AN ONTOLOGY-DRIVEN VISUALIZATION TOOL FOR SOFTWARE REQUIREMENTS TRACEABILITY MATRIX AND TEST DOCUMENTS

Padungpong Khuiarphai, Twittie Senivongse
Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University
254 Phyaithai Road, Pathumwan, Bangkok 10330 Thailand
padungpong.k@student.chula.ac.th, twittie.s@chula.ac.th

ABSTRACT
Software quality is one of the factors that affect the acceptance of software products and it is often assessed in terms of defects found in the software and whether the software can meet specified user requirements. It is hence a valuable practice within software projects to perform requirements tracing to ensure that each requirement is tied to a number of software deliverable documents and none of the requirements are left unanswered by the final software product. Requirements traceability matrix (RTM) is a tool that can help with requirements tracing as it documents the relationships between a requirement specification and other software documents such as design, source code, test, and defect documents. Since there are several deliverable documents with complex information involved, it can be difficult to find relevant documents and information. We propose a visualization tool that helps the software development team in tracking information related to the RTM. In particular, the tool supports tracing between user requirements and the documents related to testing of the software and the defects found when testing the software. Since there are several documents with complex information involved, it can be difficult to find relevant documents and information when tracing the requirements. Let us consider the relationships between the RTM and test-related documents in Figure 1.

KEY WORDS
Software Engineering, Requirements Traceability Matrix, Ontology, Visualization, Test Documents, Software Defects

1. Introduction
Software quality is one of the factors that affect the acceptance of software products and it is often assessed in terms of defects found in the software and whether the software can meet specified user requirements. It is hence a good practice for software companies to perform requirements tracing to ensure that each requirement is tied to a number of software deliverable documents and none of the requirements are left unanswered by the final software product. However, in a software development project, the number of user requirements can increase or change frequently, and hence there is a requirement for the project to keep track of the requirements and the deliverables within the project. Requirements traceability refers to “the ability to describe and follow the life of a requirement in both forward and backward direction” [1]. It becomes a valuable practice in software projects to ensure that each requirement is tied to a number of software deliverables and none of the requirements are left unanswered by the final software product. Requirements traceability matrix (RTM) [2] is a tool that can help with requirements tracing. It is a document, usually in the form of a table, which maintains records of the relationships between a requirement specification and other software documents such as design, source code, test, and defect documents. Backward and forward tracing can be done in order to verify that all requirements are met, and to identify the scope of impact on deliverables when requirement changes occur.

In this paper, we are interested in tracking the requirements and the documents related to testing of the software as well as the defects found when testing the software. Since there are several documents with complex information involved, it can be difficult to find relevant documents and information when tracing the requirements. Let us consider the relationships between the RTM and test-related documents in Figure 1.
Each requirement in the RTM links to a test case specification document, which is created in the unit testing phase, and to a unit integration & system testing document, which is created in the system integration phase. These test documents link to test defect log documents which contain the defects, as referenced by defect numbers, which are found during testing phases. The problems encountered when using these documents in requirement tracing are:

1) The RTM document itself is complex containing lots of information, including the references to other deliverable documents. It is difficult to find the required information.
2) Since there are multiple documents involved, it is difficult to understand the relationships between a requirement and other associated deliverables.
3) It is difficult to obtain some additional information such as what kinds of software defects are discovered during the project.

To tackle these problems, we propose a visualization tool that can help a software development team in tracing the requirements. We model the RTM and testing-related information as an ontology [3] that describes concepts of requirements within a requirement specification and its relationships to other information in test and defect log documents. In addition, we follow the IEEE standard classification of software anomalies [4] and model classes of defects in the ontology in order to allow reasoning about defect types that are associated with the tests conducted on the software. Using the ontology, the tool can support semantics-based queries and provide graphical representation of query results. In this way, the tool can aid the visualization and understanding of the connection and relationships between traced information. We use a case of core banking software, developed by a software company in Thailand, as an example.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes our RTM and test documents ontology, followed by the visualization tool and its architecture in Section 4. In Section 5, the banking software case study is presented. Section 6 gives the conclusion and future work.

