Business Process Aware Identification of Reusable Software Components

Lerina Aversano, Marco Di Brino and Maria Tortorella
Department of Engineering, University of Sannio, p.za Roma 21, Benevento, Italy

Keywords: Business Process Modelling, Software Modelling, Impact Analysis, Software Reuse, Similarity Analysis.

Abstract: Enterprises need to follow the rapid evolution of their business processes and promptly adapt the existing software systems. A preliminary requirement is that the software components are available, working and interoperable. A widely diffused solution is moving the adopted software solution toward an evolving architecture, such as the services-based one. The objective of this paper is to propose an approach for supporting the identification of reusable components in software systems by analyzing the business process using them. The proposed solution is based on the idea that a Service Oriented Architecture can be obtained by using a wide range of existing pieces of code. Such code components can be extracted from the existing software systems by identifying those ones supporting the business activities. Then, the paper proposes an approach for identifying the software components supporting a business process activity and candidate them for implementing a service. With this purpose, the recovery of the links existing between the business process model and the supporting software systems is exploited. An impact analysis activity is also performed starting from the initial traced components.

1 INTRODUCTION

Software systems are subject to a continuous evolution due to the frequent changes of business requirements. Actually, operative business processes can change because the implementation rules change and/or new laws are introduced. This evolution forces to keep aligned software systems with the business processes they support. This implies the execution of maintenance activities for adapting the software systems to the business process changes.

In order to facilitate the maintenance activities, a widely diffused strategy is moving the adopted software solutions toward an evolving architecture, such as the services-based one. Indeed, the detection of components impacted by the business change requirements is not obvious to the maintainers and they could be more easily identified in a service-oriented architecture.

In this direction, to support the migration of a software solution towards a service-oriented architecture, a crucial aspect is the appropriate identification and comprehension of the relations existing between business process activities and software system components. Such a kind of knowledge represents a great help to detect the software components candidate to be moved towards services.

The identification of candidate services in a structured legacy system during a migration process is a challenging task (Khadka et al., 2013a; Kontogiannis et al., 2008; Zillmann et al., 2011). In fact, the lack of updated documentation and resources makes very hard the code comprehension.

This paper proposes an approach based on the analysis of the business process using the software systems to be migrated. Indeed, a requirement change is often expressed with reference to the business activities it supports. In particular, the proposed approach identifies the links between a business process description with the components of a software systems that could be used for the service oriented architecture migration (Aversano et al., 2015). It exploits a formal description of the business process based on BPEL language and identifies the software components of the examined software system that are connectable to the business activities. The application of an impact analysis process completes the definition of all the software components that could implement a service.

The following of the paper is structured as follows: Section 2 discusses the related works

Aversano, L., Brino, M. and Tortorella, M.
Business Process Aware Identification of Reusable Software Components.
DOI: 10.5220/0005985700590068
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA*, pages 59-68
ISBN: 978-989-758-194-6

regarding the identification and evaluation of services; Section 3 describes the proposed approach and supporting tool; Section 4 presents the application of the approach to a case study; and concluding remarks and future works are discussed in the last section.

2 RELATED WORKS

In the literature there are several approaches suggesting useful guidelines to identify services during the legacy systems migration toward a service-based architecture.

In (Khadka et al., 2013b), the authors proposes two current practices for a correct candidate services identification: top-down, where a business process is initially modeled based on the requirements and then it is subdivided into sub-processes until these can be mapped to legacy functions; and bottom-up, utilizing the legacy code to identify services by using various techniques, such as information retrieval, concept analysis (Zhang et al., 2006), business rule recovery (Marchetto and Ricca, 2008) and source code visualization (Van Geet and Demeyer, 2008).

In (Cetin et al., 2007), the authors describe a mash-up based strategy to be applied during a legacy system migration process. In this strategy, system components might be reusable legacy components or new developed ones, depending on if there is a gap exiting between the existing legacy component and the requirements.

In (Balasubramaniam et al., 2008) the authors discusses an architecture-based and requirement-driven service-oriented re-engineering method. This method entails the availability of architectural and requirement information. The services identification is performed by the domain analysis and business function identification.

There are also other approaches that can be used to evaluate services. One of them is the one proposed in (Matos and Heckel, 2008) that performs either code pattern matching and graph transformation. The approach is based on source code analysis for identifying the contribution of code fragments to architectural elements and graph transformation for architectural migration, allowing for a high degree of automation.

Another approach proposes to evaluate services by feature location (Chen et al., 2005). The more practical definition of a feature is used as a coherent and identifiable bundle of system functionality that is visible to the user via the user interface (Eisenbarth et al., 2003; Turner et al., 1999). Then, to discover feature implementation, feature location is applied. It is a re-engineering technology used to locate a particular feature in the most relevant code, understand it and make the change so as to minimize unwanted side effects (Turner et al., 1999). After identifying the source code which is involved in the implementation of a particular feature, the implementation modules are aggregated into one module. Therefore, the core source code of the service operations can be extracted and the service identification is achieved.

