StruAP: A Tool for Bundling Linguistic Trees through Structure-based Abstract Pattern

Kohsuke Yanai, Misa Sato, Toshihiko Yanase, Kenzo Kurotsuchi,
Yuta Koreeda, and **Yoshiki Niwa**
Research & Development Group, Hitachi, Ltd.

Abstract

We present a tool for developing tree structure patterns that makes it easy to define the relations among textual phrases and create a search index for these newly defined relations. By using the proposed tool, users develop tree structure patterns through abstracting syntax trees. The tool features (1) intuitive pattern syntax, (2) unique functions such as recursive call of patterns and lexicon reference, and (3) whole workflow support for relation development and validation. We report the current implementation of the tool and its effectiveness.

1 Introduction

This paper describes a tool that helps users semantically bundle linguistic trees such as a constituency tree and a dependency tree. We refer to the tool as StruAP (Structure-based Absract Pattern). By using the proposed tool, the user can easily define relations that are specific to a given business use case and create a search index for the newly defined relations. The search index allows the user to retrieve sentences that include the defined relations. For instance, we can interpret the following sentence as including a *spin-off* relation between *Japanese electronics maker Hitachi* and *home appliance and industrial equipment divisions*.

> *Japanese electronics maker Hitachi will spin off its home appliance and industrial equipment divisions by April to become quicker in decision-making to respond to market changes.* [1]

[1] © 1994–2010 The Associated Press, 2001/9

If we define a *spin-off* relation and extract textual phrases consisting of the relation from large amounts of documents, we can investigate which companies spin off a given business segment, such as *home appliance*, by using standard information retrieval techniques.

By using the proposed tool StruAP, users develop tree structure patterns of the relations through abstracting syntax trees. We assume that in most practical use cases, newly defining relations specific to the use case and developing the corresponding relation extraction modules, instead of use of a universal relation taxonomy and a general extraction algorithm, are required. For example, in a use case of investment decisions, more than ten relations, such as “acquire”, “sue”, and “penalize”, are important to investigate companies. However, it is difficult to develop and maintain a relation extraction module based on linguistic trees because implementation of the logics for traversing trees tends to be complicated. Thus, a tool to help develop and maintain structural patterns of relations would be useful.

The proposed tool is related to semantic role labelling. There are several tools available (Punyakanok et al., 2008; Collobert et al., 2011; Kshirsagar et al., 2015) . However, these tools implicitly assume a kind of general relation taxonomy. When adding new relations in these tools, users need to prepare a certain amount of training data for each relation. The proposed tool aims to help the user to newly define relations for each use case and develop extraction modules within several hours.

The features of the proposed tool are as follows:

1. Intuitive pattern syntax, which is an abstracted representation of outputs of syntax parsers,
2. Unique functions such as recursive call of

patterns and lexicon reference for word level pattern matching,

3. Whole workflow support for relation development and validation.

This paper is structured as follows. We describe related work in Section 2 and the proposed tool in Section 3. We explain the implementation of the tool in Section 4. Section 5 discusses the effectiveness of the tool and Section 6 concludes this paper.

2 Related Work

Tgrep2 is a grep-like tool for tree expressions (Rohde, 2005). The tool allows users to search tree expressions with a given tree query. The expressivity of Tgrep2 has been expanded and a tree query tool Tregex has been developed (Levy and Andrew, 2006). Besides these tools, several tools for tree queries are already available, such as TIGERSearch (Brants et al., 2002), NiteQL (Heid et al., 2004), LPath+ (Lai and Bird, 2005), and a query language for threaded trees (Singh, 2012). However, all these tools are for grep-like purposes but do not support the whole workflow of the relation development. Although Odin's Runes (Valenzuela-Escárcega et al., 2016) provides an information extraction framework, it does not cover the whole workflow. In addition, their pattern languages are path-based and difficult to intuitively understand. For example, a Tregex tree query for the *spin-off* relation can be written as follows:

```
(VP<(VB<<#spin)<(PRT(RP<<#off)))
  $--MD>VP$--NP>S
```

Please refer to (Levy and Andrew, 2006) for the detailed pattern syntax of Tregex. The above pattern expression is different from that of the usual parser outputs. Our proposed tool allows users to easily create a structure-based pattern by directly editing a constituency tree or a dependency tree generated by syntax parsers.