2. Related Work

Our work is motivated greatly by the research of Guo et al. [5], Heim et al. [6], and Ha et al. [7]. Guo et al. [5] argue that traditional RTM mainly supports vertical traceability, i.e. tracing between functional requirements and software products that respond to the requirements, but it lacks tracing between functional requirements themselves. Such lateral traceability should make defect modifications and response to requirement changes more efficient. They model the RTM as an ontology to support vertical and lateral tracing queries. The ontology comprises the concepts of requirement, design, code, and test documents. Particularly, they define different kinds of relationships which specify how a functional requirement is related to another in order to support lateral tracing, i.e. extend, generalization, sequence, invoke, choice, and collateral relationships. Heim et al. [6] present graph-based visualization of the relationships between requirements. To deal with the problem of graph-based visualization not scaling well to large datasets, their approach allows users to view only a limited set of information that is related to the requirement of interest. That is, they allow the users to view requirements which relate to each other by user-defined relations, content relations based on similarity of requirement contents, and shared metadata relations. The work by Ha et al. [7] addresses a problem about technical manuals being diverse and containing large amount of information. It is therefore difficult for manual users to grasp information about the relationships between manual contents and laborious to search for specific information. Their approach builds an ontology to semantically represent and retrieve the manual contents. In their case the aircraft maintenance manuals are used, and then a visualization system is developed to help visualize the information so that the mechanics can grasp technical information more intuitively. Another visualization work by Muto et al. [8] helps software testers in unit testing. Their proposed tool can visualize passage rate and results of static checking which verifies whether Java source code satisfies invariants, pre-conditions, and post-conditions. The tool presents caller-callee relation between classes and determines the priority of classes on the basis of the weight of the caller-callee relation.

On semantic modeling, it can be seen that ontology has been widely adopted in various problem areas. Supnithi et al. [9] present a smart knowledge service to support all processes in the logistic chain of orchid industry. The service incorporates an orchid knowledge platform which is built upon an orchid ontology. Through this knowledge platform, users can query orchid-related information based on inference rules. Another interesting example related to our work is the research by Iliev et al. [10]. They predict severity levels for defects found when testing software. The prediction would be useful for prioritizing the defect-fixing activities. Using ontology, they model defects and defect attributes according to IEEE standard defect classification also [4]. The defect attributes include defect types, impact of the defects on software quality attributes, and the phases in which the defects are inserted and detected. Then, rules are defined for severity levels based on these data. For example, a defect which is a data/interface/logic problem, inserted early in the requirement or design phases, detected in the supplier testing or coding phases, and affects at least two software quality attributes will be assigned with major severity level and should get immediate attention.

3. RTM and Test Documents Ontology

As mentioned earlier, the visualization system is based on the RTM and test documents ontology. This ontology
defines vocabulary or terms within our domain of RTM and test documents. Each term is either an individual, class, property, or relation [3]. An individual is a specific object in the domain. Any group of objects with common properties can be grouped into a class. A property is a nature or characteristic of the class. The relation refers to the relation between the objects in the ontology.

We build the RTM and test documents ontology in OWL language using Hozo ontology editor [11]. The result is shown in Figure 2. The ontology building process is as follows.

1) Gather related RTM and test documents of the software project of interest, i.e. RTM, test case specification documents, unit integration & system testing documents, and test defect log documents. They can be spreadsheet files.
2) Define classes based on the schema of the information in the documents. Figure 2 shows main classes derived from the RTM and test documents, such as RTM, TestDocument, TestStep, Phase, DefectType, and DefectLogDocument. See more detail in Section 4.1.
3) Define classes of defect types based on IEEE standard classification of software anomalies [4]. Figure 2 shows only some part of the whole classification. There are eight different classes of SoftwareAnomalies, i.e. LogicProblem, ComputationProblem, Interface/Timing Problem, DataHandlingProblem, DataProblem, DocumentQualityProblem, and Enhancement. Each class is further divided into subcategories such as LogicProblem is subcategorized into ForgottenCasesOrSteps, DuplicateLogic, UnnecessaryFunction etc.
4) Define relations between the classes based on primitive relations, i.e. is-a, part-of, and attribute-of relations. Is-a relation is a class-subclass relation. The defect types and subcategories are defined using the is-a relation. The part-of relation (p/o) represents the whole-part relation and the attribute-of relation (a/o) represents the properties of each class [12]. For example, in Figure 2, TestStep is a part of TestDocument and TestAction is an attribute of TestStep.
5) Repeat step 2-4, if necessary, to improve and finalize the ontology.

