Software Artefact Traceability Analyser: A Case Study on POS System

I. D. Rubasinghe
ireshar@cse.mrt.ac.lk

D. A. Meedeniya
dulanim@cse.mrt.ac.lk

G. I. U. S. Perera
indika@cse.mrt.ac.lk

Department of Computer Science and Engineering,
University of Moratuwa, Sri Lanka

ABSTRACT

Software traceability is a key notion in the software development. The paper explores the previously developed research-based Software Artefact Traceability Analyser tool called 'SAT-Analyser'. The workflow and capabilities of SAT-Analyser tool are described and evaluated using a case study of a Point of Sale system. Phases such as software artefact identification, data pre-processing, data extraction and traceability establishment methodologies used in the tool SAT-Analyser are presented with graph-based traceability outcome. The case-study based evaluation shows positive accuracy results for the SAT-Analyser tool. Moreover, the proposed traceability management framework for the entire software development life cycle is presented.

CCS Concepts

• Software and its engineering → Software creation and management → Software post-development issues → Software evolution

Keywords

Traceability establishment; Visualization; Traceability graph, SAT-Analyser tool.

1. INTRODUCTION

Software system development is challenging due to the changes occur in requirements, business organizations, legal rules and improper use of tools and technologies. Managing these changes is difficult and affects the success or the failure of a software system. Thus, it is essential to have appropriate solutions to handle the changes during the Software Development Life Cycle (SDLC). The changes can occur at any phase to any intermediate software outcomes, which are called artefacts. An alteration to a single artefact can affect one or more other artefacts in one to many phases with different severities. Therefore, identification of the changes, affected artefacts, severity and the consequences is important to manage artefact traceability throughout the SDLC. Accordingly, the notion of software traceability has been evolved to enable tracing capabilities among software artefates.

ICCBN 2018, February 24–26, 2018, Singapore, Singapore
© 2018 Association for Computing Machinery
ACM ISBN 978-1-4503-6360-0/18/02...$15.00
https://doi.org/10.1145/3193092.3193094

Consequently, today different software traceability support tools and frameworks can be identified [1][2]. However, most solutions are research-based due to the challenging limitations.

Software Artefact Traceability Analyser (SAT-Analyser) tool described in this paper is one such software traceability support tool. It is capable of establishing traceability among software artefacts in requirement, design and source code level and to visualize the traceability graph for a given software application. Thus, this paper describes a Point of Sale (POS) based case study demonstrating the process, workflow of the SAT-Analyser and evaluates the accuracy of the traceability establishment process.

The paper is structured as follows. Section 2 explores set of related work and Section 3 describes the case study application. The accuracy of the tool is evaluated in Section 4 and Section 5 concludes the paper with possible future extensions.

2. BACKGROUND STUDY

Software traceability is the ability to track artefact behaviors during the software development process by providing a logical connection among artifacts. Software traceability process consists of several sub processes such as establishing traceability links among artefacts and traceability maintenance [3].

An architecture-centric, stakeholder-driven, industry-oriented and open hypermedia traceability approach influenced by e-Science technologies, is presented in [4]. It has addressed the multi-faceted traceability problem by integrating the implementation to a traceability tool named ArchStudio. They have followed a rule-based classification approach for establishing artefact-link relationships and n-ary first class links for trace relationships. A facet-based approach by Grammel [5], has narrowed the traceability scope into model-driven software development. They have traced the model transformations using a domain specific language (DSL) called trace-DSL for data extraction. A work on source code and test case artefacts traceability using gamification technologies is presented in [6], with a proof-of-concept prototype called GamiTraci. It is highly influenced by the similar previous work [7], that has used slicing and conceptual coupling techniques in establishing test to code traceability.

The accuracy of the traceability results is a major challenge. The reliability of the traceability is addressed in [8], that can be applied for safety critical software systems. It can be identified as a light-weight results-oriented solution. This trace link model separates the untrusted links and conducts a remediation process continuously. However, there is a limitation of trace maintenance facilities. Traceclipse [9], is a research-based Eclipse plugin targeted for traceability link recovery and maintenance. Its link recovery process is influenced by Information Retrieval (IR)

techniques and limited for source code artefact in Java. A similar work on semi-automated traceability recovery with the use of IR and classification is presented in [10]. An ontology-based attempt has been conducted in [11], by mapping domain concepts and artefact indexes into an ontology. But the traceability support of it is limited only for Unified Process based software development. Although, there exist limitations and challenges in achieving traceability within the SDLC, traceability management has been an active research area in modern software development [12][13].