Another approach proposes to evaluate services by formal concept analysis (Chen et al., 2009). The identification process of service candidate is based on the mapping between Functionality Ontology and Software Component Ontology and adopts relational concept analysis.

In (Sneed, 2006), an automatic approach to evaluate candidate services in a migration process from legacy system to SOA is proposed. Groups of object-oriented classes are considered as candidate services and evaluated in terms of development, maintenance and estimated replacement costs. If user organizations want to move toward a service oriented architecture, it must make a portfolio analysis of their existing applications and to list out the essential business rules.

In (Sneed et al., 2012), the authors proposes a tool for assisting the reuse of existing software systems in a service oriented architecture by linking the description of existing COBOL programs to the overlying business processes. For linking models of existing code to a business process model, this approach applies the interpretation of the code interfaces as separate service subjects. The approach fits better to the concept of a service-oriented architecture and is also more intuitive, as it is possible to generate a service layer within a BPM suite that links it to the underlying code. The actual business process events take place above this layer. They guide the human users through their tasks, telling them what to do next. These higher level control subjects are equivalent to work flow control procedures written in a job control language (JCL). In this way, a mixture of bottom-up and top-down approach to SOA design is supported. First, access subjects are created bottom-up to link the business model to the underlying code base, then process control subjects are defined top-down to depict the actual business work flows, but based on the lower level of BPM service layer.

Also the approch proposed in this paper exploits the business knowledge derivig from the business process using the software systems considered for

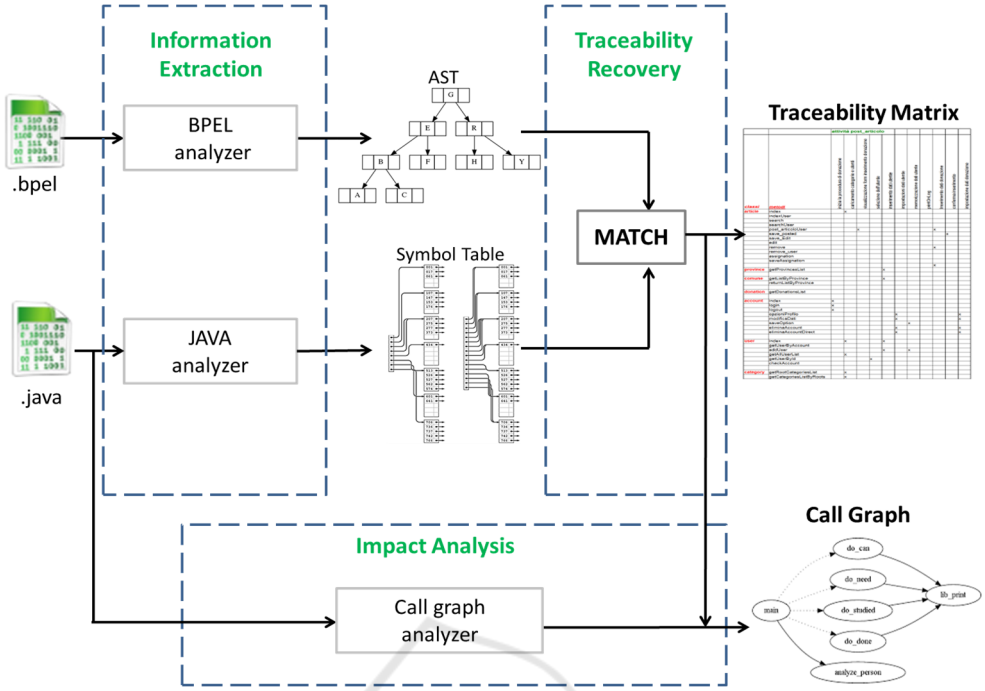

Figure 1: Overview of the approach for detecting candidate services.

identifying reusable components. As the approach works on business and software models, it is independent from the specific programming langage and if the business process and software systems are adequately modelled, it can be applied for identifying reusable components of different programming languages.

3 APPROACH FOR CANDIDATE SERVICES DETECTION

The proposed approach aims at identifying the traceability links between business process activities and supporting software system components. It is based on the analysis of the business process entities and software components models. The software components linked to the business process entities represent the initial components for identifying candidate pieces of software to be migrated toward services.