3 Proposed Model

Table 1 illustrates the concept of our proposed model. First, we get a syntax tree by using a parser, such as Stanford's CoreNLP (Manning et al., 2014). The left column shows the syntax tree obtained by parsing the sample text shown in Section 1. We change the expression a little from the original output of Stanford's CoreNLP by adding `lemma=`, `POS=`, and so on. The bold face parts indicate important structures to represent the *spin-off* relation. Thus, we can obtain the generalized structure of the *spin-off* relation by removing non-essential subtrees and lemma information. The right column shows an example of generalized structure. This paper refers to the generalized structure as a structure-based pattern, in contrast to a path-based pattern. Here, `*` means a repeat of any subtree, while `#a[0-2]` means target subtrees to be extracted as elements of the relation. Table 2 shows examples of extracted relations by applying the pattern shown in the right column of Table 1. From a computational point of view, the proposed tool bundles multiple tree instances based on abstract structure, and gives an index for retrieval of the tree instances. In the context of NLP (natural language processing), the main application is relation extraction.

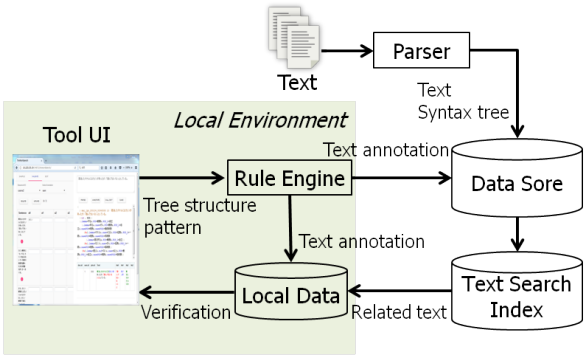

Figure 1: Architecture

3.1 Architecture and Workflow

Figure 1 describes the architecture of the proposed tool. We parse texts that are utilized for a target application and store the generated linguistic trees in a data store in advance. Then, we prepare a text search index for full text search, which is also used later to retrieve relations extracted by the tool. The data store and text search index are running on the server side. If users want to, they can use the proposed tool in only a local environment without server side settings.

Figure 2 shows snapshots of the web-based user interface of the proposed tool. There are 4 tabs: SAMPLE, VALIDATE, EDIT, and INDEX. The procedure to develop a pattern of the *spin-off* relation is as follows:

(.POS=ROOT&.lemma=will **(.POS=S**&.lemma=will **(.POS=NP**&.lemma=Hitachi (.POS=JJ&.lemma=japanese _) (.POS=NNS&.lemma=electronics _) (.POS=NN&.lemma=maker _) (.POS=NNP&.lemma=Hitachi _)) **(.POS=VP**&.lemma=will **(.POS=MD**&.lemma=will _) **(.POS=VP**&.lemma=spin **(.POS=VB&.lemma=spin _)** **(.POS=PRT**&.lemma=off **(.POS=RP&.lemma=off _))** **(.POS=NP**&.lemma=division (.POS=PRP$&.lemma=its _) (.POS=NN&.lemma=home _) (.POS=NN&.lemma=appliance _) (.POS=CC&.lemma=and _) (.POS=JJ&.lemma=industrial _) (.POS=NN&.lemma=equipment _) (.POS=NNS&.lemma=division _)) (.POS=PP&.lemma=by ...	(.POS=S * (#a1.POS=NP *) (.POS=VP (*.POS=MD _) * (.POS=VP (#a0.POS=VB&.lemma=spin _) (.POS=PRT (.POS=RP&.lemma=off _)) (#a2.POS=NP *) *) *) *)

Table 1: Syntax tree (left) and structure-based pattern (right).