Our previous work [14], has evaluated different traceability and consistency management techniques. We have proposed an extended framework for SAT-Analyzer to be compatible with traceability management and continuous integration for DevOps environments. Another research on managing traceability in self-adaptive systems is presented in [15], which is a generic toolkit with a interlink visualizer for inconsistency detection. Further, considering the software artefacts in later stages of software development with DevOps practices is discussed in [16] providing continuous integration capabilities by using Jenkins.

3. SAT-ANALYSER

3.1 Design Considerations

SAT-Analyser is a traceability management tool capable of tracing software requirement artefact, Unified Modelling Language (UML) class diagram artefact and the Java source code artefact [17][1][18]. Thus, it can be used for traceability in requirements, design and development stages of the SDLC. The system design is shown in Figure 1.

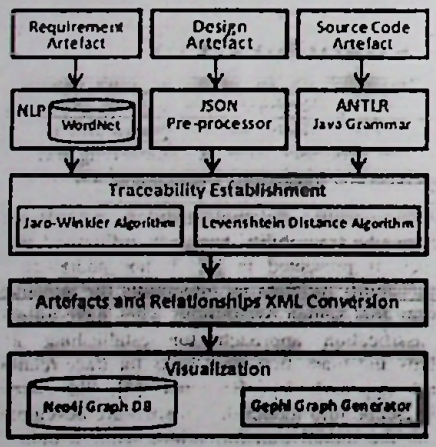

Figure 1. SAT-Analyser system design

SAT-Analyser tool considers three main artefacts; natural language based requirements, UML (Unified Modelling Language) class diagram as design artefact and Java source code for the implementation phase artefact. Initially, data pre-processing techniques are applied for all three types of artefacts, retrieve necessary information and transform them into a common format in XML. Then traceability links are established between the dependent artefacts and visualized the traceability relationships in a traceability graph.

3.2 Workflow: Point of Sale Application

A case study based evaluation is performed using the tool SAT-Analyser. The selected case study is a Point of Sale (POS) system, where a customer can place orders consist of items. An order can be either a special order having the online ordering feature or a normal order having only the cash on delivery facility. These requirements are stated in the software requirement specification in natural language. Figure 2. shows a section of the natural language requirements considered for this study.

The corresponding design in UML class diagram is shown in Figure 3. The main classes are identified as Customer, Order and Item. An Order is specialized into SpecialOrder and NormalOrder. Since the entity Order is composed of set of Item entities, there is an aggregation relationship. Similarly, the association between the entities, Customer class and Order class, is a composition relationship, which is a strong aggregation. Thus, if the Customer entity is deleted, then Order (part) entity is deleted as well.

> In a shop, a customer can place more than one order. An order can have more than one item. Customer details must record the name and location. Item details must record the item number and price. A customer can send and receive the order using the system. The customer can order in two types. Orders are special order and normal order. An order can be confirmed and closed by the customer. The special order can order items online. Normal order can order items in cash on delivery. An item can be added and removed.

Figure 2. POS requirements in natural language

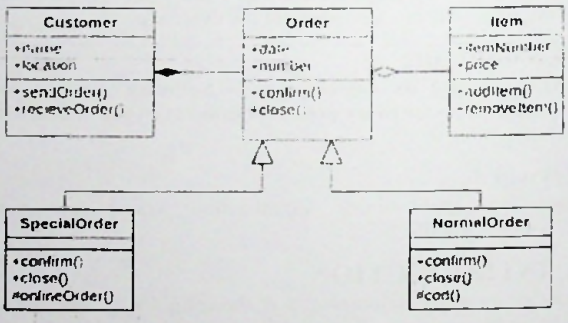

Figure 3. POS UML class diagram

```
import java.util.HashMap;
import java.util.Iterator;
public static void main(String[ ] args) {
  Customer customer = new Customer (name, address);
  ItemManager itemManager = new ItemManager();
  HashMap<String, Item> itemMap = itemManager.getAllItems();
  Iterator<String> availableList = itemMap.keySet().availableList();
  System.out.println("Order Items");
  Order order = new Order (date, number, type);
  while (availableList.hasNext()) {
   Item item = itemMap.get(availableList.next());
   order.addItem(item);
   System.out.println(item.name + " " + item.price");
   }
order.confirm()
 if (order.type == "SpecialOrder"){
   SpecialOrder specialorder = new SpecialOrder(date, number);
   specialorder.onlineOrder();
 }
 else{
   NormalOrder specialorder = new NormalOrder(date, number);
   order.cashOnDelivery();
 }
}
```

Figure 4. POS source code

The relevant source code artefacts are given in Java programing language as a set of class files. A part of the main class file of the source code of this POS system is shown in Figure 4. The corresponding classes for object creation and method calling are implemented separately, and considered as the code artefacts.

