Journal of Software Engineering

ISSN 1819-4311

Academic Journals Inc.

www.academicjournals.com

OPEN ACCESS Journal of Software Engineering

ISSN 1819-4311
DOI: 10.3923/jse.2017.282.289

Research Article
Requirements Traceability Model Generation for Evolving Software Systems

Milu Mary Philip, Rahul Varma and Vijayakumar Balakrishnan

Department of Computer Science, Dubai Campus, Birla Institute of Technology and Science, Pilani, Dubai, UAE

Abstract

Background and Objective: A software application system can undergo changes during one or more phases of the software development lifecycle. Managing the changes in the requirements is an expensive task. The different variation points of the system should be described and followed right from the design stage to its implementation. This study aimed to develop a generalized requirements traceability model, which provides a clear insight into the relationship between each feature and the requirements and also explains the inbound and outbound dependencies between each of the features. **Methodology:** The user provided inputs and their dependencies with the help of a graphical user interface. These inputs were validated for correctness and if there were any conflicts, the user will be notified of the same. The present study focused on handling two types of conflicts namely, transitive and symmetric. **Results:** The results showed the dependency traceability model for conflict free user inputs. The various features and their dependencies were highlighted in the model generated. This model was considered as a promising and stable model to analyze the order in which the features were processed. In case of any conflicts in the user entry, valid error messages will be sent back to the user and the traceability model will not be generated. **Conclusion:** This model acted as a baseline for the software architect, since it checked the feasibility of the variation points in depth. The developer can refer this model for the implementation of the application system. This model has been validated successfully for an image processing application.

Key words: Architectural pattern, architectural specifications, image processing, software architecture, software requirement specifications, traceability models

Received: April 11, 2017 Accepted: May 25, 2017 Published: June 15, 2017

Citation: Milu Mary Philip, Rahul Varma and Vijayakumar Balakrishnan, 2017. Requirements traceability model generation for evolving software systems. J. Software Eng., 11: 282-289.

Corresponding Author: Milu Mary Philip, Dubai Campus, Birla Institute of Technology and Science, Pilani, Dubai, P.O. Box 345055, DIAC, Dubai, UAE
Tel: 00971556605818

Competing Interest: The authors have declared that no competing interest exists.

Data Availability: All relevant data are within the paper and its supporting information files.

INTRODUCTION

According to the ANSI/IEEE Standard 830-1984[1], a Software Requirements Specification (SRS) is traceable if the origin of each requirement is clear and if it facilitates the referencing of each requirement in future development or enhanced documentation. The details regarding the software application system-like the various components and the way they are connected is clearly documented in the SRS document. The SRS acts as a foundation for the design of the application system.

One of the major factors in developing a software system with high quality depends on its capacity to trace the requirements right from the initial stages of design and specification to the final stages of its implementation and testing[2]. Refinements to the requirements of a software application system can be brought in at any of these stages in the software development lifecycle. An increase in the changes that are to be made to the system symbolizes a greater need for traceability models[3]. The maintenance of the system is made easier using the requirements traceability models. It maintains valid links between the items developed during the process of software development[4]. It provides a clear relationship between the requirements of the stakeholder and the artifacts produced during the software development lifecycle[5]. The requirements of an application system will be related to each other and hence any modifications to a feature might affect all the other interdependent features. Hence, the maintenance of the system will be a tedious task. For the proper maintenance of an application system, it is quite essential that the system should be implemented taking into consideration the traceability mechanisms[6].

Requirements traceability is proved to be an effective technique for the software application systems, where the failure of the system cannot be tolerated. It allows the developer to understand and account for the consequences of the modifications in the system[3]. The present study deals with the development of dependency traceability model for the user inputs provided through the user interface. This model provided the order of processing of the features. The variation points (various features and their order of processing) are traced using the traceability model generated. The traceability model acted as the baseline for the software architect, to develop the architectural specification for the various components and connectors to be present in the system. Moreover, if there are any changes in the requirements, all the affected features or modules can be identified from the model and hence the maintenance of the software system will be made easier[4]. The present study considers the image processing application scenario.