The proposed approach is illustrated in Figure 1 and entails three processing phases described in the following: *Information extraction, Traceability recovery and Impact analysis.*

3.1 Information Extraction

The information extraction phase regards the extraction of semantic information from both business process and software system source code. Figure1 shows that this phase in based on the use of two parsers for analyzing Java code and BPEL files, and obtaining all the needed information for performing the next traceability recovery. The Java and BPEL parsers used were implemented by using the JavaCC (Java Compiler Compiler) parser generator, after having defined the appropriate grammars.

With reference to the business process, the BPEL parser allowed the construction of the model description syntax tree. This Abstract Syntax Tree depicts the hierarchical relation existing between business activities, composing sub-activities and artefacts needed for executing them.

After obtaining the syntax tree, it was enriched with additional nodes for inserting comments, and identifying the associations existing between them and code representing activities. Therefore, the analysis of the BPEL AST allowed the identification of the identifiers for describing the analysed business process.

With reference to the software system, the Java parser constructed a symbol table used to keep track of the source program constructs and, in particular, the semantics of the identifiers concerning the packages, classes, methods, instance variables and local method variable declarations.

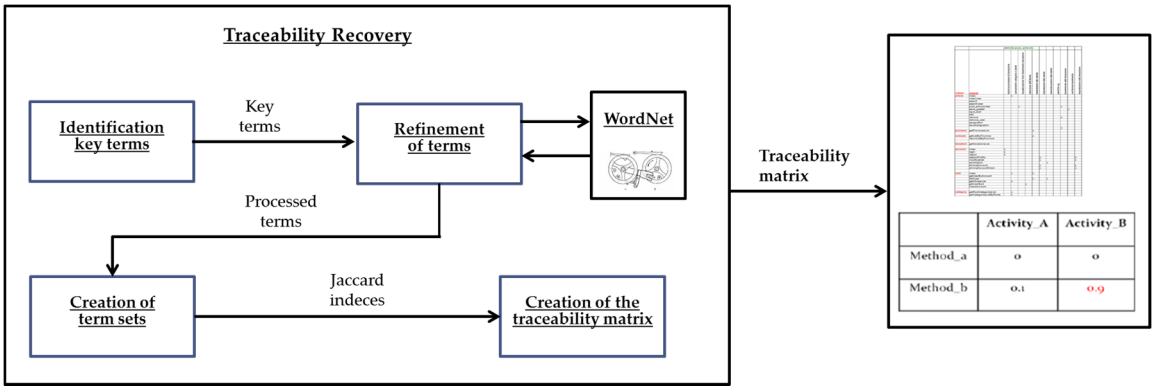

Figure 2: Traceability recovery phase.

The produced symbol table structure is also hierarchical. The first layer contains a list of all packages declared in the considered project. Each record of this first layer contains a reference to another list, regarding the classes defined in the considered package. The set of all classes is included in the second layer of the symbol table. Therefore, each class contains a references to the members it declares, such as methods and instance variables, and inner class. Each method could have another layer representing the set of local variables it declares. Each inner class may refer to other layers grouping its methods and variables.

The comments are analysed in the pre-processing phase. Once a comment is identified, it is saved into a map, which also stores the appearance order of the various comments.

When the pre-processing phase is completed, the parser considers the comments for identifying additional semantic information contained in the code.

3.2 Traceability Recovery

The traceability recovery aims at discovering the connections existing between the business activities and software system components.

Once obtained the syntax tree for the BPEL business description and the symbol table for the Java software system, they are visited in a post-order manner for collecting the information required for continuing the analysis.

This processing phase was divided into 4 different steps, summarized in the chart drawn in Figure 2.

Identification of Key Terms. This is the first step of the traceability recovery phase. It requires the visits of the BPEL syntax tree and the creation of an array of BPEL activities, called *Activity*. Each *Activity* objects includes the BPEL file name, the task name and the set of related terms. As an example, for every single *invoke* activity, information is collected regarding its parameters, like *name* and *operations*, while for every *reply* and *receive* activities information is collected regarding *portType* and *partnerLink* parameters. Similarly, the visit of the Java symbols table permits to identify all the key terms related to the methods. Once identified one of the keys, a string set containing the method name, any local variables name and inner classes, is created.

Refinement of Terms. The second step of traceability recovery phase entails 3 tasks. The first task regards the tokenization of the selected terms. Then, every composed term is split into two or more words and each term is normalized. As an example, for a method called *GetCustomerName()*, the terms *GetCustomerName*, *get*, *customer* and *name* are obtained and included in the collection. The second task makes it possible a refinement of the terms, eliminating the stopwords, that are the most common words included in the English grammar, and the Java and BPEL keywords, as they do not add any additional information regarding Java methods or BPEL process content. The third step aims at collecting a set of synonyms for each term obtained from the previous tasks. This operation is made by using the WordNet library, which is a lexical-semantic database for the English language, developed from Princeton University. Therefore, a vector of synonyms is generated for each term within the set of created words, which is added to the starting sets of terms.

