Building a Process Description Repository with Knowledge Acquisition

Diyin Zhou, Hye-Young Paik(✉), Seung Hwan Ryu, John Shepherd, and Paul Compton

School of Computer Science and Engineering,
University of New South Wales, Sydney, Australia
{dzho186,hpaik,seungr,jas,compton}@cse.unsw.edu.au

Abstract. Although there is an abundance of how-to guides online, systematically utilising the collective knowledge represented in such guides has been limited. This is primarily due to how-to guides (effectively, informal process descriptions) being expressed in natural language, which complicates the process of extracting actions and data. This paper describes the use of Ripple-Down Rules (RDR) over the Stanford NLP toolkit to improve the extraction of actions and data from process descriptions in text documents. Using RDR, we can incrementally and rapidly build rules to refine the performance of the underlying extraction system. Although RDR has been widely applied, it has not so far been used with NLP phrase structure representations. We show, through implementation and evaluation, how the use of action-data extraction rules and knowledge acquisition in RDR is both feasible and effective.

1 Introduction

In daily life, people often undertake processes[1] whose intrinsic details are unknown because they are being encountered for the first time. Such processes could be as simple as cooking a dish or as complex as buying a house. To distinguish them from the conventional (and more widely-studied) organisational workflows, we refer to these kinds of processes as "*personal processes*". Descriptions of personal processes are often shared on-line in the form of how-to guides (e.g., on-line recipe sites, eHow[2], WikiHow[3]), which are normally written in natural language as a set of step-by-step instructions.

Although the collective knowledge and data represented in such descriptions could be useful in many applications, the potential for using them is curbed by their natural language format. The ultimate goals of our project [3] are (i) to build a repository of personal process descriptions based on a formal model and (ii) to investigate novel query techniques and data analytics on such descriptions.

[1] In this paper, we use the terms process and workflow interchangeably.
[2] www.ehow.com, eHow.
[3] www.wikihow.com, WikiHow.

H. Ohwada and K. Yoshida (Eds.): PKAW 2016, LNAI 9806, pp. 86–101, 2016.
DOI: 10.1007/978-3-319-42706-5_7

In this paper, as a step towards building the repository, we describe *RDR-ADE* (RDR Action Data Extractor), which aims to extract the basic constructs of a process description from how-to guides.

We adopt a knowledge acquisition approach (using Ripple-Down Rules [1]) to allow human users to improve the performance of a standard NLP parser [10] in identifying verbs and objects in personal process descriptions. Although RDR has been used in conjunction with NLP tools in the past, the aim of our approach is to extract basic process constructs (i.e., actions and data) from text, which has not been done with RDR before. Specifically, we make the following contributions.

- We propose the notion of *action-data extraction rules* (henceforth *extraction rules*). These rules represent user knowledge about how the actions and their associated data can be identified from process descriptions.
- We present *incremental knowledge acquisition techniques* to build and update a set of extraction rules. The continuous update of extraction rules enables our system, over time, to make finer-grained extraction decisions.
- We present an implementation of *RDR-ADE* and provide the evaluation results that show the feasibility and effectiveness of our proposed approach.

While the ideas we present should apply to many types of process descriptions, we consider here only cooking recipes due to the availability of large numbers of relatively compact online descriptions[4].

The rest of the paper is structured as follows: In Sect. 2, we discuss the use of RDR in other systems as a way to improve the underlying techniques. We then briefly introduce some background concepts such as a model for personal processes and Stanford CoreNLP. In Sects. 4, 5 and 6 we describe the details of the rule models and implementation, followed by evaluation results and conclusion.

2 Related Work

Ripple Down Rules (RDR) [1] is a knowledge acquisition technique that allows experts to rapidly construct knowledge bases on a case-by-case basis while a system is already in use. It can be built incrementally on top of an existing system to improve the underlying system. RDR and its variants have been successfully used in a wide range of application domains, such as Web document classification [8], diagnosis [9], information extraction [7], and have had significant commercial uptake [2,9]. A wide range of RDR research covering a range of RDR methods is reviewed in [12].

The key idea behind RDR is that cases are processed by the RDR system and when the output provided by the system is not correct, a new rule is added to give the correct output for that case. In this process, the expert does not need to consider the structure of the knowledge base or modify other rules. They build the rule to give the correct conclusion for the case they are trying to correct,

[4] Recipes are also useful for subsequent studies in personal process analysis, but that is beyond the scope of this paper.

and in most RDR systems this rule is then checked against cases for which rules were previously added to see that none of these cases are misclassified. If necessary the expert adds further rule conditions, making the rule more specific for the case in hand excluding the previous cases. The system then automatically adds the new rule into the knowledge base. This is a very rapid process and log data from commercial systems shows that across 10 s of 1000 s of rule and 100 s of knowledge bases, the average time to add a rule was under two minutes [9].

