Archives of
Transport System
Telematics

Volume 7

Issue 2

May 2014

Concept of a system for building shared expert knowledge base of vehicle repairs

B. ADAMCZYK[a], Ł. KONIECZNY[a], R. BURDZIK[a]
[a]SILESIAN UNIVERSITY OF TECHNOLOGY, Gliwice, Poland
EMAIL: blazej.adamczyk@polsl.pl

ABSTRACT
The paper focuses on technical aspects of creating a centralized expert knowledge base of vehicle repairs which is shared among its contributors. The proposed system is storing unstructured data gathered over the network from different sources such as workshops and authorized resellers. The novelty of the proposed system is to use a semantic web data store in form of OWL (Web Ontology Language) ontology in order to classify and explore the gathered data to significantly improve the process of resolution of challenging vehicle repair problems. Similar problems can be identified by using appropriate pattern recognition techniques and algorithms utilizing the Resource Description Framework (RDF) and its query language - SPARQL Protocol and RDF Query Language (SPARQL).

KEYWORDS: expert knowledge base, vehicle repairs, RDF, SPARQL

1 Introduction

Automotive technology is strongly expanding every year making vehicle diagnosis and repair process more complicated. Every car make sells several different vehicle models. Further, each model has several different variants which contain different components. The combination of the above can cause more complex and rare problems which may be very hard to identify and resolve.

In this paper we propose a system which gathers the information about car repairs from distributed computer systems over the Internet. The final goal is to gather data from different workshops and resellers around Poland. At the moment we have created a working proof of concept which is based on data from one exemplary company. In the proposed system we introduced a method of gathering data from the software which is already being used in the companies without affecting the user process of handling new orders. We also present the backend technology which allows to store unstructured data. Finally, we will discuss the pattern recognition algorithm which allowed to search through the data and correctly detect similarities. At the end we show some exemplary results of a working proof of concept, we present our plan for the future work and we draw the conclusions.

1.1 The knowledge base of car repairs

This paper aims to present a solution for an existing problem in the automotive industry – solving uncommon vehicle problems. Authors have verified the problem really exists by reviewing several car repair experts. There are few online knowledge base services existing in the Internet (e.g. Bosch Trouble Ticket System [1] or the International Automotive Technicians Network Knowledge Base [2]), however all of them require the additional step of filling the database with the information about the issues and their solutions. What is more, the existing systems require to additionally do a manual search for a solution of a challenging problem. This results in very poor growth of the knowledge base and strongly limits its usefulness.

1.2 The improvement idea

The novelty of the presented herein system is threefold:
- autom

- atic database contribution,
- use of unstructured semantic data store instead of traditional relational databases,
- use of language stemmer and synonym dictionary to improve search effectiveness.

Automatic expansion of the database without any additional user actions can be achieved by integrating with the order management software being used in the examined companies. In the next chapter we will describe in details how an integration can be achieved using the Graphical User Interface (GUI) system libraries without affecting the existing order management process. Briefly speaking, the system is being run in background and "hooks" into the order management application without any side effects. This way, when a user enters a new order description it can be automatically uploaded to the knowledge base system. Additionally, the system can be also searched for some already existing, similar issues in the database. The already solved orders can be quietly displayed on screen while the person is creating the new order. Thus, the user is being informed when a strange problem has already been solved and can contact the other database contributor (the author of the similar order) and ask for help if needed. Additionally, in the future, the software could also gather data about the performed services and the used components thus giving tips what was the real cause of the problem without contacting the source contributor.

To accumulate data from different order management applications it is required to store them in a data store which does not have any concrete structure. Thus, the second advantage of the proposed system over the existing knowledge bases is the used semantic network triple data store. This approach allows to gather differently structured data in one place in form of Resource Description Framework (RDF) [3] triples and additionally query it with the use of SPARQL Protocol and RDF Query Language (SPARQL) [4,5].

Finally in order to improve effectiveness of the pattern recognition algorithms we have used a language stemmer and a synonym dictionary. The order description provided by the contributing company is being split into sentences and words. Further, each word is converted to a base form and related with other words of the same meaning. This way the information stored in different orders have common elements and thus can be identified.