Creation of Term Sets. The third step of traceability recovery phase regards the creation of different subsets of terms obtained by the previous steps. Specifically, for the *invoke* activity of the BPEL model, the following sets are obtained:

- a set of the terms included in the string associated to the *operation* argument;
- a set of the terms included in the string associated to the *name* argument;
- a complete set of the terms contained in strings associated to the following arguments: *operation*, *name*, *partnerLink*, *inputVariabile* and *outputVariable*.

The created sets for the *reply* and/or *receive* activities are the following:

- a set of the terms contained in the string associated to *portType* argument;
- a set of the terms contained in the in the string associated to *partnerLink* argument;
- a complete set including the terms contained in the strings associated to the *portType* and *partnerLink* arguments..

With reference to the Java methods, the following sets are defined:

- a set with the terms of the considered method identifier;
- a complete set including the method name, its local variable names and inner classes.

Creation of the Traceability Matrix. Once completed the terms preprocessing, it is possible to continue computing the similarity between the identified terms, in order to obtain the traceability matrix. The adopted similarity coefficient is the Jaccard index that is a statistical index used to compare the similarity and diversity of sample sets. The value of this coefficient is defined in a range of values going from 0 to 1 and it is defined as the size of the intersection of the sets of samples divided by the size of their union:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

In the proposed approach, the sets of terms used for applying the Jaccard index are those described in the previous section. Using the Jaccard similarity the traceability matrix is produced. It is organized so that the rows contain the method of the analysed software system, and the columns regard the BPEL activities extracted by the parser. Therefore, cell *i,j* of the matrix contains the value of Jaccard index of the terms regarding method *i* and those ones concerning activity *j*.

A preliminary investigation suggested to consider as relevant values of the traceability matrix, those ones greater of 0.85. Then, whenever there is a correspondence involving a Java method and a BPEL activity with the relative Jaccard index higher than 0.85, it is marked as meaningful and it is indicated with a different colour.

3.3 Impact Analysis

Once the traceability matrix is obtained, it is possible to make a further analysis of the software components, focusing on the identification of the software system classes connected to classes identified in traceability recovery phase. This phase is divided into 2 different steps as indicated in Figure 3.

Creation of Call Graphs. Analysing from the source code of Java software system, a call graph is obtained representing the call relationship existing between the methods of a software system. Specifically, each node represents a class or a method and each edge *(a,b)* indicates that method *a* or a method included in class *a* calls method *b* or one included in a class *b*. The Doxygen tool (www.**doxygen**.org) was used for generating the call relationship. In particular, Doxygen can produce different outputs from a set of documented source files. In the proposed approach, the considered output was represented by files *.dot*, which contain the call graph definition. Doxygen creates a *.dot* file for each method of the analyzed software system that calls at least another method of the software system itself. Every single call graph starts from the method node that is being analyzed and browse the entire chain of calls it triggers.

Call Graphs Analysis. The second step of the impact analysis phase visits the obtained call graphs for identifying the impacted software components. For improving the readability of this new graph and performing a high-level analysis, each of its nodes represents a single class of the software system, rather than a class method. Furthermore, a numeric label is associated to each edge for counting the number of calls occurring between the two involved classes. Furthermore, the various nodes have been grouped into subgraphs, each of which represents a package of the analyzed software product and contains the nodes representing its classes.

The information obtained by analysing the traceability recovery phase and regarding a candidate method for a future migration to service oriented architecture, is particularly relevant also in the impact analysis step. Actually, it represents the triggering software component that permits to obtain the set of components it impacts and that can be clustered together for being migrated to a service.

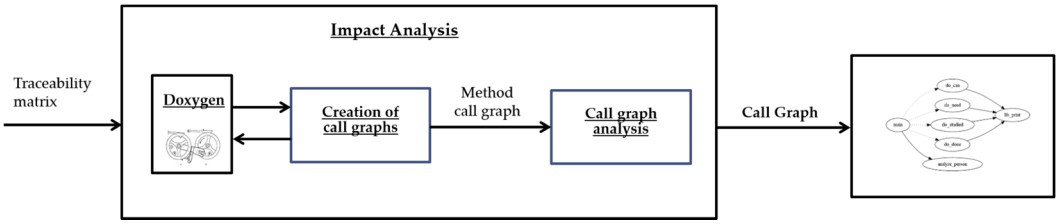

Figure 3: Impact Analysis phase.

BARC Plugin. A prototype tool has been implemented for supporting the application of the proposed approach. It is an Eclipse plugin named BARC, that is an acronym for Business process Aware identification of Reusable software Components. The plugin requires the project path of the various Java and BPEL files, through message dialog, at runtime. After entering the path, the plugin will analyses the source code and produces 2 outputs.