In terms of utilising RDR with NLP tools, [6] shows how the informal writing style of Web documents tends to negatively affect the performance of NER (Named Entity Recogniser) parsers. The paper proposes an RDR-based knowledge acquisition process where the rules are used to correct spelling errors or missing/unclassified named entity (NE) tags (e.g., 'YouTube' and 'YOUTUBE' should be classified as 'ORG' (organisation)). The same authors also applied the same principles to extract *Open Relations* between named entities. Open Relation Extraction systems seek to extract all potential relations from the text rather than extracting pre-defined ones [7]. The only previous work on RDR for parts of speech and phrase representations was an automated learner using the RDR structure, rather than actual knowledge acquisition [11].

Our aim is to not only demonstrate improvement on the NLP Stanford parser, but to improve the extraction of action and data information from the text, which is something that has not been done before with RDR.

3 Preliminaries

In this section, we give the background concepts that are necessary to describe the *RDR-ADE* system: the target schema that *RDR-ADE* works towards and the extraction process of action/data pairs using the Stanford NLP parser.

3.1 PPDG: A Model for Personal Process Descriptions

To formally represent and analyse the process instructions in how-to guides, we proposed a graph-based model, called "Personal Process Description Graph" and a query language over this model [5,14]. PPDGs are labelled directed graphs that include both actions and data. In a PPDG, there are different node types for actions and data, different arc types to represent action flow and data flow, and associations between actions and data. The formal definitions and properties of the model are omitted here as the main topic of this paper does not directly involve PPDGs. Our goal is to extract, from a process description, the information required to define action nodes and data nodes to build a PPDG.

3.2 Action and Data Extraction Process

The Stanford CoreNLP toolkit [10] provides a collection of NLP tools. The tools that are important for our purposes are the part-of-speech (POS) tagger, which

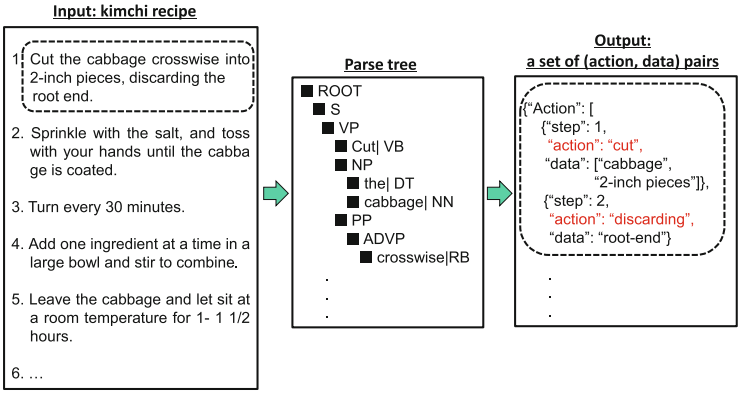


Fig. 1. Example input and output for the *RDR-ADE* system

marks words with their syntactic function, and the parser, which produces a parse tree for a chunk of natural language text.

The overall process of extracting actions and data from a description is shown in Fig. 1. We break this process into the following phases:

1. *Input Text Segmentation.* A process description consists of a number of "input units" where each "input unit" contains one or more sentences and each sentence contains one or more phrases. Each phrase corresponds to a single step in the process. Input units are separated by line breaks. The parser partitions each input unit into sentences and phrases.
2. *Parse Tree Generation.* Using the Standford NLP parser, this phase takes an input unit and produces as output a parse tree. As in Fig. 1, the non-leaf nodes of the parse tree carry the phrase structure (via tags such as NP (Noun Phrase) or VP (Verb Phrase)), while leaf nodes carry the part-of-speech (POS) for individual words (via tags such as NN (Noun), VB (Verb)).
3. *Action and Data Extraction.* In this phase, we extract a set of (action, data) pairs from the parse trees generated from the previous phase. In identifying each pair, we use the phrase structure and consider a word with VB (and derivations of it) as an action and NN as a data item. The extracted pairs are written as JSON objects for further processing (e.g., mapping to a PPDG).

The phrase structure contained in a parse tree is useful for identifying the most likely places from which to find verbs and objects. In each sentence, the parse tree tags a verb phrase (VP) and within a verb phrase, we may find a relevant noun phrase (NP).

While the NLP toolkit is useful for our purpose, it has limitations for the kind of informal text that we typically find in process descriptions.

For example, the default POS tagger and parser are trained to assume that the first phrase in a sentence will be a noun phrase referring to the sentence subject. However, recipe sentences generally start with the verb, and have an implicit subject ("you"). This results in common types of recipe sentences (e.g. "smoke

the salmon") being misinterpreted as starting with a noun (e.g. "smoke") if the first word can be interpreted as a noun. Another type of sentence which caused problems for the parser was one in which an adverbial phrase was placed at the start of the sentence (e.g. "with a sharp knife, chop the cabbage" vs "chop the cabbage with a sharp knife").

As we have briefly outlined in Sect. 2, instead of re-training the CoreNLP tools, we employ a similar approach to [6,7]. We exploit the RDR technique to build and update action-data extraction rules over the baseline system (i.e., CoreNLP tools). These rules are designed to address the said problems. The continuous updates of extraction rules enable our system to perform more precise extraction.

4 Harnessing User Knowledge

In this section, we describe the notion of *extraction rules* and their management. As noted above, previous RDR techniques on NLP tools were targeted on extracting named entities or open relations - utilising POS tags. The knowledge we need to represent in *RDR-ADE* has to be expressed over the parse tree structure. The rule model we present below is designed to express user knowledge about which situations the actions and their relevant data can be identified in a given parse tree.