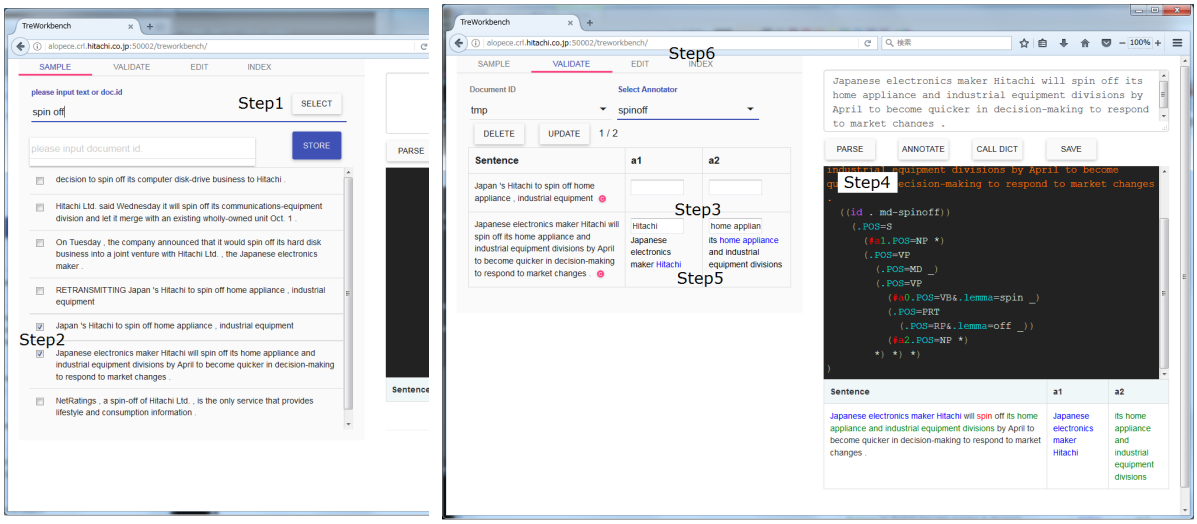

Figure 2: Snapshots of StruAP.

Step 1 In the SAMPLE tab, find several texts of a basis of a tree structure pattern by keyword search. The keyword "spin off" draws relevant sentences from the data store by using the text search index.

Step 2 Choose proper sentences and click the STORE button to save the sentences into the local data store.

Step 3 In the VALIDATE tab, prepare a validation dataset before developing a tree structure pattern. In Figure 2, we decode *Hitachi* for a1 as a subject phrase, and *home appliance* for a2 as a object phrase. Fill out the a[1-2] text fields with the corresponding words.

Step 4 Click the "c" button to parse the sentence and generate its syntax tree in the text area of the right pane. Change the syntax tree to a tree structure pattern through abstraction as shown in Table 1, and click the ANNOTATE button (or Ctrl-Shift-o) to confirm if the pattern works well. A user can see the respective phrases of a[1-2] in the table area below. Although no syntactic error correction is available, frequent use of the shortcut key is helpful to avoid syntactic errors.

Step 5 Click the SAVE button to add the edited pattern into the corresponding pattern file. The rule engine generates text annotations by applying the tree structure pattern, and writes the annotations into the local data store. In the VALIDATE tab, the user can check the differences of a[1-2] between the validation dataset written in Step 3 and the generated

annotations. The differences are highlighted in red.

Step 6 Click the UPDATE button in the INDEX tab to write the annotations for all of the stored documents into the data store on the server side, and update the text search index for extracted relations.

By following the above steps, a text search index for relations is ready for an external application. The CALL DICT button and EDIT tab are related to the lexicon reference explained in Section 3.2.

3.2 Detailed Pattern Specifications

Table 3 lists a sample of pattern expressions of the proposed model. Our model permits leaf node match (_), alternatives (|), subtree retrieval (#), negation (!), and quantification (*, ?, +). In addition, the user can use two unique functions: lexicon reference and subpattern call.

Lexicon Reference The following pattern is a simple one representing a causal relation between subtree a1 and subtree a2.

```
(.POS=S *
 (#a1.POS=NP *)
 (.POS=VP * (.lemma=increase|cause _)
  (#a2.POS=NP) *) *)
```

Here, the pattern `(.lemma=increase|...` would be long because many words can be used as a lemma for the leaf node. In this case, users can use word lists described in separate files, instead of writing down all of the words in the pattern subtree. A pattern using lexicon reference is as follows:

```
(.POS=S *
 (#a1.POS=NP *)
 (.POS=VP * (.lemma=\dic.affect _)
  (#a2.POS=NP) *) *)
```

The `affect` represents the name of a word list containing the words *increase* and *cause*. Users can also define a list of words for a1 and a2.

Subpattern Call It is useful to partially refer to pre-defined subpatterns because the same subpatterns occur repeatedly in different tree structure patterns. The following is an example of a subpattern definition.

```
(
 ((ref . (leaf pos)))
 (.POS=VP *
  (.POS=\arg.pos&.lemma=\dic.spinoff *)
  (#a2.POS=NP *) *)
)
```

Here, `ref` is a reserved word meaning subpattern definition, `leaf` is the name of the subpattern, `pos` is the name of the argument variable. `\arg.pos` is expanded to a given argument. The user can call the subpattern as follows:

```
(.POS=S *
 (#a1.POS=NP *)
 (.POS=VP *
  (\ref.leaf VB|VBZ) *) *)
```

Here, `\ref.leaf` means the call of the subpattern whose name is `leaf`. `VB|VBZ` is the argument of the subpattern and is substituted for `\arg.pos` in the subpattern. When creating a pattern for a sentence with a complicated syntax structure, the recursive call of a subpattern is useful. An example of a recursive call is as follows:

```
(
 ((ref . (lib.vp_loop leaf_pattern pos)))
 (.POS=VP *
  (\ref.vp_loop \arg.leaf_pattern VB) *)
)
(
 ((ref . (lib.vp_loop leaf_pattern pos)))
 (\ref.\arg.leaf_pattern \arg.pos)
)

(.POS=S *
 (#a1.POS=NP *)
 (\ref.vp_loop leaf VB|VBZ)
 *)
```

This pattern partially corresponds to the pattern syntax `A .+(VP) B` of Tregex.