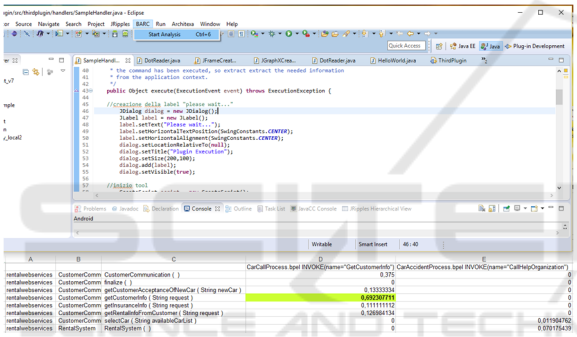

Figure 4: Screen shot of the implemented Eclipse plugin.

The first output is the traceability matrix, created by Traceability recovery phase. While, the second output is the result of the Impact analysis step, that is a class call graph. In this graph, the nodes represent Java classes and each edge represents a link between two classes. The construction of this graph was made by using JGraphX: it is a Java library that allows the creation of the call graph and represents it through a JFrame window.

4 VALIDATION

The effectiveness of the presented approach has been validated using a case study regarding a web Java project, dealing with the management of a dealership. It consist of 1066 Java files (code lines 124459) and 33 BPEL files. The project provides functionalities such as: user registration, access to the services, browse the offered catalogue, add or remove a product to the cart, confirm the purchase.

The BPEL model of the business process has been obtained analyzing the executed processes and using the available knowledge, without considering the source code of the used software system. The obtained model was also manually verified.

It follows a description of the application of the approach, and the obtained results. This project entails the execution of 3 macro operations:

- **Sales:** including the functionality allowing the management of some services concerning the car selling for both new and used cars, such as: Car Selling (information service, choosing car, providing price, negotiating and making contract details, payment style etc.), Car Return (car checking, invoke approvement etc.), Customer Service (car cancellation, car fuelling, car cleaning, car checking, car accident etc.), Choosing Car and Customer Reception;
- **Rental:** including the operations allowing the management of some services relating to the car rental, such as: Car Renting (input customer and car information, input pick-up/drop-off location and date, price offer, rental acceptance, payment), Car Reservation (reservation of a favourite car model in a preferred date and location), Car Insurance (Insurance to the rental car), Car Service (car checking, car cleaning, car fuelling), Reminder (reminding customer to pay, warning payment expiry, arranging collection company), etc.;
- **Customer Communications:** including the operations concerning the communication form/to a user, such as: get the information about a user, a rental or an insurance.

The verification and interpretation of the results required the selection of just five interesting Java classes. Each of them contributes to implement the previously described macro functionality, and includes the needed methods. Each BPEL file represents one of the macro service operations. They are: *NewCarSellingProcess.bpel*, *SaleInformation.bpel*, *CheckDrivingLicenseProcess.bpel* and so on.

In the following, the application of the steps described in the previous section is discussed.

```
INVOKE
inputVariable => "getCustomerInfoRequest"
name => "GetCustomerInfo"
operation => "getDriverInfo"
outputVariable => "getCustomerInfoResponse"
partnerLink => "RentalSystemService"
portType => "ns1:RentalSystem"
-----------------------------------------------
-----------------------------------------------
INVOKE
inputVariable => "callForHelpRequest"
name => "CallHelpOrganization"
operation => "callForHelp"
partnerLink => "Accident_HelpOrganization"
portType => "ns2:HelpOrganization"
```

Figure 5: Information extraction about an invoke activity.

Identification of Key Terms

The information regarding the activities are selected by analyzing the BPEL AST. Figure 5 shows an example of an invoke activity, which is called *getCustomerInfo*, that is contained in the BPEL file called *CarAccidentCallProcess.bpel*. This file handles the calls arriving in dealer after a car accident. The activities the figure shows are used for getting personal information about the car driver.

Afterwards, all the information of interest is extracted from the activity shown in Figure 5. The values of the specific example are the following:

- inputVariable (*getCustomerInfoRequest*);
- operation (*getDriverInfo*);
- outputVariable (*getCustomerInfoResponse*);
- partnerLink (*RentalSystemService*);
- portType (*ns1: RentalSystem*).

The information regarding each method is extracted from the Java symbol table, including the method name and all the identifiers linked to it.

Figure 6 shows an example of a method, called *initialize*, contained in a Java file called *RentalSystem.java*, which is the Java class that deals with the management of some services relating the car rental. The figure highlights all the identifiers that are caught in this step. They regard the name of: the step that is equal to the method name (*initialize*), local variables (*maxX*, *maxY*, *minX*, *minY*), and the method inner class (*initializeVector*).