4.1 Extraction Rule Representation Model

Each rule has two components: a condition and a conclusion. The conclusion part of the rule simply states how a word is to be labelled as 'action' or 'data', or left unlabelled. A condition consists of a conjunction of predicates on parse trees. To express a condition relating to the parse tree, we propose the following rule syntax components: nodes, test values and operators.

Nodes: To express conditions over nodes in the parse tree, we provide intuitive access names for the nodes (following the XML document model). Some of the examples of the possible names are: `currentNode`, `parentNode`, `allAncestors`, $\texttt{x}^{th}\texttt{LevelParentNode}$, `firstChild`, `lastChild`, `nextSibling`, `prevSibling`, etc.

Test Values: The test values can be of two types: Tags or Regular Expressions. Tags represent the parse tree tag values that a node could be associated with,

Table 1. A sample list of tags

PT*	Description	WT+	Description	WT+	Description
ADVP	Adverb phrase	CD	Cardinal number	NN, NNS	Noun, plural
NP	Noun phrase	DT	Determiner	VB, VBG	Verb, gerund
PP	Prepositional phrase	IN	Preposition	VBN	Verb past participle
VP	Verb phrase	JJ	Adjective	PRP$	Possessive pronoun

*PT = Phrase Level Tags, +WT = POS Word Tags.

such as VP, NN, and DT. Table 1 describes some of the phrase structure tags and part-of-speech tags[5] used in our system. Besides the standard tags, we have our own custom tags: 'ACTION' and 'DATA'.

A test value could also contain a regular expression. This is useful, for example, when the user wants to match any POS tags that are derivations of a base form (e.g., VBD, VBG, VBN, VBP, VBZ are derivations of VB). Note that a regular expression could also include a literal value (e.g., 'Oil').

Operators: The set of operators we currently support allows for a given node to be tested against certain properties. For example, we could test if a node is a verb phrase, or has a text value of 'X'. The design and implementation of these operators is at the heart of the rule design. Our current implementation supports the following operations[6]:

- `HasPhraseTag:` returns true if the node tested has a phrase tag give in the test value, e.g., PP (Prepositional phrase), VP (Verb phrase).
- `HasWordTag:` returns true if the node tested has a part-of-speech tag give in the test value, e.g., DT (Determiner), CD (Cardinal number), VB (Verb). We use the term word tags for part-of-speech tags.
- `HasActionObjectTag:` returns true if the node tested is labelled with the our custom tags "Action" or "Data".
- `IsLeafNode:` returns true if the node tested is a leaf node in the parse tree.
- `HasText:` returns true if the node tested has the text given in the test value.
- `CanBeOfWordType:` returns true if the text value of the node tested is of the word type given in the test value, e.g., (NN) Noun, (VB) Verb, or (JJ) Adjective.

We use the WordNet API[7] to implement the `CanBeOfWordType()` operator. This operator is used to see if a given word could have different functions in a sentence. For example, 'oil' could be a noun (as in 'olive oil') or a verb (as in 'oil the fish').

Using these components, we define an extraction rule as follows:

Definition 1 ***(Extraction Rule).*** *Let N_t be a set of nodes in a parse tree t. An extraction rule has the form:* `IF` P `THEN` C *where P is a conjunction of predicates on tree nodes and test values, and C is a conclusion. Each predicate has the form $(node, op, val)$, in which $node \in N_t$, $op \in \{HasPhraseTag, IsLeafNode, ...\}$, and $val \in \{VP, NN, VB, NP, ...\}$. The conclusion has the form $(node, action/data)$.*

For example, the rule `(currentNode,HasWordTag,NN)→(currentNode, 'Data')` checks whether the current tree node has a tag of `NN` and then determines that it must be a data item. If the node was `cabbage|NN`, the conclusion would be that "cabbage" was data for the current action.

[5] For the complete set of part-of-speech tags generated by the Standford parser, see http://www.comp.leeds.ac.uk/amalgam/tagsets/upenn.html.

[6] However, adding a new operator is a straightforward task in our system.

[7] WordNet 3.0, https://wordnet.princeton.edu.

4.2 Matching Extraction Rules

When a parse tree t is generated from a case, the system identifies two extraction rules where one is for identifying actions and the other is for identifying their associated data. For this, we provide an operation called `MatchRule()`, which takes as input t and produces as output the rules for the action and data identification process. The system matches t against the conditions of a set of extraction rules. It evaluates the rules at the first level of rule tree. Then, the next level of rules are evaluated, if their parent rules are satisfied. If no extraction rule is found to be appropriate to t, the user might build a new rule with the help of rule editor provided by our system.

