An Approach of Test Case Generation with Software Requirement Ontology

Adisak Intana, Kuljaree Tantayakul, Kanjana Laosen, Suraiya Charoenreh
College of Computing, Prince of Songkla University, Phuket, Thailand

***Abstract*—Software testing plays an essential role in software development process since it helps to ensure that the developed software product is free from errors and meets the defined specifications before the delivery. As the software specification is mostly written in the form of natural language, this may lead to the ambiguity and misunderstanding by software developers and results in the incorrect test cases to be generated from this unclear specification. Therefore, to solve this problem, this paper presents a novel hybrid approach, Software *Req*uirement *Onto*logies based *Test* Case *Gen*eration (*ReqOntoTestGen*) to enhance the reliability of existing software testing techniques. This approach enables a framework that combines ontology engineering with the software test case generation approaches. Controlled Natural Language (CNL) provided by the ROO (Rabbit to OWL Ontologies Authoring) tools is used by the framework to build the software requirement ontology from unstructured functional requirements. This eliminates the inconsistency and ambiguity of requirements before test case generation. The OWL ontology resulted from ontology engineering is then transformed into the XML file of data dictionary. Combination of Equivalence and Classification Tree Method (CCTM) is used to generate test cases from this XML file with the decision tree. This allows us to reduce redundancy of test cases and increase testing coverage. The proposed approach is demonstrated with the developed prototype tool. The contribution of the tool is confirmed by the validation and evaluation result with two real case studies, Library Management System (LMS) and Kidney Failure Diagnosis (KFD) Subsystem, as we expected.**

***Keywords*—Software testing; software requirement specification; ontology; test case; equivalence and classification tree method**

I. Introduction

Software testing is one of the most important stages to detect errors in software development. The number of software bugs are not mainly caused by the code or design. One of the main causes of software bugs is from the specification [1][2]. As software specification gathered from the user's needs is mostly written in common natural languages in the Software Requirements Specification (SRS) document [3][4][5], this leads unstructured requirements to be ambiguous and misunderstood by software developers [4][6][7]. Furthermore, in system and user acceptance testing, test cases are generated from the SRS. This may result in incorrect test cases to be generated from the unclear specification. Therefore, it is necessary that the requirement specification needs to be very clear and well-defined before generating test cases.

Ontology engineering has been applied in Requirements Engineering (RE). An ontology is a formal representation of entities and relationships in a domain of interest [8]. As the semantics of concepts are formally defined, an ontology can be used as a formal specification for a program. A domain vocabulary, essential concepts with their taxonomy, relationships (and constraints) between concepts, and domain axioms are defined for specific program applications [8][9]. Thus, using ontologies to express requirement specifications has implications for advantage in managing complexity, contradictions, or detecting ambiguity and incompleteness of requirements [4][10][11]. The application of ontology to requirement specification can help to eliminate the problem of erroneous test case generation from ambiguous, inconsistent, or incomplete requirements. Thus, our challenge is to add value to software testing with ontology modelling in requirement specification [12].

Therefore, in our previous work [13], we presented how ontology engineering approach can enhance practical software testing. We proposed a conceptual vision of framework called *ReqOntoTestGen* (Requirement Ontology Testcase Generation) that combines the benefit of ontology to represent the semantics of requirement specification with Control Natural Language (CNL) and Classification Tree Method (CCTM) [14][15] testing technique to generate test cases. The ROO (Rabbit to OWL Ontology Authoring) tool [16] is used by this framework to design and develop an ontology with CNL or Rabbit Language. This results in the complexity of requirements in natural languages to be reduced and the semantic of requirements formally defined. The specific syntax of this tool increases the structure and eliminates the ambiguity of the requirement ontology. The result of this tool is an export in Web Ontology Languages (OWL) format to transform into a structured data dictionary, before it is considered with decision tree to generate all possible test cases. Furthermore, CCTM provided by *ReqOntoTestGen* framework also allows the number of generated test cases to be minimized by reducing the redundant test cases and the testing coverage that covers all possible testing scenarios to be maximized. We demonstrated manually the effectiveness of the framework with a real case study, Library Management System (LMS).