Refinement of Terms

All of attributes of the selected BPEL activity have a compound name that can be split. For example, the name of its *operation* attribute (*getDriverInfo*) can be split into 3 different words: *get*, *driver* and *info*. Similarly, the name of a Java method can be split into different words without considering the list of arguments.

The obtained set of words is then refined going to remove the terms included in the list of English stopwords, such as *get*. Then, the Java identifier *getDriverInfo* provides just the two terms *driver* and *info* for the next phase.

The synonyms of each identified single word are searched by using the WordNet library. For example, WordNet provides *information* as synonym of *info*, and *device driver* and *number one wood* for the term *driver*.

Creation of the Term Sets

With reference to the used invoke activity, three sets of terms are created. For example, the following sets are defined for activity *getDriverInfo*:

- a set containing the words: *getDriverInfo*, that is the complete name of the *operation* attribute, and *driver* and *info*, which are the words obtained by the splitting;
- a set containing the words: *getCustomerName*, that is the complete name of the *name* attribute, and *customer* and *name;*
- a set containing the words: *driver*, *info*, *customer*, *rental*, *system* (from its attributes), *information*, *client*, *lease*, *letting*, *renting*, *scheme*, *organization*, *organisation*, *arrangement* (from WordNet).

In the same way, for each Java method, two set of terms are created. For example, if method *getDriverInfo* is considered, the first sets of terms includes the words deriving from the method name, while the second set contains the words *driver*, *info*, *information* (obtained from its name), *sql*, *stm*, *rset*, *e*, which are the names of the method local variables.

Creating Traceability Matrix

For calculating the Jaccard index for the set of terms, the first sets that must be compared are the BPEL set that includes information regarding *operation* attribute (e.g., words: *getDriverInfo*, *driver*, *info*) and the Java set that includes information regarding the related method (e.g., the words: *getDriverInfo*, *driver*, *info*). As it is possible to see in the example, these sets contains the same words, and, then, the Jaccard index resulting from their comparison will be equal to 1, which indicates a full matching.

If the match between the BPEL activity and Java method has already been found, it is useless to make comparisons with other sets of terms, and this step may end with the setting of the obtained values within the traceability matrix.

```
name: initialize --- kind: method --- scope:  --- type_return: void --- args: Container parent
    name: maxX --- kind: local variable --- scope:  --- type_return: int --- args:
    name: maxY --- kind: local variable --- scope:  --- type_return: int --- args:
    name: minX --- kind: local variable --- scope:  --- type_return: int --- args:
    name: minY --- kind: local variable --- scope:  --- type_return: int --- args:
    name: initializeVector --- kind: inner class --- scope: public --- type_return: --- args:
```

Figure 6: Information extraction about a Java method example.

Table 1 shows the Jaccard values of the method and activity considered in the example, and highlights the correspondence found between the BPEL business process and Java system software.

Table 1: Example of matrix produced by prototype tool.

	RECEIVE (name="receiveInput")	INVOKE (name="GetCustomerInfo")	INVOKE (name="CallHelpOrganization")
getCarInfo (String carID)	0,0874999985098839	0,333333343267441	0
getDriverInfo (String custID)	0	1	0
getEmployeeInfo (String employeeID)	0	0,333333343267441	0

```
digraph G
{
  edge [fontname="Helvetica",fontsize=10,labelfontname="Helvetica"];
  node [fontname="Helvetica",fontsize=10,shape=record];
  rankdir=LR;

  Node1 [label="Controller::RentalSystem::getDriverInfo",
         height=0.2,width=0.4,color="white", style="filled"];
  Node1 -> Node3 [color="midnightblue",fontsize=10,style="solid"];
  Node2 [label="Controller::RentalSystem::getCustomerName",
         height=0.2,width=0.4,color="white", style="filled"];
  Node2 -> Node4 [color="midnightblue",fontsize=10,style="solid"];
  Node3 [label="Model::Customer::getCar",
         height=0.2,width=0.4,color="white", style="filled"];
  Node1 -> Node4 [color="midnightblue",fontsize=10,style="solid"];
  Node4 [label="Model::Customer::getUser",
         height=0.2,width=0.4,color="white", style="filled"];
}
```

Figure 7: Information obtained from the Doxygen analysis.

If the match between the BPEL activity and Java method has already been found, it is useless to make comparisons with other sets of terms, and this step may end with the setting of the obtained values within the traceability matrix. Table 1 shows the Jaccard values of the method and activity considered in the example, and highlights the correspondence found between the BPEL business process and Java system software.

Call Graphs Creation

This new processing phase required the analysis of the Java source files and the information obtained in the extraction phase.

Every single Java file in the analyzed software product will examined by using Doxygen. A correct use of this tool required the configuration of the file including all information regarding the examined software product.

The execution of Doxygen produced a temporary folder with all the files created by the tool, such as files with *.dot* extension.