The information related to few operational terms that are referred to in this study are given as:

- **Features:** They refer to the various image processing modules that are provided by the user for the processing of input images. As an example, crop, rotate, resize refers to the features in the image processing application scenario
- **Order of processing:** It refers to the sequence of operations for the various modules in the application scenario
- **Traceability model:** It shows how the requirements of the application system are related to one another
- **Variation point:** It represents the points in the architecture or source code where the different variants can be inserted into

In this study, the different features and their order of processing were considered as the variation points.

An overview of the existing works carried out in the area of requirements traceability was discussed. The success of an application system was determined by the ease by which it was able to evolve and continue to address the ever-changing needs of the market[7]. The modifications or variations to the system is a key requirement to which the system has to adapt itself. Variation points are defined as the places in the architecture or the source code where the different variants can be inserted into. The variation points that we consider in our work deals with the various filters for the processing of the images and their order of processing.

The different variation points were tracked right from the design stage to its testing stage, so that the changes are made visible to all the team members involved in software development. The traceability of software requirements was very important aspect in supporting the various activities in the software development lifecycle[8]. It helps greatly in maintaining the quality of the software application system. It allows the different users to better understand the system and its functioning and also helps in making up a clear and consistent documentation of the application system. It provided the relationships between the various software artifacts like the test models, source code and requirements specifications. These relations can be used to assist in merging the modifications to the system during the development or

maintenance of a software system. The traceability maintained a mesh of dependencies between the various artefacts of the software system[9].

The traceability of the requirements can be defined as the ability to describe and follow the life of a requirement in both forward and backward direction, right from its origin to its implementation[10]. The requirements traceability allows the programmers to understand and account for the consequences of the refinements made to the system[3]. An increase in the refinements to be made to the system, implies the need for traceability. One of the early study by Ramesh[11], clearly mentioned the need, techniques and deployment of traceability for the development of software systems. Marlowe and Kirova[9] pointed down the benefits of traceability to the product organization and to the stakeholder. The requirements for traceability improved the quality of the product by ensuring that the product is being implemented according to the stakeholder needs. The requirements were tracked right from its design stage, to its implementation, testing and its final delivery to the stakeholder[12].

Requirements traceability commanded to attain a successful requirement management process. The work clearly explains how Unified Modelling Language (UML) model elements can be used to provide a framework for traceability[10]. The UML is used in the work to characterize a standard framework to support requirements and modeling specifications. The textual specifications are integrated with the standard UML specifications. Hammad *et al.*[13] proposed a mechanism using which it can be identified whether a change in the source code will affect the design of the software system. The traceability from the code to its design is maintained throughout the development of the application system. The design is commonly documented and described using UML class diagrams. Mader *et al.*[14] suggested that the modifications to the UML elements can be made possible by adding new components, deleting existing components or extending the components that are already available in the system. The requirements are considered as nodes of a directed acyclic graph. They represent elements in different models and the link between the components represent the traceability relations. A set of possible types of changes and the required modifications to the existing traceability relations are also discussed in this study.

The traceability between the source code of a system and its architecture makes it easier for the developer to understand the system design and hence makes the maintenance of the system much more simpler[6]. With each stage in the evolution of the software system, the implementation and the architecture of the application system keeps on updating. Hence, the traceability links between them is not consistent. The traceability link should be updated with each modification, so that it is made easy for the maintenance of the application system. The validation and verification of the application system can also be performed using the traceability links. A framework for managing the traceability with the development of the application system is built up by using trace meta models[15]. Traceability between feature models and architecture models can be formed by identifying the mapping between the two models used the formal concept analysis technique[16]. The matching elements in each of the models were originated using the functional decomposition of each models. The features Model driven engineering is a unique perspective to the development of an application system as it emphasizes on the use of models for application development[17]. These models can be used for the traceability of the various features from the requirement phase to its implementation and also to check whether the application system meets the quality standards. Traceability can be strengthened with machine learning techniques. Thomas *et al.*[18], in their study, proposed a method that integrates traceability with topic modeling (machine learning technique). It records the traceability links during the application system development and learns a probabilistic topic model over the various software artifacts which is used for the classification of the various software artifacts.