The work of this paper is extended from the previous work [13]. This paper proposed a semi-automatic approach for test case generation from the requirement specification ontology based on use case-based requirement specification. To demonstrate the practical implementation of the approach, we developed a prototype tool according to *ReqOntoTestGen* framework in which the ontology engineering and test case generation algorithm is implemented in the tool. Control Natural Language (CNL) enabled by the ROO tool is used to be a guideline and build conceptual ontologies from the requirement specification. To generate test cases, the result from the ROO tool, the ontology represented in terms of OWL format, is transformed into the XML file of data dictionary. The OWL and XML transformation rules were designed and

implemented into our prototype tool for this data dictionary transformation. CCTM techniques were implemented in the tool for automatic test case generation purposes. The XML file of decision tree specifying the constraint of test case generation is considered with the XML file of data dictionary to generate test cases. Moreover, the validity, effectiveness and accuracy of the approach and tool were guaranteed by two different case studies formulated from real-world systems, Library Management System (LMS) and Kidney Failure Diagnosis (KFD) Subsystem. We compared the actual result of test case generation from these case studies by the tool with the expected test cases calculated manually by the practical testers. Furthermore, the satisfaction level of the proposed approach and tool was evaluated by practical specialists for future use.

The remainder of the paper is organised as follows. Firstly, Section II explains an overview of the necessary background and related work, before the proposed approach and the real-world case studies for experimenting the effectiveness of the approach are described in Section III. In Section IV, the proof of concept of our proposed approach are demonstrated through the evaluation of the implemented prototype with the case studies. Section V discusses the lesson learned experienced from the study result. Finally, the conclusion and future work are described in Section VI.

II. Related Work

Several research studies have been interested in using ontology in the software development process to increase the efficiency of developed products. For instance, [17] proposed the software process automation ontology (Sponto) that applied the ontology-based approach to generate a set of artefacts for the software development process such as user stories for requirement specification and SQL for database scripts. Another example is the work of [18] which introduced the mechanism for transforming security requirements described in the form of the natural language into a structured ontology. The inconsistencies of security requirements were also checked by this mechanism.

However, most studies focus on using ontology to represent the conceptualisation and knowledge information regarding software development domains. [19], for example, proposed the application of ontology to define the information and knowledge semantics in RE. Instead of using ontology to represent the semantic of the requirement itself, this work focused on the use of ontology to describe the way of structuring requirements in the SRS document. Similar to this, [20], [21] and [22] proposed a domain ontology for software requirement change management, requirement classification and use story assessment in requirement artefacts respectively. In [23], they proposed ROoST (Reference Ontology on Software Testing) that builds a set of interrelated ontology patterns related to the software testing concepts including its process, activities, artefacts and testing techniques for test case design in order to associate semantics to a large amount of test information. Similar to this [24], [25] and [26] applied the ontology-based method to represent the knowledge related to software testing activities. The common well-established vocabulary for testing is used in the ontology application. Their developed ontologies influence the benefit of knowledge sharing among the development team.

Furthermore, some studies focus more on the application of requirement ontologies to generate test cases in practice. [23], for instance, presented a combined inference to software requirement ontology to generate test cases based on software requirement specification. The test cases were obtained from test input, test procedure, and expected test results. The work proposed by [23][27] used inference rules based on reasoner to generate test cases and improve requirements coverage and domain coverage. Furthermore, [28] presented Web Ontology Language for Web Service (OWL-S) to describe the workflow in the web service application. Petri-Net is used to represent the meaning of the test process and OWL-S is used to generate test data. In addition, [29] presented application of OWL ontologies to generate test cases and test procedures based on controlled-English model. The closely related work is proposed by [30]. They proposed test case generation using a learning-based software testing approach based on requirement ontology to generate test cases. However, those research studies mentioned earlier only focus on the application of requirement specification ontology to generate test cases, they did not consider testing coverage in test case generation. Based on our literature reviews, it can conclude that most of the existing research studies focus on using ontology to represent the software testing concept and knowledge sharing in software engineering communities. A few studies considered more important in the use of requirement specification ontology in the software testing process to generate comprehensive test cases together with testing coverage analysis of test case generation.

III. Materials and Approach

A. ReqOntoTestGen Framework

Fig. 1 shows a framework of the test case generation with software requirement ontology (*ReqOntoTestGen*) proposed in our previous work [13]. There are four steps in this framework. (1) *Ontology Engineering* generates the ontology according to CNL from the functional requirement definition described in terms of natural language by using ROO-CNL authoring. CNL in ROO authoring enables the complex requirement to be transformed into a very simple requirement before generating the ontology. Then, the achieved requirement ontology is exported in terms of OWL format, before (2) *XML Generation* transforms the exported OWL into the XML data dictionary metadata. In (3) *Variable and Decision Tree Management*, it starts with the variable information extracted from the XML of use case defined in the SRS document, before the corresponding data structure of the extracted variable is extracted from the XML data dictionary. This, then, is considered with the XML file of decision tree for test cases generation. Finally, (4) *Test Case Generation* creates test cases from the variable and its conditions by using CCTM technique. CCTM test case generation technique was chosen to be implemented in the proposed framework as it provides the benefit in which the number of test cases are minimized by eliminating the redundant test cases and the testing coverage is maximized in which all possible range value of test input variables is expanded.