Figure 7 include an example of a *.dot* file. Each node is represented by a unique identifier (e.g., *Node1*) and some parameters, enclosed in square brackets. In addition, for purely graphic purposes, parameter *label* also defines the node name, composed of the union of the package name, class name and referenced method name. For example, information relating method *getDriverInfo* are contained in the label of a node called *Node1*. In the same file, the various edges connecting two nodes are also defined; in Figure 7, *Node1* is connected with *Node3* (*getCustomerInfo*) and *Node4* (*getUser*).

Call Graphs Analysis

Once the call graph of software system has been obtained, it is possible to create a call graph at the class level that includes all the obtained information. For example, the nodes related to methods *getCustomerName* and *getDriverInfo* can be grouped into a new single node, that is *RentalSystem*. In the same example, method *getDriverInfo* is one of those methods identified in the Traceability recovery phase as a possible candidate to be turned into a service. This information is also saved into the new call graph. Actually, in the *label* parameter of its class, the candidate method name itself is added.

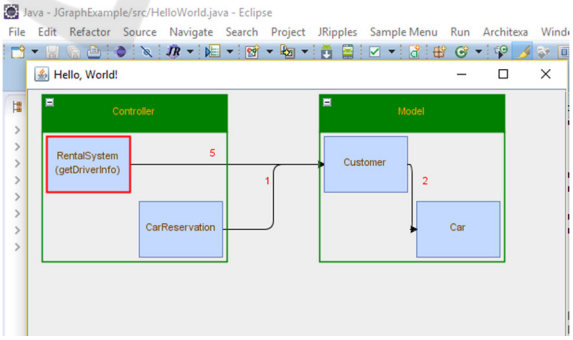

Figure 8: Classes call graph produced by prototype tool.

The final result regarding a call graph at the class level, is shown in Figure 8. The figure includes a screen shot of the Eclipse plugin that has been implemented for supporting the application of the approach. In the figure, there are 2 packages, called *Controller* and *Model*, that enclose some nodes,

including node *RentalSystem*. The edge that connects this node with the *Customer* class node has a weight (in this case is equal to 5), which indicates that *RentalSystem* class and its methods, interacts with the *Customer* class and its methods for 5 times.

Table 2 contains a summary of the results obtained for the case study in terms of identified interactions in the traceability matrix between business activities and software components. This analysis has been performed by comparing the automatically obtained results with those ones attained through a manually investigation of the software system. Table 2 includes indications of: the false positives, indicating the detected but not real correspondences; true positives, regarding detected and real correspondences; false negatives, concerning no found but actually present correspondences; and true negatives indicating not found and not really present correspondences. It can be observed that the value of the false negatives is very low, just 1, and this indicates that the proposed approach detects the correspondence correctly, when it exists. The number of false positives, 15, are due to correspondences that do not exist, but they have been detected because the analyzed activity has a nomenclature similar to the one of a Java method, but there is no real correspondence between them.

Tables 3 contains the results obtained by the evaluation of the Precision, Recall and F-Measure in the dealer project. It is possible to observe that the obtained results are enough high, indicating the goodness of the results regarding the obtained correspondence among business and software terms. This results have been achieved also thanks to the meaninglessness of the used terms.

Table 2: Experimental results for dealership project.

Case study	False Positives	False Negatives	True Positives	True Negatives
Dealership	15	1	60	13769

Table 3: Precision, Recall, F-Measure for dealership project.

Precision	Recall	F-Measure
0.8	0.98	0.88

Additional results can be obtained by considering the comments in both BPEL and Java files. In this case, the association of a single comment to the BPEL activity requires the addition of further comments nodes to the AST, including the detected comments and relative source code.

The comment-based approach was also analysed on the same software system and the new Jaccard index values were compared with the previous ones. Different results were obtained mainly due to the considerable increase of terms to be considered in the business and software term sets. Thus, because of the considerable decrease of common terms compared to the total number of terms, many of the real correspondence existing between Java methods and BPEL activities (i.e. true positives) were not found. This experience indicated that considering comments does not contribute to improve the results but it just increases the number of terms to be considered for identifying the correct correspondence between BPEL and Java terms.

5 CONCLUSIONS

The paper presented an approach aiming at supporting the reuse of the existing software systems components connected to a business process. In particular, this facilitation is provided through the possibility of detecting the correspondences existing between source code components and business activities of a process modelled by using the BPEL language.

The approach execution entailed the use of two parsers. The information extracted by using the parsers have been expanded and refined for being used in the traceability link recovery. The evaluation and selection of such correspondences have been performed by using a similarity measure defined in the paper. Each identified software component was considered in an impact factor activity aimed at searching all the software components it called for encapsulating all of them in a service.