The details of used technologies and theory behind the data store and pattern recognition is further described in chapter 3.

2. Data acquisition process

The majority of car workshops and services is currently using some kind of order management software to maintain the history of repairs. Such software is used every time when a new customer orders a service. Usually some employee reviews the customer regarding the order details. This way the description of an order is prepared which is further printed and forwarded to the engineers.

In our system we propose a software (further referred to as client) which is installed on the computer which runs the existing order management application and is invisibly taking part of the mentioned process. It gathers the entered data and sends it to a centralized server for further processing. The server analyses and expands the knowledge database with provided new order information. Then it searches the existing data store for similar issues and finally sends the results in a response.

In order to properly read the entered data, the client software needs to be integrated with the existing order management application. Such integration can be achieved on several levels depending on the type of this software:

- database level,
- Application Programming Interface (API),
- Graphical User Interface (GUI) integration through operating system libraries.

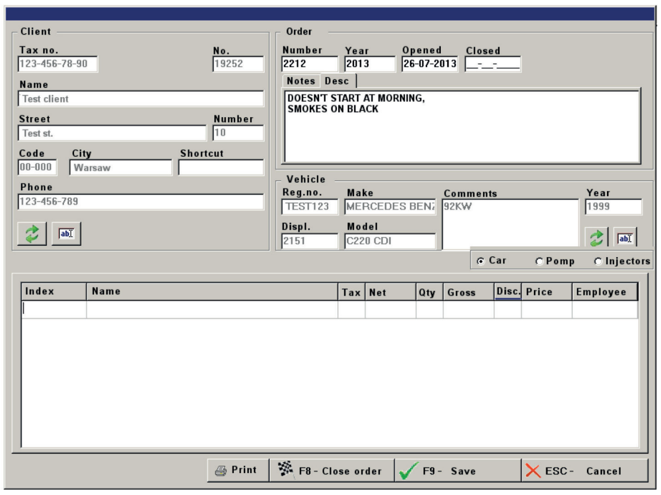
The first two integration techniques strictly depend on the technology used in the integrated application and thus would require to build a dedicated client software for certain application types. On the other hand, integration through database or API brings the best performance and most detailed order information. GUI integration, on contrary, is a more generic and dynamic solution. The drawback of this technique is that depending on application it may not be able to retrieve every entered piece of information automatically.

In this paper we want to focus on the generic GUI integration technique as it allows to create one software which will be able to integrate with any type of application. Such integration is achieved through operating system libraries which provide an interface to query the displayed application windows and controls. The most commonly used operating systems with graphical interface support provide such libraries. For example, in Windows operating system the integration can be achieved by using Windows API[6] system calls, in Linux similar goal can be achieved by using X11[7] system libraries. In this paper we focus on Windows platform as it seems it is more popular among workshops and car services in Poland.

The Windows API exposes information about the displayed user interface mostly through the User32.dll library. It can be used to retrieve information about all displayed windows and controls.

The architecture of the user interface in Windows operating system is as follows. The screen is divided onto a moveable and resizable areas called windows. Each interactive program can create several windows. They are used to perform the interaction with user – usually by means of so called controls. Controls are utilized to get some form of input from the user, for example a TextBox control allows the user to enter some text. Of course, by design, the controls created by one program are usually handled only by this program itself. However, what is important, the Windows API allows a program to access the windows and controls of any other program. This way it is possible to create an application which monitors another software and automatically retrieves the entered information. For more information about Windows API we would refer the reader to [8].

For the purpose of this paper we have integrated the client software with an order management system used by the cooperating car service. Figure 1 presents an order description window of the integrated application.