Algorithm 1. MatchRule

Input: Parse tree t and a set of extraction rules R
Output: A set of matched extraction rules
begin
1: Let *satisfiedRules*:= ϕ;
2: // C is a condition of a rule
3: // p is a predicate of C
4: **foreach** $r \in R$ **do**
5: C:= *getCondition(r)*;
6: *allPredicatesSatisfied*:= true;
7: **foreach** $p \in C$ **do**
8: **if** not *isSatisfiedBy(p, t)* **then**
9: *allPredicatesSatisfied := false*;
11: **endif**
11: **endfor**
12: **if** *allPredicatesSatisfied* **then**
13: *satisfiedRules*:= *satisfiedRules* $\cup$ r;
14: **endif**
15: **endfor**
16: **return** *satisfiedRules*;
end

5 Incremental Knowledge Acquisition

This section presents how to incrementally obtain the extraction rules from users.

5.1 Knowledge Acquisition Method: Ripple Down Rules

To build and update extraction rules, we use the RDR [1] knowledge acquisition method because: (i) it provides a *simple* approach to knowledge acquisition and maintenance; (ii) it works *incrementally*, in that users can start with an empty rule base and gradually add rules while processing new cases.

RDR organizes the extraction rules as a tree. In *RDR-ADE*, we have two rule trees: one for action extraction (Fig. 2(a)), the other for data extraction (Fig. 2(b)). For example, the rule tree for action extraction has *Action_DefaultRule* and *Action_BaseRule*. The exceptions to the base rule are named *AE_Rule1*,

AE_Rule2, ... *AE_RuleX* according to the creation order. *Action_DefaultRule* is the rule that is fired initially for every *case*. The rules underneath it are more specialized rules created by adding exception conditions to their parent rules. The *rule inference* in RDR starts from the root node and traverses the tree, until there are no more children to evaluate. The conditions of nodes are examined via depth-first traversal, which means the traversal result is the rule whose condition is satisfied *last*. The same applies to the rule tree for data extraction. We note that for each case, *RDR-ADE* evaluates both the action extraction and data extraction rule trees to produce the JSON output.

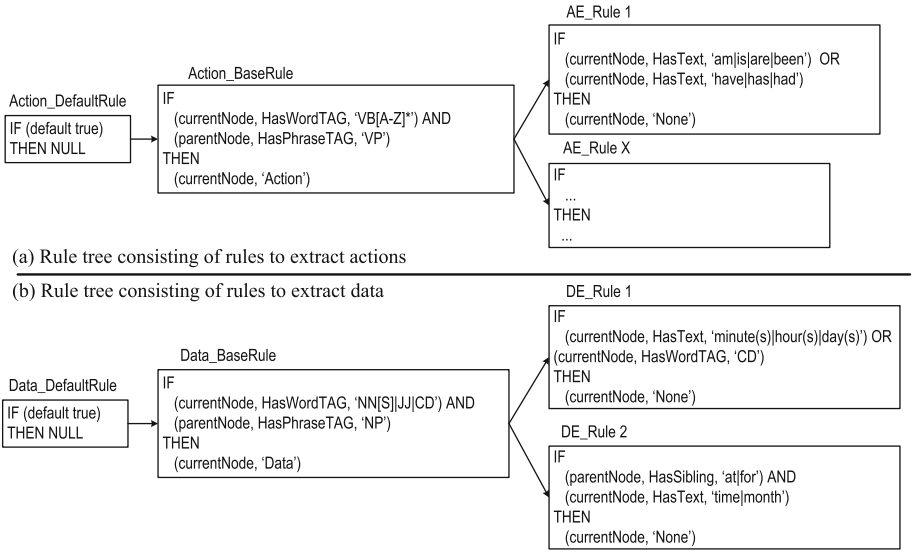


Fig. 2. Example RDR trees (abbreviated)

5.2 Acquiring User Knowledge Incrementally

In what follows, we demonstrate how error-correcting rules are acquired from the user incrementally using a sequence of cases as an example scenario. In the cases, actions are underlined and data is shown in **bold**.

Case 1. **"Cut the cabbage crosswise into 2-inch pieces, discarding the root end".**

From the sentence in Case 1, our system generates the parse tree shown in Fig. 3. At this point, there is one default rule in each rule tree. These rules are applied to this parse tree and NULL values are returned from each rule tree.

The user considers this as an incorrect result and adds new rules *Action_BaseRule* and *Data_BaseRule* under the default rules as shown in Fig. 2.

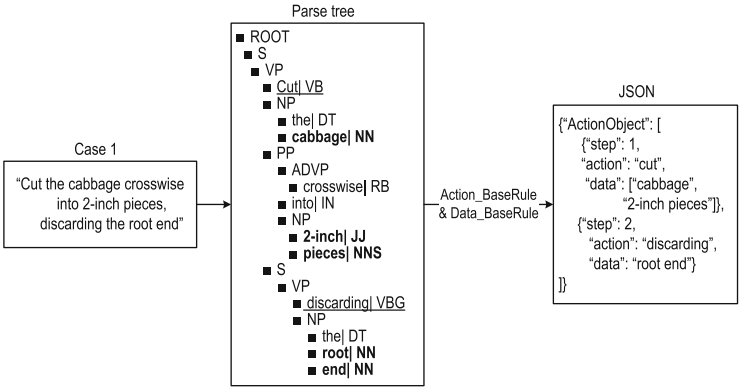


Fig. 3. Case 1 applying two new exception rules

Action_BaseRule specifies that, if a node has a word tag matching a regular expression 'VB[A-Z]*' and its parent node has a phrase tag 'VP', the word is labelled as 'Action'. By applying this rule to the parse tree of Case 1, the system returns a set of actions {cut, discarding}. On the other hand, *Data_BaseRule* states that if a node has a word tag 'NN[S]', 'JJ' or 'CD' and its parent node has a phrase tag 'NP', the word should be labelled as 'Data'. From the parse tree, this rule returns {cabbage, 2-inch pieces, root end}. Figure 3 shows the results of applying these two new rules to the case, which is now considered correct.