The BARC eclipse plug-in was implemented for automatically supporting the application of the approach.

The comments in the code were also initially analysed, but successively discarded as it was observed that their use leads to worse results.

The preliminary results obtained by the application of the proposed approach are encouraging and represent a starting point, for the identification of parts of the code from an existing software system with the aim of defining new services to be used in a service oriented architecture. The values of *precision, recall, f-measure* show the potentiality of the proposed approach.

The future work will concern the refinement of the selection of the correspondences in the matrix (refining the values in the range used for the analysis of Jaccard indexes), expanding test cases and extending the analysis also to WSDL files.

REFERENCES

Aversano, L., Di Brino, M., Di Notte, P., Martino, D., Tortorella, M., 2015. Linking Business Process and Software Syste. In *BMSD 2015*, *5th International Symposium on Business Modeling and Software Design*. SCITEPRESS.

Balasubramaniam, S., Lewis, G. A., Morris, E. J., Simanta, S., Smith, D. B., 2008. SMART: application of a method for migration of legacy systems to SOA environments. In *ICSOC'08*, *6th International Conference on Service-Oriented Computing*. Springer-Verlag.

Cetin, S., Altintas, N. I., Oguztuzun, H., Dogru, A. H., Tufekci, O., Suloglu, S., 2007. A mashup-based strategy for migration to service-oriented computing. In *ICPS'07*, *International Conference on Pervasive Services*. IEEE Comp.Soc. Press.

Chen, F., Li, S., Chu, W. C., 2005. Feature analysis for service-oriented reengineering. In *APSEC'05*, *12th Asia-Pacific Software Engineering Conference*. IEEE Comp.Soc. Press.

Chen, F., Zhang, Z., Li, J., Kang, J., Yang, H., 2009. Service identification via ontology mapping. In *COMPSAC'09*, *33th Annual International Computer Software and Applications Conference*. IEEE Comp.Soc. Press.

Eisenbarth, T., Koschke, R., Simon, D., 2003. Locating features in source code. In *IEEE Transaction on Software Engineering*, Vol. 29, No. 3.

Ganter, B., Wille, R., 1999. *Formal Concept Analysis: Mathematical Foundations*, Springer-Verlag.

Khadka, R., Saeidi, A., Idu, A., Hage, J., Jansen, S., 2013a. Legacy to SOA evolution: a systematic literature review. In *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. A. D. Ionita, M. Litoiu, G. Lewis Editions. IGI Global.

Khadka, R., Saeidi, A., Jansen, S., Hage, J., 2013b. A structured legacy to SOA migration process and its evaluation in practice. In *MESOCA'13*, *7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*. IEEE Comp.Soc. Press.

Kontogiannis, K., Lewis, G., Smith, D., 2008. A research agenda for service-oriented architecture. In *SDSOA'08*, *2nd international workshop on Systems development in SOA environments*. ACM press.

Marchetto, A., Ricca, F., 2008. Transforming a Java application in an equivalent Web-services based application: toward a tool supported stepwise approach. In *WSE'08*, *10th International Symposium on Web Site Evolution*. IEEE Comp. Soc. press.

Matos, C. M. P., Heckel, R., 2008. Migrating legacy systems to service-oriented architectures. In *ICGT 2008*, *Doctoral Symposium at the International Conference on Graph Transformation*. Electronic Communications of the EASST.

Sneed, H. M., 2006. Integrating legacy software into a service oriented architecture. In *CSMR'06*, *10th European Conference on Software Maintenance and Reengineering*. IEEE Comp. Soc. press.

Sneed, H. M., Schedl, M., Sneed, S. H., 2012. Linking legacy services to the business process model. In *MESOCA'12*, *6th IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*. IEEE Comp. Soc. press.

Turner, C. R., Fuggetta, A., Lavazza, L., Wolf, A. L., 1999. *A conceptual basis for feature engineering. Journal of System and Software*, Vol. 49, Issue 1. Elsevier press.

Van Geet, J., Demeyer, S., 2008. Lightweight visualisations of COBOL code for supporting migration to SOA. In *Evol'07*, *3rd International ERCIM Symposium on Software Evolution*. Electronic Communications of the EASST.

Zhang, Z., Yang, H., Chu, W., 2006. Extracting reusable object-oriented legacy code segments with combined formal concept analysis and slicing techniques for service integration. In *QSIC'06*, *6th International Conference on Quality Software*. IEEE Comp. Soc. press.

Zillmann, C., Winter, A., Herget, A., Teppe, W., Theurer, M., Fuhr, A., Horn, T., Riediger, V., Erdmenger, U., Kaiser, U., et al., 2011. The SOAMIG Process Model in Industrial Applications. In CMSR'11, *15th European Conference on Software Maintenance and Reengineering*. IEEE Comp. Soc. press.