Syntactic and semantic parser based on ABBYY Compreno linguistic technologies, Computational Linguistics and Intellectual Technologies: Proceedings of the International Conference "Dialog" [Komp'iuternaia Lingvistika i Intellektual'nye Tehnologii: Trudy Mezhdunarodnoj Konferentsii "Dialog"], Bekasovo, pp. 90–103.
2. *Chomsky N.* (1955), Logical Structure of Linguistic Theory, MIT Humanities Library, Microfilm.
3. Defining N-ary Relations on the Semantic Web, available at http://www.w3.org/TR/swbp-n-aryRelations
4. *Fillmore Ch. J.* (1968), The Case for Case, Universals in Linguistic Theory edited by Emmon Bach and Robert T. Harms, Holt, Rinehart and Winston, New York, pp. 1–88.
5. *Gavrilova T. A., Khoroshevskij V. F.* (2000) Knowledge Bases of Intellectual Systems [Bazy znanij intellektualnyh system], Piter, St. Petersburg, Russia
6. *Grishman R., Sundheim B.* (1996), Message Understanding Conference—6: A Brief History, available at: http://acl.ldc.upenn.edu/C/C96/C96-1079.pdf.
7. ISO/IEC 16262:2011, Information technology—Programming languages, their environments and system software interfaces—ECMAScript language specification.
8. *Karasev V., Khoroshevsky V., Shafirin A.* (2004), New Flexible KRL JAPE+: Development & Implementation, Knowledge-Based Software Engineering. Proceedings of the Sixth Joint Conference on Knowledge-Based Software Engineering, Amsterdam.Language-Independent Named Entity Recognition, available at http://www.cnts.ua.ac.be/conll2003/ner/
9. *Mel'chuk I. A.* (1995), Russian Language in the model «Meaning <=> Text» [Russkij jazyk v modeli «Smysl <=> Tekst»], Shkola "Jazyki russkoj kul'tury" Moscow-Vienna.
10. OWL Web Ontology Language Overview, available at http://www.w3.org/TR/2004/REC-owl-features-20040210
11. *Pospelov D. A.* (1989) Modelling Reasoning. Experience in the Analysis of Mental Acts[Modelirovanije Rassuzhdenij. Opyt Analiza Myslitel'nyh Aktov]. Radio i Svayz, Moscow, Russia.
12. Resource Description Framework, available at http://www.w3.org/RDF/
13. *Sundheim B.* (1996), Overview of results of the MUC-6 evaluation, available at: http://aclweb.org/anthology/M/M95/M95-1002.pdf
14. *Zuev K. A., Indenbom M. E., Judina M. V.* (2013), Statistical machine translation with linguistic language model, Computational Linguistics and Intellectual Technologies: Proceedings of the International Conference "Dialog" [Komp'iuternaia Lingvistika i Intellektual'nye Tehnologii: Trudy Mezhdunarodnoj Konferentsii "Dialog"], Bekasovo, vol. 2, pp. 164–172.