B. ReqOntoTestGen Algorithm

To achieve a better understanding of our *ReqOntoTestGen* Framework explained in Section III-A, this section describes

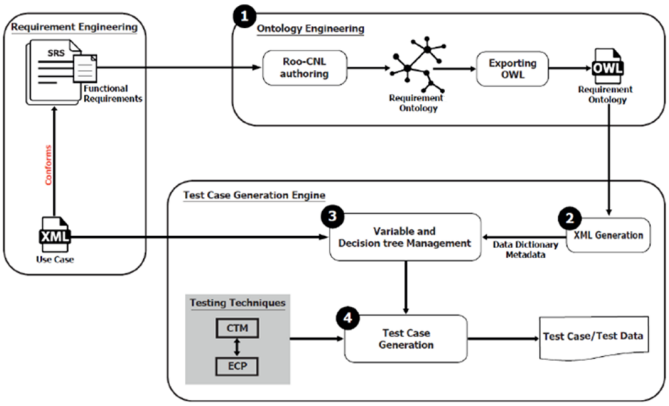


Fig. 1. The *ReqOntoTestGen* framework [13].

the algorithm of the framework in detail.

1) Step 1: Ontology engineering: In this step, the complexity of natural language based functional requirement and its corresponding constraints are reduced by transforming them into CNL structure. Then, the target ontology is developed from the transformed CNL based requirement. Table I shows an example of mapping from natural language-based requirement to ROO-CNL Structure and OWL2 Functional Syntax respectively. When classifying and structuring the ROO-CNL successfully, the ontology is exported in terms of file OWL format for data dictionary metadata generation in the next step.

TABLE I. EXAMPLE OF ROO STRUCTURE

Description	Roo-CNL Structure	OWL 2 Functional Syntax
Class Declaration	*cname* is a concept	Declaration(Class(*:cname*))
Subclass	Every *scname* is a kind of *cname*	SubClassOf(*:scname :cname*)
Relationship Declaration	*rname* is a relationship Every *cname1 rname cname2*	Declaration(ObjectProperty(*:rname*)) ObjectPropertyDomain(*:rname :cname1*) ObjectPropertyRange(:rname :cname2)
Instance Declaration	*insname* is a *cname*	Declaration(NameIndividual(*:insname*)) ClassAssertion(*:cname :insname*)

2) Step 2: The XML generation: In this step, the OWL files obtained from the ontology are transformed into XML structures of data dictionary that is used for test case generation. The XML format is used for the target file transformed from the source of OWL file to make it easier to exchange data between programs [31]. Based on the study of [32][33], 13 relevant transformation rules are designed and used for transformation. All rules are available on our tool website[1].

Fig. 2 shows an example of transformation rules, consisting of the first column (Rules) as the rules of transformation. The second column (OWL2 Functional) is an OWL syntax. The last column (XML Schema) is an XML syntax. The transformation consists of three main categories, the structure of classes and relations, object property restrictions, and data property restrictions.

3) Step 3: The variable and decision tree management: This step considers two input files, the XML file of use cases and the XML file of data dictionary transformed from the OWL of requirement ontology. The use case files are designed from requirements in the SRS document according to UML Development Guidelines Version 2.0 [34]. The use case normally demonstrates the overview of functionality and procedure of the system to generate test cases. Fig. 3 shows