The RTM and test documents ontology is used as a foundation for building the visualization system.

4. Visualization System for RTM and Test Documents

The architecture of the visualization system built on top of the RTM and test document ontology is depicted in Figure 3. It consists of three layers which are data, application, and presentation layers. The detail of each layer is described below.

Figure 2. Classes and their properties and relationships within RTM and test documents ontology
4.1 Data Layer

In this section, we will first give more detail about the main classes within the RTM and test documents ontology in Figure 2 and then describe how to process this ontology.

The main classes of the ontology are shown in Figure 4. RTM refers to the RTM document. TestDocument refers to a test case specification document or a unit integration & system testing document. DefectLogDocument refers to a test defect log document. DefectType refers to any class of software defects. Phase refers to a phase within the software development life cycle. Among these classes, TestDocument has a direct relation to RTM; it is part-of RTM whereas TestStep is part-of TestDocument.

The class TestDocument contains the test document name and links to the class RTM. Its properties are listed in Table 2.

The class TestStep which is a part of TestDocument has its test action and result. This class links to TestDocument by test document ID. Its properties are listed in Table 3.

The class DefectType corresponds to defect classification. Its properties are listed in Table 4.
The class Phase corresponds to the software development life cycle phases such as design, build, test, and deployment. Its properties are listed in Table 5.

Table 5  
Property of class Phase
\[
\begin{array}{|c|c|} 
\hline
Property & Description \\
\hline
hasPhaseID & A phase has its unique ID. \\
hasPhase & A phase has its name. \\
\hline
\end{array}
\]

The class DefectLogDocument contains log information of the defects, i.e. defect no., defect description, defect type, originator, injected phase, and detected phase. In addition, it links to the class TestStep since the defect can be discovered at any test step. Table 6 lists the properties of this class.

Table 6  
Property of class DefectLogDocument
\[
\begin{array}{|c|c|} 
\hline
Property & Description \\
\hline
hasDefectNo & A defect has its unique ID. \\
hasDefectDescription & A defect has its defect description. \\
hasDefectType & A defect has its type. \\
hasOriginator & A defect has owner such as programmers, business analyst. \\
hasPhaseInjected & A defect has its injected phase. \\
hasPhaseDetected & A defect has its detected phase. \\
hasTestStep & A defect is discovered at a test step. \\
\hline
\end{array}
\]

To build the data layer of Figure 3, first we create a relational database; the tables correspond to the RTM and test documents and the columns of each table correspond to the schema of the information within each document. We then populate the tables with all the requirements and test-related data from the RTM and test documents. After that, we use the Ontology Application Management Framework (OAM) [12] which provides an ontology processing API [13] to map between the OWL data model in Figure 2 and the data entries in the database before generating all information in RDF format. This information, comprising all ontology concepts and individuals, is kept in an RDF repository and will be used for query in the application layer.

4.2 Application Layer

The application layer has a query process module whose task is to find information related to the input key word of the user. We use the methods of the class OntologyProcessing of the OAM framework as listed in Table 7 to process the queries. The key word is used to look up in the RDF repository and the query process module enables us to find appropriate information which has relations with the key word as defined in the ontology. In the case that the user inputs a defect ID, the query result will be the information related to the defect, such as the application that causes the defect, the requirement function impacted by this defect, and other defects of the same type. The query result will be used by the render module in the presentation layer.