Requirements, design and development related artefacts are given as the inputs to the SAT-Analyser in their raw formats, namely requirements document in .txt, UML class diagram in either .mdj or .xmi format and the source code in one or more .java files. Then, SAT-Analyser performs the artefact extraction in the data pre-processing stage. Since the inputs are in three different formats, (1) requirements are processed using the Stanford Core NLP libraries [19] and WordNet lexical database; (2) design class diagram follows a JSON structure at the backend of its .mdj and .xmi formats; (3) source code is processed using the ANother Tool for Language Recognition (ANTLR) [20] Java 8 Grammar to identify the required artefact sub elements. The artefact elements include the requirements, classes, methods, attributes and the relationships inheritance, association and generalization. Next the extracted artefacts are listed and initiate the traceability establishment process. The traces are generated and mapped based on a string comparison as give in Algorithm 1.

Algorithm 1 Traceability link generation

Require: Software artefacts
Ensure: Building relationships among artefacts

```
1.  input: artefacts a
2.  for (a )
3.     get synonyms from WordNet
4.     String comparison for names of classes, attributes,
       methods and relationships
5.        matchDistance = Jaro Winkler algorithm
                          similarity (element1,element2)
6.        If (matchDistance > = 0.8 and < = 1.0)
7.           Build trace link among two artefact elements
8.        Else
9.           editDistance= Levenshtein Distance algorithm
10.                        distance (element1,element2)
11.          matchDistance = 1 - editDistance
12.          If (matchDistance > = 0.8 and < = 1.0)
13.             Build trace link among two artefact elements
14.    XML Writer (nodes, links)
15. output: XML conversion of artefact traceability links
            (Relations.xml)
```

Algorithm 1, handles the pre-processed artifact data towards the traceability link generation. It ensures the relationship building among the extracted artefact elements that are input for the algorithm. Then using the WordNet synonyms and pre-defined dictionary ontology, a string similarity computation is performed using the Jaro-Winkler algorithm [21] and Levenshtein Distance Algorithm [22]. Jaro-Winkler algorithm is selected prominently due to its efficiency than Levenshtein algorithm [23]. The former algorithm considers that, the differences near the start of the strings are more significant than differences close to the end of the strings, while Levenshtein algorithm computes the number of edits needed to convert one string to another. Fixed threshold values are associated for both algorithms and Levenshtein is used for deep comparison if the Jaro-Winkler similarity measure is not in the range of 0.8 and 1.0. Additionally, the WordNet synonym selection is done using the Levenshtein Distance algorithm with a threshold of 0.85.

Consequently, a similarity is marked if either threshold is met by [illegible] the artefact elements. Next, the artefacts and their established trace links are parsed through the Document Object Model (DOM) parser [24] and converted into a predefined XML structure. Figure 5 shows a section of the structure of the generated intermediate XML file for the UML class diagram artefact; Customer and Order class.

```
<?xml version="1.0" encoding="UTF-8"?>
<Artefacts>
 <Artefact type="UMLDiagram">
  <ArtefactElement id="D1" name="Customer" type="Class">
   <ArtefactSubElement id="D1_F1" name="name"
     type="UMLAttribute" variableType="" visibility="public"/>
   <ArtefactSubElement id="D1_F2" name="location"
     type="UMLAttribute" variableType="" visibility="public"/>
   <ArtefactSubElement id="D1_M1" name="sendOrder"
     parameters="" returnType="" status=""
     type="UMLOperation" visibility="public"/>
   <ArtefactSubElement id="D1_M2" name="receiveOrder"
     parameters="" returnType="" status=""
     type="UMLOperation" visibility="public"/>
  </ArtefactElement>
  <ArtefactElement id="D2" name="Order" type="Class">
   <ArtefactSubElement id="D2_F1" name="date"
type="UMLAttribute" variableType="" visibility="public"/>
   <ArtefactSubElement id="D2_F2" name="number"
type="UMLAttribute" variableType="" visibility="public"/>
   <ArtefactSubElement id="D2_M1" name="confirm"
parameters="" returnType="" status="" type="UMLOperation"
visibility="public"/>
   <ArtefactSubElement id="D2_M2" name="close" parameters=""
returnType="" status="" type="UMLOperation" visibility="public" />
  </ArtefactElement>
```