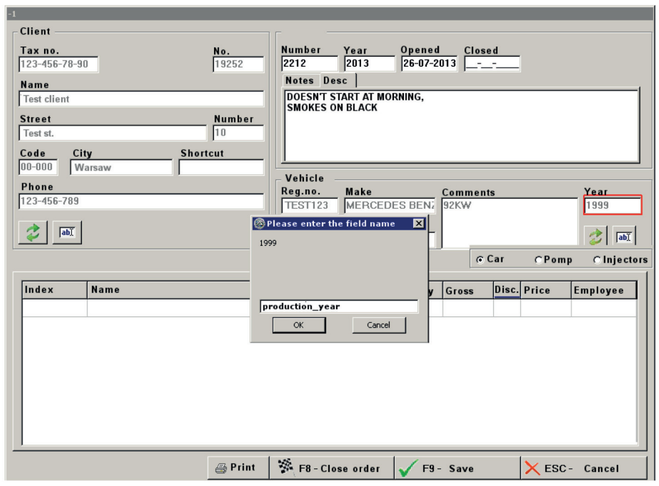

Client
Tax no. 123-456-78-90 No. 19252
Name: Test client
Street: Test st. Number: 10
Code: 00-000 City: Warsaw Shortcut:
Phone: 123-456-789

Order
Number: 2212 Year: 2013 Opened: 26-07-2013 Closed: __-__-____
Notes | Desc
DOESN'T START AT MORNING,
SMOKES ON BLACK

Vehicle
Reg.no.: TEST123 Make: MERCEDES BENZ Comments: 92KW Year: 1999
Displ.: 2151 Model: C220 CDI
(•) Car ( ) Pomp ( ) Injectors

Index	Name	Tax	Net	Qty	Gross	Disc.	Price	Employee

Fig. 1. Order description window of the integrated system [personal elaboration using the examined order management software translated from Polish]


Client
Tax no. 123-456-78-90 No. 19252
Name: Test client
Street: Test st. Number: 10
Code: 00-000 City: Warsaw Shortcut:
Phone: 123-456-789

Order
Number: 2212 Year: 2013 Opened: 26-07-2013 Closed: __-__-____
Notes | Desc
DOESN'T START AT MORNING,
SMOKES ON BLACK

Vehicle
Reg.no.: TEST123 Make: MERCEDES BENZ Comments: 92KW Year: 1999
(•) Car ( ) Pomp ( ) Injectors

Please enter the field name
1999
production_year

Fig. 2. The configuration process of client software [personal elaboration using the examined order management software translated from Polish]

The client software uses Windows API scripting tool suite called AutoIt [9] which makes the integration process easier. Using the Windows API it is possible to detect the order description window and retrieve the entered values of the interesting fields. In order to make such integration client work it is necessary to configure it properly. We have prepared a simple to use program which allows the user to configure the client software. It asks the user to select the order description window and then scans the window for all existing controls. For each detected control the program highlights it and asks for a name of the field (see Fig.2). It also displays the contents of the control so that the user can determine if this is a necessary knowledge database information. This way the user can select only the controls which store information relevant from the perspective of car repair knowledge base, for example the car make and model, its year of production and so on. For the examined order management application we have selected the following fields which are important from the perspective of the shared issue knowledge base:

- Vehicle make (*make*)
- Vehicle model (*model*)
- Vehicle engine displacement (*displacement*)
- Vehicle year of production (*production_year*)
- Additional vehicle information (*comments*)
- Order number (*number*)
- Order year (*year*)
- Order description (*description*)

Finally after this process the configuration program creates a configuration file which then can be used by the client software. It stores the important information which allow to identify correct order management application window and all the important controls to gather the entered data. An example configuration file is presented in the following listing.

```
[CLASS:Tzlecenieform]
production_year="[CLASS:TLabeledEdit;
INSTANCE:9]"
model="[CLASS:TLabeledEdit; INSTANCE:10]"
make="[CLASS:TLabeledEdit; INSTANCE:11]"
displacement="[CLASS:TLabeledEdit;
INSTANCE:12]"
year="[CLASS:TLabeledEdit; INSTANCE:14]"
number="[CLASS:TLabeledEdit; INSTANCE:15]"
comments="[CLASS:TMemo; INSTANCE:1]"
description="[CLASS:TMemo; INSTANCE:3]"
```

Finally the client software can be executed. It runs as a background process and monitors all existing windows looking for the order description window. If the window is found it scans all the configured controls creating a so called order fingerprint which can be send to the server software to update the central database. The order fingerprint is also used to identify similar orders on the server side. This way the user can be informed if similar issues have been noticed before and maybe can ask the other contributors for help in case of more challenging problem. An exemplary order fingerprint (for the example order presented in Fig. 1 and 2) is presented in the following listing.

```
model=C220 CDI
make=MERCEDES BENZ
displacement=2151
production_year=1999
comments=92KW
year=2013
number=2212
description=DOESN'T START AT MORNING,
SMOKES ON BLACK
```