4 Implementation

The proposed tool StruAP is implemented in Python. Syntax trees are expanded on python data structures and tree structure patterns are directly applied on the data structure. This means we do not use translation from our model to existing query lauguage, such as SQL. In the current implementation, we use Cassandra[2] as a data store and Solr[3] as a text search index. We also use Hadoop[4] for parallel processing of the rule engine. While the tool comes with the web-based user interface shown in Figure 2, command line tools for the Linux environment are also available. Users can easily start up the web version of the tool by using a docker image [5] with several configurations for the data store and the text search index. For the command line version, users edit files for tree patterns and lexicon reference with their

[2] Cassandra, http://cassandra.apache.org
[3] Apache Solr, http://lucene.apache.org/solr
[4] Hadoop, http://hadoop.apache.org
[5] docker, https://www.docker.com/

a1	a2	text
Pepsi	the restaurant unit	Pepsi will spin off the restaurant unit by the end of the year .
Mannesmann	its Internet business	The chairman also said Mannesmann might spin off its Internet business .
Rhone-Poulenc	part of its chemicals activities	As part of the plan Rhone-Poulenc would spin off part of its chemicals activities to focus on healthcare , in a bid to boost its share price and earnings .

Table 2: Examples of extracted relations. ©1994–2010 New York Times.

Pattern	Subtree to be matched
`(.POS=PRT *)`	Subtree whose POS is PRT.
`(.POS=VB|VBD _)`	Leaf node whose POS is VB or VBD.
`(.POS=VB&.lemma=spin _)`	Leaf node POS of which is VB, and the lemma of which is *spin*.
`(#a0.POS=VB _)`	Leaf node whose POS is VB. Retrieve the matched node with the name of a0.
`(.POS=NP&.type!=tmod *)`	Subtree POS of which is NP, the dependency type of which is not *tmod*.
`*`	Zero or more occurrences of any subtree.
`(*.POS!=NP|NN|NNS *)`	Zero or more occurrences of the subtree whose POS is not NP, NN, nor NNS.
`(?.POS=MD|CC|VP *)`	Zero or one occurrences of the subtree whose POS is MD, CC or VP.
`(+.POS=NN|NP *)`	One or more occurrences of the subtree whose POS is NN or NP.

Table 3: Sample of pattern expressions.

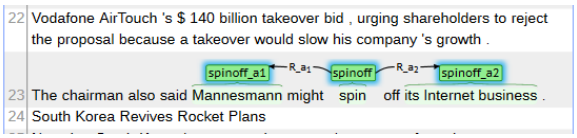

Figure 3: brat view of generated annotations.

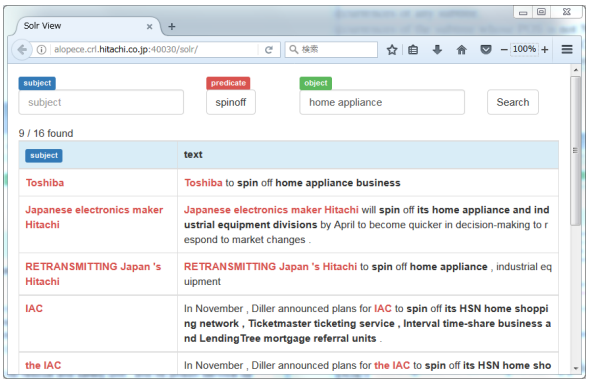

Figure 4: Example of application using extracted relations.

favorite editors and run the rule engine from the Linux command line. We implemented the tool for the English language and Japanese language. Stanford's CoreNLP (Manning et al., 2014) and CaboCha (Kudo and Matsumoto, 2002) are used for the English language and Japanese language, respectively. For the English version, the user can use both constituent trees and dependency trees as the basis of the tree structure patterns. On the other hand, only dependency trees are available for the Japanese version because CaboCha does not output constituent trees.

We can easily integrate the annotation tool brat [6] into the proposed tool via the data store. Figure 3 visualizes the generated annotations in the data store with brat view.