Currently, there are few tools which provide the necessary traceability relationships. ArchTrace[19,20], is one such tool that emphasizes on synchronizing the architecture and its code. MolhadoArch[21] is another tool which used the configuration management approach to traceability. It models the architecture as a graph, captures the various connections and their interconnections, where each node represents a structural unit in the code and the traceability relationships are represented using the edges. Traceability links between the code and architectural tactics can be built up using the tactic traceability information models[22]. This model uses machine learning and information retrieval methods to train a classifier that points out the tactic related classes. Another study by Ubayashi and Kamei[23], uses architecture mappings and architecture points to assist traceability between code implementation and architecture.

Umar and Khan[24] proposed a meta model where the non-functional requirements are also being traced along with the functional requirements through the complete software development lifecycle. Tracing a change applied to a non-functional requirement in the traceability model is being discussed in this study. This study captured the

functional and non-functional requirements and their relations throughout the software development process. The non-functional requirements has various interdependencies between the different components present in the application system[25]. Drivalos *et al.*[26] in their study presented a traceability meta modelling language which deals with the development and maintenance of the traceability meta models.

The existing works in the area of requirements traceability have been pointed out here. The traceability model that is generated by this study, helps in easily identifying the relationship between each feature. The order of processing can be derived from the traceability model. The variation points considered in this study include various features and their order of processing are traced from its design stage, to the implementation and testing stage.

The specific objectives of this study were stated as follows:

- Develop the user interface that is required to allow the user to input the required features and their dependencies
- Generate the dependency traceability model for input entered by the user
- Test the above model for the image processing application system

MATERIALS AND METHODS

The development of the requirement traceability models consisted of the user interface. The user provided the list of features and their order of processing through the user interface. The user inputs were validated for correctness and conflicts if any and notified to the user. Any type of conflicts in the user input was not allowed in the application system. There are two types of conflicts that have considered in our algorithm:

Transitive conflicts: The user enters 3 features F_1, F_2, F_3 and its dependencies in the user interface. The order of processing of the features is given as:

$$F_1 \rightarrow F_2 \rightarrow F_3$$

'→' implies the 'Depends on' relationship.

Thus, it can be inferred from the above relationship that:

- F_2 depends on F_1
- F_3 depends on F_2

Based on the transitive property, the above given relationship implies that F_3 depends on F_1 which means that F_3 has a transitive dependency on F_1. Thus, F_3 can be done only if F_1 finished its processing. Hence, the entry F_1 depends on F_3, when specified by the user, creates a conflict and hence the traceability model will not be generated and the user will be notified of the same.

Symmetric conflicts: The user enters 2 features F_1 and F_2. The order of processing is given as:

$$F_1 \rightarrow F_2$$

It is inferred that F_2 depends on F_1, which implies that F_1 can not depend on F_2.

It means that F_1 will be processed first and the output will be provided as input to F_2 for its processing. Thus, the entry F_1 depends on F_2 when specified by the user, will be considered as a conflict and the user will be notified of the same. The traceability model for this scenario will not be generated.

The inbound and outbound dependency relationship between the features will be represented, for all valid user inputs.

The algorithm shown below clearly explains the steps to follow foe the generation of the dependency traceability model.

Algorithm: DEPENDENCY_MODEL_GEN

An algorithm for the generation of the dependency traceability model described below. It was validated with an image processing application system:

1. **Input:** Required entries to be made by the user in the user interface
2. **Output:** Dependency traceability model
3. **Procedure:**