Obviously different order management applications can store different information and there is no predefined fingerprint structure. The server data store accepts input formed in any kind of structure. This is realized using the server side technology described in the next chapters. Some of the fields contain a semantic information (not an atomic value but rather a longer piece of text) like the description (*description*) in the above example. For such fields the server side can apply additional language processing to split (tokenize) and classify the words in order to match similar orders more effectively. This process is described in detail in chapter 2.3.

3. The centralized knowledge database

In order to correctly process the data gathered from different contributors we decided to use a semantic triple data store instead of traditional relational database. Such approach allows to query the unstructured information with use of SPARQL language and allows for much greater flexibility.

3.1 Triple data store

A triple data store or so called triple store is a database which stores all information in form of triples composed of a subject, predicate and object elements. The metadata model is usually based on one of Resource Description Framework (RDF) specifications defined by the World Wide Web Consortium (W3C). A triple defines a relation (in form of a predicate) between the subject and the object. Almost every kind of information can be presented in such form. Figure 3 presents a representation of the examined in the previous chapter order fingerprint inside a triple store.

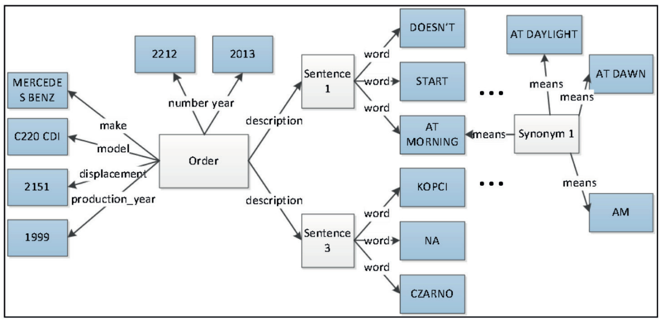


Fig. 3. Example order fingerprint triplestore representation [own study]

Subjects and objects are represented on the above figure as rectangles. Blue rectangles represent literal elements. White rectangles, on the other hand, stand for node elements. The distinction between node and literal is quite important – a node is an element which has a unique identifier inside the data store (such identifier is most commonly represented in form of Uniform Resource Identifier – URI) and the literal is identified by its value. This means that all triples relating to a concrete value (for example a number or a string value) refer to exactly the same element. This also means that if a change in a literal value is required, all triples relating to the previous value needs to be updated. Literals can only be used as objects in triples (as leaves of the tree). Nodes can be used in all three triple elements. Both literals and nodes can appear in several triples connecting the data into a graph.

Predicates are presented on Figure 3 in form of edges. Usually predicates are represented as verbs to make the triples similar to a real life sentences or facts. In this paper however predicates represent the name of some value provided in the configuration phase and thus are in form of nouns.

Figure 3 presents only one order fingerprint. It is important to realize that other orders are also linking to the same literal elements what makes the orders interconnected with each other forming a global graph of orders. Such global graph can be searched to find similarities. This can be done using SPARQL query language. The details of the pattern recognition SPARQL query are presented in the next section.

3.2 SPARQL pattern recognition

Information stored in a triple store can be queried using the SPARQL language. It allows the user to bind and select variables matching certain criteria. In other words, it allows to retrieve interesting data similarly to SQL language in traditional relational database.

The language itself is defined as a W3C Recommendation[4] and was extended to version 1.1 in [5]. The most important functionalities provided by this language are the following:

- Projection – allows to select subjects, predicates or objects in form of bounded variables,
- Filtering – projection of elements which satisfy given criteria,
- Grouping – grouping results by value allowing to perform aggregate functions over a group like sum, average etc. (GROUP BY keyword),
- Ordering – ordering results by value (ORDER BY keyword),
- Matching alternatives (UNION keyword),
- Optional binding (OPTIONAL keyword),
- Binding an expression result to projection results (BIND keyword).