In fact, as indicated by their names, we consider these two rules as the base rules in our system for extracting actions and data respectively.

Now we consider the next case, Case 2 whose parse tree is shown in Fig. 4.

Case 2. **"**Sprinkle with **salt**; toss with your **hands** until the cabbage is coated**"**.

Using the parse tree for this case, the two base rules are fired and the system returns as actions {sprinkle, toss, is coated} and as data {salt, hands, cabbage}.

The user considers the results and decides to exclude 'is coated' from the action list. As a general rule in our system, we ignore forms of the verb 'to be' (and sometimes 'to have') when used as an auxiliary together with the past participle of a transitive verb, especially when a word like 'until' is used as a subordinating conjunction to connect another action to a point in time.

To ignore BE-verbs (e.g. am, are, is, been, ...) and HAVE-verbs (have, has, had, ...) from the actions, the user adds the following rule as an exception to *Action_BaseRule*:

AE_Rule1: For current node $n \in N_t$, *(n, HasText, 'am,is,are,been') or (n, HasText, 'have,has,had')* $\rightarrow$ *n is not labelled as 'action'.*

According to *AE_Rule1*, if the current node contains either a BE-word or a HAVE-word, the word associated with the node is ignored from action labelling. Thus, from the same parse tree in Fig. 4, the rule matching algorithm now generates the final JSON object by applying *AE_Rule1* and *Data_BaseRule* instead

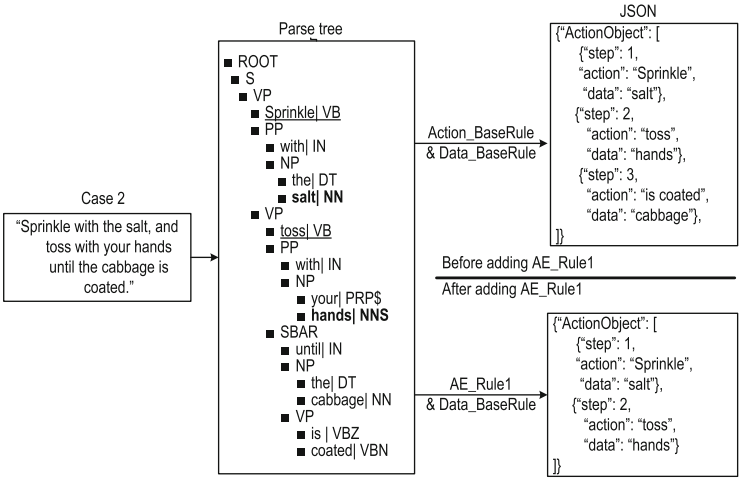


Fig. 4. Case 2 ignoring HAVE-verbs and BE-verbs.

of *Action_BaseRule* and *Data_BaseRule*. Here, we do not extract the `cabbage` as data because its associated verb is not identified as an action.

Case 3. **"**Turn every 30 minutes**"**.

In Case 3, according to the existing rules so far, `30 minutes` is classified as 'Data' (by *Data_BaseRule*). In this scenario, the user decides to ignore numbers or units such as `30 minutes`, `2 days`, `30 cm` and so on, because she considers them as auxiliary information that is certainly useful but not part of the key action/data constructs in a process. She may want to consult the system developer to define a new type of label "UNITS", but for now, she adds a new rule *DE_Rule1* as an exception of *Data_BaseRule*.

DE_Rule1: For the current node $n \in N_t$, *(n, HasText, 'minutes(s), hour(s), day(s)') or (n, HasWordTag, 'CD')* $\rightarrow$ *n is not labelled as 'data'.*

DE_Rule1 states that, if a node has a time-word or has a word tag `CD` or `DT`, then it is not labelled as data. After this rule is defined, in the final JSON object in Fig. 5(a), we extract only the action "turn" from the sentence.

Case 4. **"**Add **one ingredient** at a time in a **large bowl** and stir to combine**"**.

Now consider Case 4. According to the rules so far, *Data_BaseRule* will make `time`|NN, under NP as 'Data'. The user considers that the result is not what she expected, and decides that she does not want to extract data from propositional phrases such as `at a time, in half, for up to one month`, etc. She adds the following rule *DE_Rule2*.