    1. In the user interface, select the number of features, the feature names and the required order of processing as shown in the user interface (UI)
       Action:
       Step 1.1: Enter the number of features in the UI
       Step 1.2: Enter the names of the features in the space provided in the interface
       Step 1.3: Enter the dependencies for each of the selected features and then click the Save button
    2. Check for the conflicts in the dependencies entered by the user, using the algorithm check_conflict as discussed
       Action:
       Step 2.1: Based on the list of dependencies that the user has entered for each feature, a transitive dependency list is created using the algorithm transitive_List_Generation (Fig. 1)
       Step 2.2: Check for the symmetric conflicts in the updated list of features
       Step 2.3: If there are conflicts in the user entry, the user is notified about it by the Check_Conflict algorithm and the system requests the user to modify his entry (step 1.3 described above)
       Step 2.4: If the Check_Conflict algorithm does not find any conflicts, proceed to the generation of traceability model
    3. The traceability model for the user entry would be generated

```
Begin
List = [(f1,f2), (f2, f3)...(f_{i-1},f_i),(f_{n-1}, f_n)
features_List = set(List)
trans_Value = empty set
while true
{
      for all (x, y) in feature_List
          for all (q, w) in feature_List
                if(q == y)
                    trans_Value = set (x, w)

          new_FeaturePair = feature_List ‖ trans_Value
          if (feature_List == new_FeaturePair)
                break
}
feature_List = new_FeaturePair
returns list (feature_List)
End
```

Fig. 1: Transitive list generation algorithm

Algorithm-TRANSITIVE_LIST_GENERATION

The algorithm in Fig. 1 describe the generation of the transitive dependencies from the order of processing entered by the user:

- **Input:** User_List-ordered list of features according to the order of processing provided by the user
  'n'-Number of features
- **Output:** Feature_List-the transitive dependency list of features

The algorithm starts with an input list of feature pairs entered by the user and given as (f_{i-1}, f_i). Each feature pair was compared with every other pair in the feature_List and if there was a transitive property between the pairs, then the transitive pair of features were updated to trans_Value. The list of features was updated with the new feature pair list. This was repeated till the new feature pair list is same as the list of feature pairs in the previous iteration. The initial list of dependencies of features (as provided by the user) was updated with a new list of transitive dependencies and returned to the Check_Conflict algorithm as shown in Fig. 2, where it was checked for any type of conflicts.

Algorithm-CHECK_CONFLICT

- **Input:** List with the order of processing of features provided by the user. The user entered features were placed in a matrix F[i][j]
- **Output:**
  1. If TRUE-signifies that there are no conflicts in the user input and hence the dependency traceability model will be generated
  2. If FALSE-signifies a conflict and hence the user will be notified of the same and would be asked to modify his input

The user entered features, (in the same order as the user selects it in the interface) are listed across the rows and columns of a matrix F[i][j], with 'i' representing the rows and 'j' representing the columns of the matrix.

```
Begin
    trans_List = transitive_FeatureList (list)
    newList = []
    for (x, y) in trans_List
        v1, v2 = x, y
        if (i>j)
            new List.append (v2, v1)
        else
            newList.append (v1, v2)
    if (len (set (newList)) ! = len (set(trans_List))
        return False
    return True
End
```

Fig. 2: Check_Conflict algorithm

The algorithm shown above examines conflicts in the user entered dependencies. The updated list of feature pairs, provided by the transitive_List_Generation algorithm, was checked for both transitive as well as symmetric conflicts. With respect to the matrix F[i][j], each feature pair in this list, was assigned a row index ('i') as well as a column index value ('j'). The list (newList) was created, where the feature pairs were appended based on the row and column index values of the pairs. The length of the newList is compared with that of the transitive list. If there was a match, then the algorithm returns TRUE. If the algorithm returns true, it means that there is no conflict in the dependencies entered by the user and therefore it generates the dependency traceability model. In this model, the rows and columns correspond to the different features present in the application system. The resultant traceability model can then be used by the developer to understand the different variation points of the system and design the system accordingly. The architectural design of the system will be designed in such a way that it can handle all the variations.

User interface: The user interface which would be available to the user, so that the user can enter the required features and their order of processing described in this study.

The user interface as shown in Fig. 3, will have an option to enter the required number of features and their names. The bottom section of the user interface provides the space to enter the order of processing of the selected features using the drop-down menu. Once the user enters all the features and their dependencies, the user can click the Save button.