Figure 5. UML Artefact XML file

Accordingly, the classes are considered as the major artefact elements and are given a unique id. The corresponding attributes and the methods are listed as the artefact sub elements for each artefact element with a unique identifier starting with the id of the parent artefact element. For an example, the customer is identified as a class name and the attributes of it are the name and location, while the methods are sendOrder and receiveOrder.

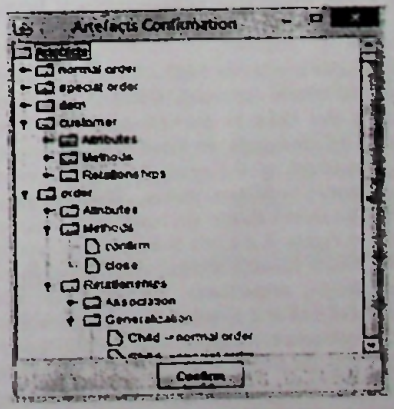

Figure 6. Artefact Extraction Confirmation Window

At the end of these backend data pre-processing, data extraction and traceability establishment, the results are presented in a tree structure by the artefact confirmation option of the tool as shown in Figure 6. The use of the DOM parser is benefitted, since it is capable of loading the full XML documents into a tree structure.

Hence, the user can alter, delete or add any misinterpreted artefact elements prior the confirmation.

The generated intermediate XML files would be modified accordingly and soon after the traceability project is created to the user. Afterwards, all these set of XML files are converted into an array format that follows a key-value pair structure using DOM parser and the Simple API for XML (SAX) parser's exception handling capabilities [25] to store in the Neo4j graph database [26]. Then the open graph visualization platform Gephi [27] is used for the graph generation using the nodes and links stored in Neo4j. Consequently, the SAT-Analyser visualizes the traceability links among artefacts or any selected artefact sub elements. The set of visualization filtering are as follow.

- Full graph view with artefacts and their links.
- Edge filtered view for the relationship among the identified classes, attributes, operations for each of the artefact in requirements, design and code.
- Artefact filtered views for each one of 3 artefacts separately.

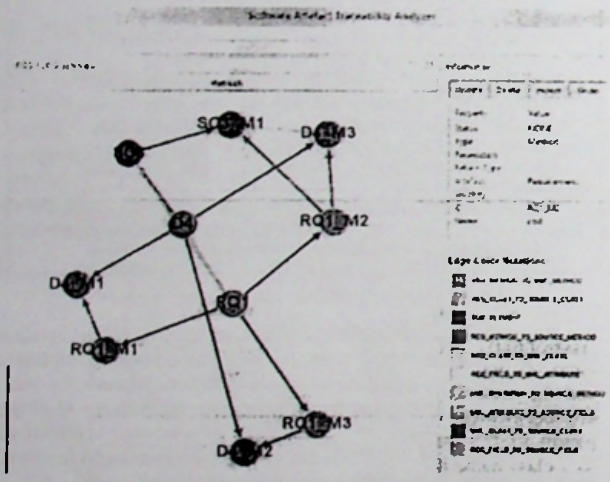

Figure 7. Full graph view of traceability in POS system

Figure 7 illustrates a selected section of the obtained full graph view of this POS system case study. Color codes are used for each type of nodes and links in the representation. Moreover, the details of each selected node are listed in the information section separately. The length of the edges denotes the strength of the similarity between each two nodes. Thus, larger the string comparison value means shorter the length of corresponding edge. For example, in Figure 7, the edit distance value among RQ1 and D4 is 0.916 which denotes normal order class in requirement artefact and design, respectively. Similarly, the value among RQ1_M2 and D4_M3 is 1.0, which represents cash on delivery method in requirements artefact and UML design artefact, respectively. Thus, the length of the edge between RQ1 and D4 is bit lengthy as the UML class diagram artefact has used the class name with naming conventions.