DE_Rule2: For current node $n \in N_t$, *parent node* $pn \in N_t$, *(pn,HasSibling, 'at,for') and (n,HasText, 'time,month')* $\rightarrow$ *n is not labelled as 'data'.*

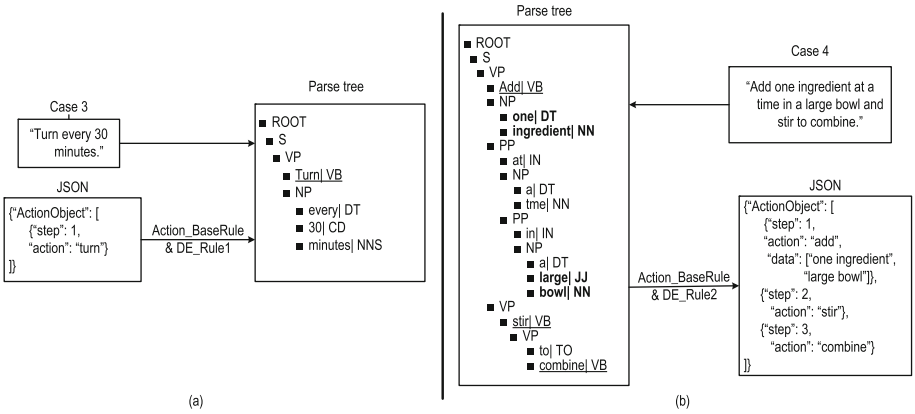


Fig. 5. Case 3 ignoring numbers or units, Case 4 ignoring prepositional phrases.

The rule says if a current node is a time-related word and its parent node has a sibling node tagged as `at` or `for`, then word associated with the node is not labelled 'Data'. The final JSON objects with this rule is shown in Fig. 5(b).

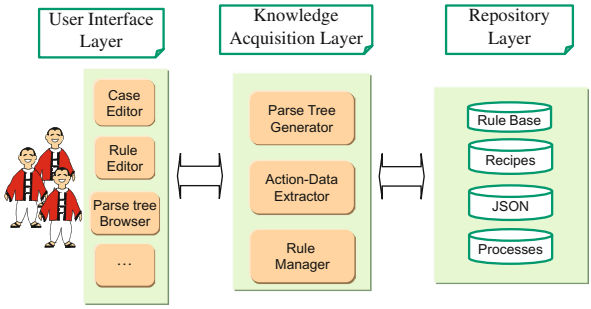


Fig. 6. *RDR-ADE* system architecture

6 Implementation and Evaluation

This section describes our prototype implementation and experimental results.

6.1 Implementation

A prototype was implemented using Java, J2EE and Web services. The RDR engine and action-data extractor are all independent Java programs that are wrapped by a REST web service and accessed through HTTP. This architecture

allows other web pages or applications to make use of the services in other ways. The *RDR-ADE* system consists of the following three layers: user interface, knowledge acquisition, and repository (see Fig. 6).

The *user interface layer* allows users to browse generated parse trees and incrementally build extraction rules using the rule editor. The *knowledge acquisition layer* is responsible for generating parse trees, extracting actions and data, creating rules, etc. The *repository layer* stores the rules, process descriptions (e.g., recipes), JSON objects, and so on. Table 2 shows a set of operations that the components of such layers can invoke to carry out their specific functions. Figure 7 gives a screen-shot of our system. Here, we see the input case on the top left panel. For the input case, the system generates a parse tree in the bottom right panel. Then, using the extraction rules in a knowledge base, it produces a set of (action, data) pairs in the format of a JSON object in the top right panel.

Table 2. The list of operations invoked in *RDR-ADE*

Parse tree generator/Rule manager operations
- *generateParseTree(c)* produces a parse tree from an input case *c*.
- *matchRule(c)* returns a list of extraction rules applicable to an input case *c*.
- *createRule(c,d)* creates a rule with a condition *c* and a conclusion *d*.
- *refineRule(r,c,d)* refines a rule *r* with a condition*c* and a conclusion *d*.
Action and data extractor operations
- *extractActionData(p)* identifies actions and data from a given parse tree *p*.
- *generateJSON(ad)* generates a JSON object from a set of (action, data) pairs *ad*.

6.2 Evaluation

We now present the evaluation results to show how effectively the *RDR-ADE* system identifies actions and their associated data from process descriptions.

Dataset. We use a dataset derived from 30 recipes. The dataset consists of 317 sentences and 4765 words. We have manually labelled the verbs (as 'action') and data items (as 'data') in each sentence to create the ground-truth. Each sentence is uniquely identified with an ID. We then processed sentences one by one in the presented order.

Evaluation Metrics. We measured the overall performance of the extraction system using the following formula.

$$\text{Accuracy} = \frac{\text{the number of correctly identified actions and data items}}{\text{the total number of labelled actions and data items}}$$