The algorithm and UI have been successfully tested for four different scenarios and are explained.

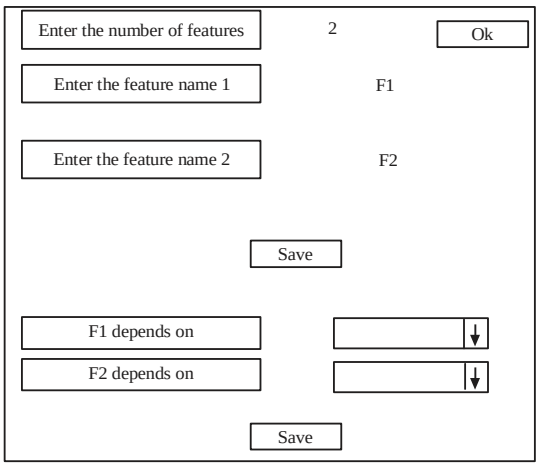

Fig. 3: User interface for traceability model generation

Experimental setup: The traceability model was developed and tested on a Linux server with UBUNTU 14.04 operating system. The implementation was carried out using the high-level programming language Python. It supports object oriented, functional and procedural styles of programming with a standard library for the UI development.

RESULTS

The proposed model is validated with an image processing application system, although the model is generic for any application. The user specific entries have been provided through the UI is shown in Fig. 3.

Test scenario 1: Valid input with 3 features: The user enters the number of features as 3. The features that was selected are crop, rotate, resize. The order of processing given as:

- Crop depends on rotate
- Rotate depends on resize

The Check_Conflict algorithm was run, which checks for the conflicts in the feature list that was created by the transitive_List_Generation algorithm. Hence, the final list of feature pairs for this case would be:

Feature_List = [(crop, rotate), (rotate, resize),(crop, resize)]

This list would be sent to the Check_Conflict algorithm which will look for the possible conflicts. As there were no conflicts, it returns TRUE and hence the traceability model for the above entry generated is shown below in Table 1.

Table 1: Traceability model for 3 features

Feature_Names (i = 0, j = 0)	Crop (i = 0, j = 1)	Rotate (i = 0, j = 2)	Resize (i = 0, j = 3)
Crop (i = 1, j = 0)	-	********	-
Rotate (i = 2, j = 0)	-	-	********
Resize (i = 3, j = 0)	-	-	-

The dependencies (as entered by the user) in this scenario are highlighted in the traceability model that was generated. Each '********' shows a dependency between the feature in the row and the feature in the column. It is implied that the feature in the row 'depends on' the feature in the column. The '-' in the traceability model implies that there was no dependency between the features

The order of processing of the images can be easily examined from the model generated in Table 1. As there was no dependency for the Resize (the last row in the traceability model has no entry), it is concluded that Resize feature should be performed first, followed by Rotate and finally Crop.

Test scenario 2: Input with transitive conflict: In this scenario, the user enters 3 features-crop, rotate and resize. The order of processing of the different features given as:

- Crop depends on Rotate
- Rotate depends on Resize
- Resize depends on Crop

The transitive_List_Generation algorithm when run with the above entries, generates the updated list as:

- Feature_List = [(Rotate, Rotate), (Resize, Crop)
- (Resize, Rotate), (Crop, Resize)
- (Rotate, Resize), (Crop, Crop)
- (Crop, Rotate), (Resize, Resize), (Rotate, Crop)]

The above shown Feature_List, with its length equal to 9, provided as input to the Check_Conflict algorithm so that it will check the possibilities of conflicts in the different combinations of image processing features. The newList created in this algorithm will have a length of 6. This was because all the symmetric entries has been removed and only unique pairs has been chosen to the newList. Due to this difference in length, it was concluded that there were conflicts in the user entry and the algorithm returns FALSE. Hence the traceability model will not be generated in this scenario and the user will be notified of the same with a message as shown below:

ERROR_MSG: CONFLICTS IN USER INPUT

Test scenario 3: Input with symmetric conflict: The user enters 2 features-crop and rotate. The order of processing is:

- Crop depends on Rotate
- Rotate depends on Crop

The transitive list generation creates the list of features as:

- Feature_List =[(rotate, rotate),(crop, crop)
- (crop, rotate),(rotate, crop)]

This scenario is an example for symmetric conflicts. The length of the Feature_List will be 4 and the newList length would be 3. Hence the algorithm returns FALSE and the traceability model will not be generated. The user will be notified of the conflict with an error message:

ERROR_MSG: CONFLICTS IN USER INPUT

DISCUSSION

In this study, the traceability model was generated for user entries that do not involve any conflict. The different components to be present in the application system as well as the order in which the components were to be processed, can be easily perceived from the generated traceability model. The conflicts in the user input were checked with the Check_Conflict algorithm described in this study. Moreover, the maintenance of the system can be made easier with traceability model. Whenever a change (either the features or their order of processing) was made to the system, it was evident from the traceability model as to which all features would be affected with the proposed change. Hence, all those affected features would be updated accordingly. The earlier studies in the area of traceability[19-21], do not directly consider the conflicts, if any, in the user entries. The algorithm proposed by this study, generates the traceability model for conflict free input requirements. In the case of invalid user entries, suitable error messages were generated and notified to the user.

The traceability model that was generated, traces the requirements in the design stage which forms the basis for the implementation of the application system. For every change in the variation points, the traceability model will be updated accordingly and hence the developer can refer this model for the successful implementation of the application system. The test cases for the testing phase can also be developed, with the help of the model generated. All the affected features that have to be tested can be acquired from the traceability model. Thus, the development, maintenance and testing of the application system would have been a tough task, without the dependency traceability model of the variation points.

CONCLUSION AND FUTURE RECOMMENDATIONS

This study focused on the generation of the traceability models, which tracks the variation points. If there were no conflicts in the modifications introduced to the variation points, the traceability model shows the features and its inbound and outbound dependencies were generated. This model helps in the assessment of the impact of the change on the various features and their dependencies. All the dependent features will have to be modified, for the proper working of the application system. This acts as a good reference model for software architect, since it explores the variation points and their feasibilities in depth. The traceability model was generated successfully for the image processing application system. The present study will significantly helps a software architect in the development of the architectural specification of an application system with changing requirements.

SIGNIFICANCE STATEMENTS

This study described the possible variation points of the application system and tracks them at all stages of the software development lifecycle. The traceability model thus developed as an outcome of the present study also explained the inbound and outbound dependencies between each of the features. This study will help the researcher to validate the user inputs for correctness. The traceability model is generated only for conflict free user inputs. The earlier studies in this area, do not directly consider the conflicts, if any, in the user entries. Thus, it acts as a good reference model to develop the architectural specification for the application system, since it explores the variation points and their feasibilities in depth.

ACKNOWLEDGMENTS

The authors would like to express their sincere gratitude to all the members of the Department of Computer Science the Birla Institute of Technology and Science-Pilani, Dubai Campus for their support in this study.

REFERENCES

1. IEEE., 1984. IEEE Guide to Software Requirements Specifications: ANSI/IEEE Standard 830. IEEE., USA.
2. Ghabi, A. and A. Egyed, 2015. Exploiting traceability uncertainty among artifacts and code. J. Syst. Software, 108: 178-192.