The above functionalities can be used to create a query which finds similar order fingerprints in the database. When a server receives an order fingerprint it adds it to the data store. Then, the data store is queried to find all other similar orders. This is achieved by executing the following SPARQL query.

```
SELECT ?name (SUM(?os) as ?order_similarity)
{
  ?z :is :order .
  ?z :name ?name .
  ?z2 :name "2212/2013" .
  {
    ?z (!:ignore)* ?mid .
    ?mid ?rel ?common .
    ?z2 (!:ignore)* ?mid2 .
    ?mid2 ?rel2 ?common .
   FILTER (isLiteral(?common) && ?common!="")
  }
  UNION
  {
     ?z (!:ignore)* ?mid .
     ?mid :word ?word .
     ?z2 (!:ignore)* ?mid2 .
     ?mid2 :word ?word2 .
     ?common :means ?word .
     ?common :means ?word2 .
     BIND (:word as ?rel)
     BIND (:word as ?rel2)
      FILTER (?word!=?word2 && ?word!="" &&
?word2!="")
   }
   OPTIONAL { ?rel :weight ?w . }
 BIND (if (?rel!=?rel2,1,if(BOUND(?w),?w,1))
as ?os)
}
GROUP BY ?z ?name
ORDER BY DESC(?order_similarity)
```

The above query searches the data store for the orders (bounded to *?z* variable), which have the biggest number of literals in common with the sample order (*?z2* – e.g. with name equal "2212/2013"). The whole logic is based on the SPARQL property paths which allow to match zero or mare predicates between two nodes (the "*" keyword). Additionally, for all word literals which are not equal (the second part of *UNION* statement) it looks for common synonyms. To correctly classify the similarities the query additionally returns a measure of similarity: *order_similarity.* It is a value calculated according to the number of similar order properties and their weights. The results are ordered descending according to this measure.

Such similarity analysis is a wide topic and is discussed in different areas. Probably one of the biggest fields for pattern recognition is the analysis of Gene Ontology. For example, in [10], authors compare two metrics of similarity of genes. One similarity metric is also based on a semantic data store of genes called Gene Ontology.

The algorithm used in the presented system allows for result tuning thanks to the *weight* parameters. Each predicate can have a defined weight which specifies how strongly its value affects the *order_similarity*. The weights defined for the tested data store were as follows (triples declared in Turtle[11] format):

```
@prefix : <http://example.org#> .
:word :weight 5 .
:model :weight 5 .
:marka :weight 5 .
```

3.3 Language analysis process

Non-atomic order properties, like the description, need additional processing in order to allow for the pattern recognition algorithm to work effectively. This is achieved by splitting the text into sentences and further into single words. Because the words are in different inflection forms, we have used a Polish language stemmer to inflect the words to the base form. The software used for the stemming process was an open source java library called Morfologik. This way the similarity detection algorithm can work much more effectively.

Additionally, we have extended the data store with a synonym dictionary. This allows to match the orders not only by similar words in the description, but also by synonyms. Synonyms are represented in the data store as nodes which are related using the *means* predicate with all the literals of the same meaning.

The abovementioned language processing increases the effectiveness of the similarity detection algorithm by far. Probably in the future the system could be additionally improved by performing a more complex syntactic or even a basic semantic analysis.

4. Results

To test the presented similarities detection algorithm we have performed multiple searches of similar orders basing on some randomly chosen orders from the cooperating company history. Exemplary results are presented in Table 1.

Table 1. Exemplary order similarities matching algorithm results. (S – *order_similarity*). Three different searches.

Name	S	Make	Model	Year	Disp.	Other	Description (translated to English)
2357/2011	-	ford	mondeo	2005	20	96kw	SHAKES, SMOKES AND NO POWER
1174/2013	26	ford	mondeo	2005	1998	85kw	POWER LOSS, SHAKES, SMOKES
2839/2013	26	ford	mondeo	1998	20		DIAGNOSIS, NO POWER, SMOKES
2613/2011	25	ford	mondeo	2004	18	80kw	NO POWER, LOUD, HIGH CONSUMPTION
4431/2009	25	ford	mondeo	2003	1998	96kw	SMOKES ON BLACK AND NO POWER
936/2014	25	ford	mondeo	2004	1998	85kw	NO POWER, SHAKES, UNEVEN WORK
1409/2014	-	skoda	octawia	2002	19		LOUDER WORK, WHISTLES
854/2010	16	skoda	octawia	2005	19		UNEVEN WORK
2326/2008	12	skoda	octawia	2002	19		WEAK, NO POWER, MAX 120
1271/2010	11	skoda	octavia	2005	19	77kw	LOUD ENGINE WORK, CLUTTER