4. EVALUATION

The evaluation of the applied POS system is conducted using correctness measures based on the artefact, relationship extraction shown in Figure 6, since proper artefact and relationship identification is crucial towards the final traceability outcomes. Accordingly, the metrics precision and recall are applied as information retrieval accuracy measurements [28]. The artefact and relationship extraction results are evaluated as follows.

$$\text{Artefact, relationship extraction precision} = \frac{\text{number of correctly identified artefacts, relations}}{\text{total numberofidentified artefacts, relations}}$$

Similarly, the recall is measured as follows.

$$\text{Artefact, relationship extraction recall} = \frac{\text{number of correctly identified artefacts, relations}}{\text{total numberofactual artefacts, relations}}$$

Moreover, F-measure (F1 score), which is the weighted average of the obtained precision and recall: is derived as follows.

$$F1 = 2\frac{\text{prescion} \cdot \text{recall}}{\text{prescion} + \text{recall}}$$

Table 1. Evaluation of traceability support techniques

Traceability Establishment	Artefact	Precision	Recall	F-Measure
Requirement – Design	Class	1	0.8	0.8
	Attribute	1	0.5	0.6
	Method	1	0.5	0.6
Design – Source code	Class	1	1	1
	Attribute	1	0.3	0.4
	Method	0.8	0.5	0.6
Requirement – Code	Class	1	1	1
	Attribute	1	0.6	0.7
	Method	1	0.6	0.7

Traceability establishment accuracy among similar artefacts in different phases of the SDLC is shown in Table 1. The precision denotes positive results for the generated trace links, while the lower recall signifies that there exist missing links among attributes and methods. It is observed that the inaccurate artefact elements extraction and identification with NLP that contain different naming conventions and less meaningful names in requirement artefacts, have led to the lack of accuracy. However, the overall F-measures are biased towards 1 and requirement to code traceability has shown a high accuracy.

5. CONCLUSION

Software traceability is essential to ensure the proper synchronization among software artefacts during the software development process. There exist various software traceability related solutions; however most of them have certain limitations. SAT-Analyser tool presented in this paper is one such tool support software requirement, design and source code artefacts. This paper highlights the accuracy of the traceability establishment process of SAT-Analyser tool using a POS based application.

Requirement, design and development related artefacts in their raw formats are fed to the tool as text, UML class diagram file and Java source code files, respectively. SAT-Analyser pre-processed the input data and extracts the relevant artefact elements. The traceability links among the artefacts are established based on a similarity calculation algorithm. Moreover, the traceability relationships are visualized using traceability graphs for developer decision making. The tool allows manual artefact trace alterations and updates the graphs accordingly.

SAT-Analyser is evaluated using the accuracy measures precision, recall and F-measure based on the established traceability links among artefacts in the considered case study. Significant positive

results have been obtained and identified possible improvements [illegible] establishment algorithm. Furthermore, the integration of continuous-integration support for the tool-with DevOps principles would be an important future work to cope with the agile based software development environments.

6. ACKNOWLEDGMENTS

The author acknowledges the support received from the Senate Research Committee Grant SRC/LT/2016/07, University of Moratuwa, Sri Lanka in publishing this paper.

7. REFERENCES

[1] Kamalabalan, K. et al. 2015. Tool Support for Traceability of Software Artefacts. In *Proceedings of the Moratuwa Eng. Research Conf. (MERCon)*. (2015). IEEE. 318-323.

[2] Satish, C. J. et al. 2016. A Review of Tools for Traceability Management in Software Projects. *Int. Journal for research in emerging science and technology*. 3, 3 (2016), 6–10.

[3] Mader, P. and Gotel, O. 2012. Towards automated traceability maintenance. *Journal of Systems and Software*. 85, 10 (2012), 2205–2227.

[4] Hazeline U. Asuncion. 2008. Towards practical software traceability. In *Companion of the 30th international conference on Software engineering* (ICSE Companion '08). ACM, NY, USA, 1023-1026.

[5] Grammel, B. and Kastenholz, S. 2010. A generic traceability framework for facet-based traceability data extraction in model-driven software development. In *Proceedings of the 6th ECMFA Traceability Workshop* (ECMFA-TW '10). ACM, NY, USA, 7-14.

[6] Parizi, R. M. On the gamification of human-centric traceability tasks in software testing and coding. In *Proceedings of the 2016 IEEE 14th Int. Conf. on Software Eng. Research, Management and Applications* (SERA). IEEE, 193–200.