3. Jane, C.H., O. Gotel and A. Zisman, 2012. Software and Systems Traceability. Vol. 2, Springer, Heidelberg, pp: 7-8.
4. Zogaan, W., I. Mujhid, J.C. Santos, D. Gonzalez and M. Mirakhorli, 2016. Automated training-set creation for software architecture traceability problem. Empirical Software Eng. 10.1007/s10664-016-9476-y
5. Anquetil, N., U. Kulesza, R. Mitschke, A. Moreira, J.C. Royer, A. Rummler and A. Sousa, 2010. A model-driven traceability framework for software product lines. Software Syst. Model., 9: 427-451.
6. Lelis, C.A.S., J.F. Tavares, M.A.P. Araujo and J.M.N. David, 2016. GiveMe trace: A software evolution traceability support tool. IEEE Latin Am. Trans., 14: 3444-3454.
7. Breivold, H.P., I. Crnkovic and M. Larsson, 2012. A systematic review of software architecture evolution research. Inform. Software Technol., 54: 16-40.
8. Winkler, S. and J. Pilgrim, 2010. A survey of traceability in requirements engineering and model-driven development. Software Syst. Model., 9: 529-565.
9. Marlowe, T.J. and V. Kirova, 2009. High-level component interfaces for collaborative development: A proposal. J. Syst. Cybern. Inform., Vol. 7.
10. Jirapanthong, W. and A. Zisman, 2009. Xtraque: Traceability for product line systems. Software Syst. Model., 8: 117-144.
11. Ramesh, B., 1998. Factors influencing requirements traceability practice. Commun. ACM., 41: 37-44.
12. Torkar, R., T. Gorschek, R. Feldt, M. Svahnberg, U.A. Raja and K. Kamran, 2012. Requirements traceability: A systematic review and industry case study. Int. J. Software Eng. Knowl. Eng., Vol. 22. 10.1142/S021819401250009X
13. Hammad, M., M.L. Collard and J.I. Maletic, 2011. Automatically identifying changes that impact code-to-design traceability during evolution. Software Qual. J., 19: 35-64.
14. Mader, P., O. Gotel and I. Philippow, 2009. Enabling Automated Traceability Maintenance Through the Upkeep of Traceability Relations. In: European Conference on Model Driven Architecture-Foundations and Applications, Paige, R.F., A. Hartman and A. Rensink (Eds.)., Springer, Berlin, Heidelberg, pp: 174-189.
15. Haidrar, S., A. Anwar and O. Roudies, 2016. Towards a generic framework for requirements traceability management for SysML language. Proceedings of the 4th IEEE International Colloquium on Information Science and Technology (CiSt), October 24-26, 2016, IEEE., pp: 210-215.
16. Diaz, J., J. Perez and J. Garbajosa, 2015. A model for tracing variability from features to product-line architectures: A case study in smart grids. Requirements Eng., 20: 323-343.
17. Unterkalmsteiner, M., T. Gorschek, A.M. Islam, C.K. Cheng, R.B. Permadi and R. Feldt, 2012. Evaluation and measurement of software process improvement: A systematic literature review. IEEE Trans. Software Eng., 38: 398-424.
18. Thomas, S.W., B. Adams, A.E. Hassan and D. Blostein, 2014. Studying software evolution using topic models. Sci. Comput. Program., 80: 457-479.
19. Murta, L.G., A. van der Hoek and C.M. Werner, 2006. Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, 18-22, 2016, IEEE., pp: 135-144.
20. Murta, L.G., A. van der Hoek and C.M. Werner, 2008. Continuous and automated evolution of architecture-to-implementation traceability links. Autom. Software Eng., 15: 75-107.
21. Yusop, O.M. and S. Ibrahim, 2011. Evaluating software configuration based approaches tosupport change maintenance of granular software artefacts. Int. J. Inform. Elect. Eng., 1: 185-189.
22. Mirakhorli, M. and J. Cleland-Huang, 2016. Detecting, tracing and monitoring architectural tactics in code. IEEE Trans. Software Eng., 42: 205-220.
23. Ubayashi, N. and Y. Kamei, 2012. Architectural point mapping for design traceability. Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages, March 26, 2012, ACM., Germany, pp: 39-44.
24. Umar, M. and M.N.A. Khan, 2012. A framework to separate non-functional requirements for system maintainability. Kuwait J. Sci. Eng., 39: 211-231.
25. Mahmoud, A. and G. Williams, 2016. Detecting, classifying and tracing non-functional software requirements. Requirements Eng., 21: 357-381.
26. Drivalos, N., D.S. Kolovos, R.F. Paige and K.J. Fernandes, 2008. Engineering a DSL for Software Traceability. In: International Conference on Software Language Engineering, Gasevic, D., R. Lammel and E. Van Wyk (Eds.). Springer, Berlin, pp: 151-167.