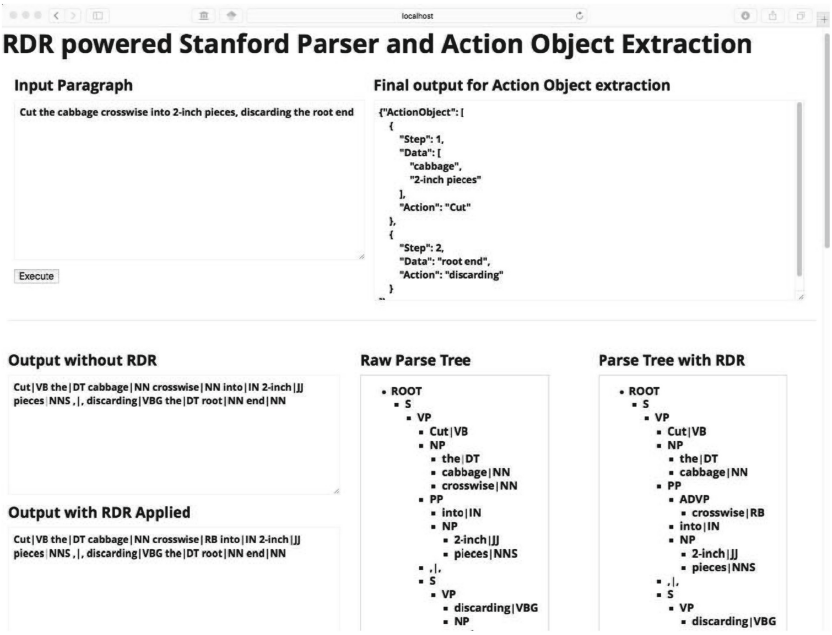


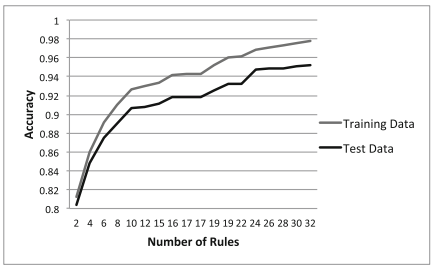
Fig. 7. Screenshot showing: input case, parse tree, and action-data pairs.

Training Phase. Starting with an empty knowledge base, we began the knowledge acquisition process by looking at the sentences one by one in the order prepared at the start of the experiment.

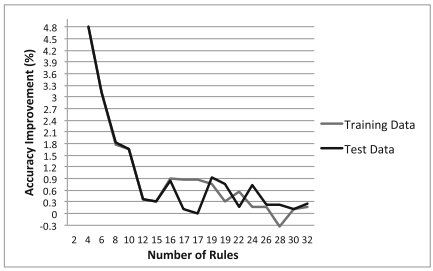
The acquisition process is defined as follows (note that this process is repeated for every sentence): (i) a sentence is given as an input case, (ii) rules are applied, (iii) we examine the result, (iv) if the result is what the user expected, the rule-base is untouched; if not, an exception rule is added to the rule base.

The above steps are repeated until all sentences are considered, or until we do not see significant improvement in the accuracy measure. With an RDR system one can keep adding rules indefinitely as increasingly rare errors are identified. In critical in application areas such as medicine, the ability to keep easily adding rules if required is a key advantage of RDR. In other domains, and in research studies such as this, it is sufficient to add rules until the performance plateaus and adding new rules has a negligible effect on the overall performance.

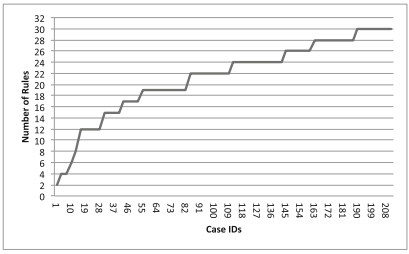
In the first run, we stopped at the 212^{th} case. In the following discussion we have called the initial 212 cases "training data" and used the remaining cases (cases 213-317) as the "test data" (i.e., unseen cases). In fact the initial 212 cases should not really be considered as "training data" until they have been processed by the system and perhaps a rule added. For example in Fig. 8(c), in processing the first 100 cases, 22 errors occurred and a rule was added for each error as it occurred. The remaining 112 cases had not yet been used for training; however, in



(a) Accuracy Results



(b) Accuracy Improvement Rate



(c) Number of Rules Created

Fig. 8. Experiment results

the following discussion of accuracy, for simplicity, all 212 cases eventually used for training are used in assessing accuracy on training data.

Observations:

– Figure 8(a) shows that, during the training phase, the performance of the system improves as more sentences are processed and more rules are added. The performance improves rapidly at the early stage of the training and gradually plateaus. At the end of the training cases, 32 rules had been added and the accuracy was 98 %. The accuracy is not 1.0 because when a new rule is added only the case for which the parent rule is added is checked; however other cases in the training data processed by this rule, might now be incorrect. The results demonstrate that even checking one case per rule provides very robust validation.

  The performance on the test data similarly improves rapidly as more rules are added to deal with training data errors. The accuracy on the test data and training data is very similar when low numbers of rules have been added, because in fact most of the "training data" is as yet unseen; however, as more and more of the training data is actually used for training, the performance on the test data is only slightly less than the training data, 96 % vs 98 %.
– Figure 8(b) shows that a new rule had bigger impact on the performance of the system at the early stage of the experiment. Then, as more and more rules were added their impact tailed off. This is because common and repeated errors are fixed earlier on, leading to substantial improvement in terms of the accuracy measure. The overall trend of the graphs shows that the improvement brought

by each rule converges at a low volume, because the errors left to be fixed at the late stage of the experiment are most likely only applicable to unique and less common cases.
- Figure 8(c) shows how quickly rules are added as the cases are processed. Rules are initially added frequently as exceptions are discovered. Since the exceptions are subsequently re-used to handle new sentences, less new rules are required. Eventually, as the other graphs show, sufficient rules have been added to the system to handle most new sentences.