[7] Qusef, A. et al. 2011. SCOTCH: Test-to-code traceability using slicing and conceptual coupling. In *Proceedings of the 2011 27th IEEE Int. Conf. on Software Maintenance* (ICSM), IEEE, 63–72.

[8] Cleland-Huang, J. et al. 2014. Achieving lightweight trustworthy traceability. In *Proceedings of the 22nd ACM SIGSOFT Int. Symposium on Foundations of Software Eng.* (FSE 2014). ACM, NY, USA, 849-852.

[9] Klock, S. et al. 2011. Traceclipse: an eclipse plug-in for traceability link recovery and management. In *Proceeding of the 6th Int. workshop on Traceability in emerging forms of Software Eng.* (TEFSE '11). ACM, NY, USA, 24-30.

[10] Mills C. Automating traceability link recovery through classification. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Eng.* (ESEC/FSE 2017). ACM, NY, USA, 1068-1070.

[11] Noll, R. P. and Ribeiro, M. B. Enhancing traceability using ontologies. In *Proceedings of the 2007 ACM symposium on Applied computing* (SAC '07). ACM, NY, USA, 1496-1497. DOI=http://dx.doi.org/10.1145/1244002.1244322.

[12] Cleland-Huang, J. Traceability research: taking the next steps. In *Proceeding of the 6th Int. workshop on Traceability in emerging forms of Software Eng.* (TEFSE '11). ACM, NY, USA, 1-2.

[13] Poshyvanyk, D. et al. 6th Int. workshop on traceability in [illegible] *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11). ACM, NY, USA, 1214-1215.

[14] Rubasinghe, I. D. et al. 2017. Towards Traceability Management in Continuous Integration with SAT-Analyser. In *Proceedings of the 3rd Int. Conf. on Communication and Information Processing*. (2017). ACM, Tokyo.

[15] Perera, I. et al. 2015. A Traceability Management Framework for Artefacts in Self-Adaptive Systems. In *Proceedings of the 10th Int. Conf. on Industrial and Information Systems(ICIIS)*. (2015). IEEE. 37–42.

[16] Palihawadana, S. et al. 2017. Tool support for traceability management of software artefacts with DevOps practices. In *Proceedings of the Moratuwa Eng. Research Conf. (MERCon)*. (2017). IEEE. 129-134.

[17] Wijesinghe, D.B. et al. 2014. Establishing traceability links among software artefacts. In *Proceedings of the 14th Int. Conf. on Advances in ICT for Emerging Regions*. (2014). IEEE. 55–62.

[18] Arunthavanathan, A. et al. 2016. Support for traceability management of software artefacts using Natural Language Processing. In *Proceedings of the 2nd Int. Moratuwa Eng. Research Conf. (MERCon)*. (2016). IEEE. 18–23.

[19] Manning, C. D. et al. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Baltimore, Maryland. 55–60.

[20] ANTLR: *http://www.antlr.org/*. Accessed: 2017-07-21.

[21] JaroWinklerDistance (LingPipe API): *http://alias-i.com/lingpipe/docs/api/com/aliasi/spell/JaroWinklerDistance.html*. Accessed: 2017-08-14.

[22] Efficient Implementation of the Levenshtein-Algorithm, Fault-tolerant Search Technology, Error-tolerant Search Technologies: *http://www.levenshtein.net/*. Accessed: 2017-10-14.

[23] Christen P. 2006. A Comparison of Personal Name Matching: Techniques and Practical Issues. In *Proceedings of the IEEE Sixth Data Mining Workshop* (ICDM '06). IEEE, Hong Kong, China.

[24] Le Hors, A. et al. 2004. *Document Object Model (DOM) Level 3 Core Specification*. W3C Technical Report. Massachusetts Institute of Technology, Cambridge, MA.

[25] Parser: *http://www.saxproject.org/apidoc/org/xml/sax/Parser.html*. Accessed: 2017-10-15.

[26] Graph Visualization for Neo4j: Tools, Methods and More: *https://neo4j.com/developer/guide-data-visualization/*. Accessed: 2017-07-23.

[27] Gephi - The Open Graph Viz Platform: *https://gephi.org/*. Accessed: 2017-10-14.

[28] Zeugmann T. et al. 2011. Precision and Recall. In *Encyclopedia of Machine Learning*. Springer US, Boston, MA, 781–781.