Figure 4 shows a simple example of an application that uses extracted relations. This application uses the same text search index as that of the proposed tool. For example, we can retrieve sentences that include the *spin-off* relation with the object phrase of *home appliance* by entering *home appliance* in the object phrase text field. The highlighted text fragments correspond to the subject phrases. By using the same search index, the tool can be used to directly maintain and update the relation data for the application. Similarly, we can implement various applications using case-specific relations through the index of the proposed tool.

[6] brat, http://brat.nlplab.org

5 Discussion

We roughly investigated the efficiency of the proposed tool in an actual relation development task. Table 4 shows the number of tree structure patterns and the lines of the word lists for lexicon reference, which are developed by two non-researchers in an hour. Although developing tree structure patterns is generally difficult, more than 54 patterns were developed within an hour. We consider that the proposed tool is easy to use and very effectively identifies essential structures for target relations because the tool provides intuitive pattern

relation	Developer A	Developer B
spin-off		
# of patterns	8	12
# of words	3	2
sue		
# of patterns	7	10
# of words	5	4
penalize		
# of patterns	8	9
# of words	8	4

Table 4: Efficiency investigation. Developer A is translator, but not researcher; Developer B is software engineer, but not researcher.

syntax and whole workflow support. The use of a bracketed syntax to define patterns is arguable. However, we suppose users can develop lots of patterns faster by editting bracketed patterns in a text-based editor, than by use of a graphical tree editor.

On the other hand, there are not enough words for lexicon reference for an actual application. Although the function of the lexicon reference is useful, the current tool is not helpful to increase the variety of predicate words and clue words.

We have already used the tool in several cases. We have developed 657 patterns in total, and the number of words in the lexicons used for the patterns is 6406. Especially, our development of end-to-end argument generation system in debating (Sato et al., 2015) relied on the proposed tool. We conclude that the proposed model is effective when it is necessary to newly define case-specific relations and non-researchers are involved in development of the relation extraction.

6 Conclusion

This paper describes a tool for developing tree structure patterns, which makes it easy to define relations among textual phrases, and creates a search index for these newly defined relations. The tool assumes that in most practical use cases, newly defining relations specific to the use cases is required. In such cases, the proposed tool is effective to identify essential tree structure patterns.

Future work includes developing semi-automatic abstraction using frequent subtree mining and integrating techniques to collect clue words for lexicon reference. Another direction is to support collaboration between users. Similar subpattern search helps users to avoid duplicating a pattern created by another user.

References

Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories*.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537.

Ulrich Heid, Holger Voormann, Jan-Torsten Milde, Ulrike Gut, Katrin Erk, and Sebastian Padó. 2004. Querying both time-aligned and hierarchical corpora with NXT search. In *Proceedings of LREC-2004*, pages 1455 – 1459.

Meghana Kshirsagar, Sam Thomson, Nathan Schneider, Jaime Carbonell, Noah A. Smith, and Chris Dyer. 2015. Frame-semantic role labeling with heterogeneous annotations. In *Proceedings of ACL-IJCNLP 2015 Short Papers*, pages 218–224.

Taku Kudo and Yuji Matsumoto. 2002. Japanese dependency analysis using cascaded chunking. In *Proceedings of CoNLL 2002 Post-Conference Workshops*, pages 63–69.

Catherine Lai and Steven Bird. 2005. Lpath+: A first-order complete language for linguistic tree query. In *Proceedings of the 19st Pacific Asia Conference on Language, Information and Computation, PACLIC*.

R. Levy and G. Andrew. 2006. Tregex and tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of LREC-2006*.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *ACL System Demonstrations*, pages 55–60.

Vasin Punyakanok, Dan Roth, and Wen-tau Yih. 2008. The importance of syntactic parsing and inference in semantic role labeling. *Comput. Linguist.*, 34(2):257–287.

Douglas L. T. Rohde. 2005. Tgrep2 user manual version 1.15.

Misa Sato, Kohsuke Yanai, Toshinori Miyoshi, Toshihiko Yanase, Makoto Iwayama, Qinghua Sun, and Yoshiki Niwa. 2015. End-to-end argument generation system in debating. In *Proceedings of ACL-IJCNLP 2015 System Demonstrations*.

Anil Kumar Singh. 2012. A concise query language with search and transform operations for corpora with multiple levels of annotation. In *Proceedings of LREC-2012*.

Marco A. Valenzuela-Escárcega, Gus Hahn-Powell, and Mihai Surdeanu. 2016. Odin's runes: A rule language for information extraction. In *Proceedings of LREC-2016*.