On Average Time Per Rule Creation: Because creating rules requires understanding of the NLP parse tree structure, we also have looked at the average time taken per rule creation. This was done by asking three users (one expert, two non-experts) to use the system. The expert who was the developer of the system had around 12 hours of experience using the rule editor. The non-experts were aware of the purpose of the system, but not necessarily familiar with the parse tree structure. They had around 2 hours of training on the system. They did not participate in building knowledge bases, but experimented with the system adding around 5-10 rules. The expert reported that it took less than 2 minutes to look at a case and create a rule if needed. For the non-experts, it took around 2 minutes to create a rule.

7 Conclusion

In this paper, we presented an RDR-based knowledge acquisition system to extract action/data pairs from text documents (specifically recipes) that contain process instructions (i.e., step-by-step guides). Although omitted for clarity, our system also provides a rule-based component that improves the POS tagging itself, which in turns improves the action-data extraction process overall. The performance of the system over a test data set showed that even with relatively short training, we could obtain 96 % accuracy on unseen data.

Our immediate future work includes performing more experiments with multiple users, using the same dataset and ground truth. This is not only to show that the results can be repeated by different users, but also to gain deeper insights into the relationships between the number and quality of the rules and the performance improvement of the system. In machine learning research, repeat experiments with different randomisations of training and test data, are used to avoid spurious results due to differences between training and test data. The results here are clearly not due to significant differences between training and test data as the increase in accuracy as cases are processed and rules are added is very similar for both training data (including cases from the 212 not yet processed in training) and test data (cases 213 to 317).

Another important aspect of the work is to design an approach that could assist users with writing the rules over the parse trees, as this requires a high level of understanding of the phrase representations. The email classification system built by [4,13] shows that a purpose-designed rule editor interface that facilitates a simple point-and-click style can help the rule building process. The extensive

evaluation plan we outlined above could help us identify critical and common patterns in the generated rules. Such patterns could be the basis for the design of a more intuitive rule editor.

References

1. Compton, P., Jansen, R.F.: A philosophical basis for knowledge acquisition. Knowl. Acquisition **2**(3), 241–258 (1990)
2. Dani, M., Faruquie, T., Garg, R., Kothari, G., et al.: A knowledge acquisition method for improving data quality in services engagements. In: Services Computing, pp. 346–353. IEEE (2010)
3. Hajimirsadeghi, S.A., Paik, H.-Y., Shepherd, J.: Social-network-based personal processes. In: Bae, J., Suriadi, S., Wen, L. (eds.) AP-BPM 2015. LNBIP, vol. 219, pp. 155–169. Springer, Heidelberg (2015)
4. Ho, V., Wobcke, W., Compton, P.: EMMA: an e-mail management assistant. In: Intelligent Agent Technology, pp. 67–74. IEEE (2003)
5. Hsu, J.O., Paik, H., Zhan, L.: Similarity search over personal process description graph. In: Wang, J., Cellary, W., Wang, D., et al. (eds.) WISE 2015, Part I. LNCS, vol. 9418, pp. 522–538. Springer, Heidelberg (2015). doi:10.1007/978-3-319-26190-4_35
6. Kim, M.H., Compton, P.: Improving the performance of a named entity recognition system with knowledge acquisition. In: ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N. (eds.) EKAW 2012. LNCS, vol. 7603, pp. 97–113. Springer, Heidelberg (2012)
7. Kim, M.H., Compton, P., Kim, Y.S.: RDR-based open IE for the web document. In: K-CAP, pp. 105–112. ACM (2011)
8. Kim, Y.S., Park, S.S., Deards, E., Kang, B.H.: Adaptive web document classification with MCRDR. In: ITCC, vol. 1, pp. 476–480 (2004)
9. Kwok, R.B.H.: Translations of ripple down rules into logic formalisms. In: Dieng, R., Corby, O. (eds.) EKAW 2000. LNCS (LNAI), vol. 1937, pp. 366–379. Springer, Heidelberg (2000)
10. Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S.J., McClosky, D.: The stanford CoreNLP natural language processing toolkit. In: ACL (Systems Demonstration), pp. 55–60 (2014)
11. Nguyen, D.Q., Pham, S.B., Pham, D.D.: Ripple down rules for part-of-speech tagging. In: Gelbukh, A.F. (ed.) CICLing 2011, Part I. LNCS, vol. 6608, pp. 190–201. Springer, Heidelberg (2011)
12. Richards, D.: Two decades of ripple down rules research. Knowl. Eng. Rev. **24**(2), 159–184 (2009)
13. Wobcke, W., Krzywicki, A., Chan, Y.-W.: A large-scale evaluation of an e-mail management assistant. In: Web Intelligence and Intelligent Agent Technology, pp. 438–442. IEEE (2008)
14. Xu, J., Paik, H., Ngu, A.H.H., Zhan, L.: Personal process description graph for describing and querying personal processes. In: Sharaf, M.A., Cheema, M.A., Qi, J. (eds.) ADC 2015. LNCS, vol. 9093, pp. 91–103. Springer, Heidelberg (2015)