3834/2010	11	chrysler	voyager	2002	25		LOUD ENGINE WORK, REPAIR
3115/2013	11	citroen	berlingo	2004	19	51kw	LOUD ENGINE WORK
1587/2014	-	fiat	doblo	2007	13td	55kw	DOESN'T START AFTER NIGHT
967/2013	20	fiat	doblo	2003	1910	46kw	DOESN'T START AT MORNING WHEN COLD
668/2012	16	fiat	ducato	2007	25		ON CAR-CARRIER, DOESN'T START
2048/2010	15	fiat	doblo	2006	19	77kw	TOWED NIE PALI COS
250/2011	15	fiat	doblo	2003	19	46kw	HARDLY STARTS
3179/2008	15	fiat	doblo	2001			DOESN'T WANT TO START, TOWED

The above and other gathered results were presented to several experts and they all agreed that such knowledge base system would be very useful in solving challenging problems. The diagnosis process could be much less time consuming in certain cases.

5. Conclusion

In this paper we presented a concept of a system which allows to build and analyse a shared knowledge base of vehicle repairs. We have presented a novel approach of gathering and storing data which may result in greater usability comparing to existing knowledge base systems. We have also created a working proof of concept and presented some exemplary results of similarity detection.

Currently the system is being tested at the cooperating company and the plans are to introduce it in other workshops and car services. In the future, we are planning to add the functionality to additionally analyse the order after repair so that the system could suggest possible problem cause. This might be possible by analysing the order details after the repair. Usually order management systems store information about the sold services and products. This data may lead to a successful fix without contacting the source contributor.

Finally, it might be possible to improve the system effectiveness by attaching a wireless OBD-II adapter during the order opening process. This adapter can perform a quick diagnosis and send this data to the client software to extend the collected order fingerprint by far.

Bibliography

1. BOSCH R.: Trouble Ticket System. Online https://www.bosch-tts-new.de (accessed: 01.06.2014).
2. International Automotive Technicians Network: Fix Database. Online http://www.iatn.net/automotive-fix-database (accessed: 01.06.2014).
3. CYGANIAK, R., WOOD, D., LANTHALER, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, 25 February 2014. URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (accessed: 01.06.2014).
4. PRUD'HOMMEAUX, E., SEABORNE, A.: SPARQL Query Language for RDF. W3C Recommendation, 15 January 2008. URL: http://www.w3.org/TR/rdf-sparql-query/ (accessed: 01.06.2014).
5. The W3C SPARQL Working Group: SPARQL 1.1 Overview. W3C Recommendation, 21 March 2013. URL: http://www.w3.org/TR/sparql11-overview/ (accessed: 01.06.2014).
6. Microsoft Corporation: Windows API. Online http://msdn.microsoft.com/en-us/library/cc433218%28VS.85%29.aspx (accessed: 01.06.2014).
7. SCHEIFLER, R.W.: RFC 1013 - X Window System Protocol, Version 11. Network Working Group, June (1987)
8. CONGER, J.L.: Windows API Bible, The Definitive Programmer's Reference. Waite Groupe Press (1992)
9. AutoIt Consulting Ltd.: AutoIt. Online: http://www.autoitscript.com/site/ (accessed: 01.06.2014).
10. KOZIELSKI, M., GRUCA, A.: Evaluation of Semantic Term and Gene Similarity Measures. Pattern Recognition and Machine Intelligence, Lecture Notes in Computer Science, Volume 6744 (2011)
11. BECKETT, D., BERNERS-LEE, T.: Turtle – Terse RDF Triple Language. W3C Recommendation, 28 March 2011. URL: http://www.w3.org/TeamSubmission/turtle/ (accessed: 01.06.2014).