Table 7  
Example of methods of class OntologyProcessing
\[
\begin{array}{|c|c|} 
\hline
Method & Method Description \\
\hline
getAllProperties(String className); & Shows all properties of a class \\
getListOfDirectSuperClasses(String className); & Shows all superclasses of a class \\
getListOfDirectSubClasses(String className) & Shows all subclasses of a class \\
getAllDataTypeProperties(String className); & Shows all properties of a class which characterize the attribute-of relation \\
getAllObjectTypeProperties(String className); & Shows all properties of a class which characterize the part-of relation \\
\hline
\end{array}
\]

4.3 Presentation Layer

The presentation layer has a render module which can present the query result from the application layer as a graph. To achieve that, we use the prefuse toolkit [14], which uses Java2D graphics library, for information visualization. At the center of the graph is the input key word, with links to other related information. The system supports a navigation function, i.e. when the user clicks on a node, it is expanded and more related information nodes are displayed. To help the user understand the graph easily, we use different colors to differentiate among different classes of information nodes.

As an example, Figure 5 depicts the query result for the input key word application ID “007”. Related classes are directly connected to this input node. The blue nodes connected to the input node at the center here represent the requirements of the class RTM which are tied to this application ID 007. The user can click on any of the blue nodes to expand its property information nodes. In this case, when the user clicks on the node that represents the requirement ID “BTC_R19.2_002”, its pink property nodes are displayed. That is, this requirement is associated with the application named “BatchTerminalCenter”, its impact description is “Perform initial validation process”, its impact types are “FunctionChange, InterfaceChange, DataChange”, and the requirement priority is “High”.
Since there can be a large amount of information in the graph, long textual information describing certain nodes (i.e., impact description, defect description, test action, and expected result) will be shortened to improve readability of the graph.

5. Case Study

This section presents the case of a software company in Thailand which develops software in core banking business. The software is a legacy system using COBOL, JCL, and CICS programming. With the complexity of the core banking system, the number of requirements increases at all time, and there are many software products involved. It is then necessary to trace requirements and our visualization system is used to support the development team in this activity.

We use some information in the RTM and test documents with regard to the Batch terminal center system in an experiment. Batch terminal center is an application that supports customers who register a direct credit, direct debit, and payroll product. Figure 6 shows the graphical user interface of the visualization system. The user can put any key word into the input textbox in the search area (A) and select the kind of the key word from the list, i.e., the key word is either an application ID, requirement ID, defect description, defect log ID, or defect type. When the user clicks the search button, the system will return the result as a graph in the display area (B).

Figure 7 shows the result of a search for a defect in the application BTC1B100 which generates the report name incorrectly. The defect description is “Report name is incorrectly displayed as SB1P523 instead of BTC1P100”. The query result shows the information related to this defect such as defect ID (i.e. BTC-001), phase detected (i.e. Build), defect type (i.e. CheckingWrongVariable), the test step that discovers this defect (i.e. Run Job BTCJ201 to execute program BTC1B100), test document which contains that test step (i.e. R19.2-AMS-TSP_APP_007-App#007 BTC_BTC1B100_UT.xls), and originator (i.e. PadungpongK). The user can track the test step in the test document which finds the defect. In addition, the system can show that the defect type “CheckingWrongVariable” is of a subcategory of the defect type “LogicProblem”. Also, if the user clicks on the requirement ID “BTC_19.2_001", the visualization system will further expand the information related to this requirement. In this way, the user can easily track the information related to the queried defect without having to browse through several documents.

6. Conclusion

This paper has described the development of a visualization tool for software requirements traceability matrix and test documents. An ontology is created to model information in those documents and the classification of software defects. The ontology enables
the visualization system to execute semantics-based queries and visually display the query results. An experiment with a case of core banking software shows that the users in a development team can conveniently trace between requirements, test, and defect documents and can visualize their relations quite easily.

We are in the process of building the next version of the visualization tool. The query process module in the application layer will incorporate rule management, i.e. certain rules can be defined to enhance query on the ontology. For example, rules can be defined on how to fix particular defects using defect resolution history. In addition, we plan to create an auto-update for the RDF repository when there are updates of requirements and test information.

References