[1]https://sites.google.com/phuket.psu.ac.th/reqontotestgen/

Rules	OWL2 Functional	XML Schema
The Structure of Classes and Relations		
1. Class Declaration	Class(*:cname*)	Local: <xs:element name="*cname*"> <xs:complexType> </xs:complexType> </xs:element> Global: <xs:element name="*cname*"/>
2. Subclass Declaration	SubClassOf(*:scname :cname*)	<xs:element name="*scname*" type="*cname*"/>
3. Object Property Declaration	ObjectProperty(*:hasClass*) ObjectPropertyDomain(*:hasClass :cdomain*) ObjectPropertyRange:(*:hasClass :crange*)	<xs:element name="*cdomain*"> <xs:complexType> <xs:sequence> <xs:element name="*hasClass*" ref="*crange*"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="*crange*"> <xs:complexType> </xs:complexType> </xs:element>
...	...	...
Object Property Restrictions		
5. Existential	EquivalentClasses(*:cdomain* ObjectSomeValuesFrom(*:hasClass :crange*))	<xs:element name="*cdomain*"> <xs:complexType> <xs:sequence> <xs:element name="*hasClass*" ref="*crange*"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="*crange*"> <xs:complexType> <xs:choice> </xs:choice> </xs:complexType> </xs:element>
6. Universal	EquivalentClasses(*:cdomain* ObjectAllValuesFrom(*:hasClass :crange*))	<xs:element name="*cdomain*"> <xs:complexType> <xs:sequence> <xs:element name="*hasClass*" ref="*crange*"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="*crange*"> <xs:complexType> <xs:all> </xs:all> </xs:complexType> </xs:element>
...	...	...
Data Property Restrictions		
...	...	...
11. Individual value	NameIndividual(*:individual1*) NameIndividual(*:individual2*) NameIndividual(*:individual3*) ClassAssertion(*:cname1 :individual1*) DataPropertyAssertion(*:dprop :individual1 "value1"^^xsd:dtype*) ClassAssertion(*:cname1 :individual2*) DataPropertyAssertion(*:dprop :individual2 "value2"^^xsd:dtype*) ClassAssertion(*:cname1 :individual3*) DataPropertyAssertion(*:dprop :individual3 "value3"^^xsd:dtype*	<xs:element name="*dprop*"> <xs:complexType> <xs:choice> <xs:element name="*value1*" type="*xs:dtype*"/> <xs:element name="*value2*" type="*xs:dtype*"/> <xs:element name="*value3*" type="*xs:dtype*"/> </xs:choice> </xs:complexType> </xs:element>
12. Minimum cardinality	EquivalentClasses(*:cname* DataMinCardinality(*min :dprop xsd:dtype*))	<xs:element name="*cname*"> <xs:complexType> <xs:sequence> <xs:element name="*dprop*" type="*xs:dtype*" minOccurs="*min*"/> </xs:sequence> </xs:complexType> </xs:element>
13. Maximum cardinality	EquivalentClasses(*:cname* DataMaxCardinality(*max :dprop xsd:dtype*)	<xs:element name="*cname*"> <xs:complexType> <xs:sequence> <xs:element name="*dprop*" type="*xs:dtype*" maxOccurs="*max*"/> </xs:sequence> </xs:complexType> </xs:element>

Fig. 2. Example of OWL and XML transformation rules.

an example of brief description of use case UC001 describing the behaviour of function *Borrow Item* of LMS. To represent the function behaviour, a sequence of the step-by-step is used including main flow of events for the most common success scenario, alternative flow of events for other less common success scenario and exception flow of events for the error management scenario. The input and output for function operation, then, are indicated from the use case corresponding steps. The data structure of input and output variables are described in the XML files of data dictionary transformed from the OWL of requirement ontology.

Use Case	Description
UC001-Borrow Item	Use Case ID: UC001 Use Case Name: Borrow Item Pre-Condition: Checking members of the library Post-Condition: Borrow items successfully Priority: High Flow of Event: 1.The system shows GUI for a list of items to borrow 2.Member select items to borrow 3.Member confirm borrow items [A1] 4.The system records the borrowed items 5.The system displays a list of borrowed items Alternative Flow: [A1] If click "Cancel" button, the system will not records [E1] Exception Flow: [E1] The system shows the warning "Confirm cancellation?"

Fig. 3. Example of use case detail.

4) Step 4: Test case generation: Our framework implements CCTM technique for test case generation. It generates test cases from the extracted input variable with the corresponding data structure. The test case generation process is described as follows.

Step 4.1: Classification Tree Generation with CTM Technique. CTM technique generates a classification tree from the information extracted from the use case. It starts with the name of the system represented by the use case name to be a root node of the tree. Then, it layers the tree from the root node to the terminal classification node with the subsystem and its corresponding variables respectively. The leaf node of the tree, terminal class, defines the range of variable values which are considered to create partitions for both valid and invalid data values by ECP technique. This data range value is used to generate test cases in the later step. An example of classification tree for function *Borrow Item* of LMS resulted from CTM is shown in Fig. 4. The variables and their corresponding range value are visualised in terminal classification (parent node) and terminal class (leaf node) of the tree respectively. This can be explained as follows: *Member* = {*AdminStaff*, *Grad*, *Lecturer*, *Undergrad*, *None*}, *Item* = {*Book*, *CD*, *DVD*, *None*}, *borrowDate* = {*beginDate-endDate*, *None*} and *maxDaysBorrow* = {*7*, *14*, *30*, *None*}. These are considered to create an equivalence class partitioning in the next step.

Step 4.2: Test Case Generation with ECP Technique. In the

Fig. 4. Example of a classification tree for function *Borrow Item* of LMS.

classification tree achieved from CTM technique, ECP divides the terminal classification into equivalence classes for each possible range of data values. The framework implemented a strong robust format [4] to generate a test case. The equivalence class in this form considers both valid and invalid values of all classes of equivalence and allows the test case generation to cover every possible value of all equivalence classes. An example ECP for function *Borrow Item* of LMS is shown in Fig. 5.

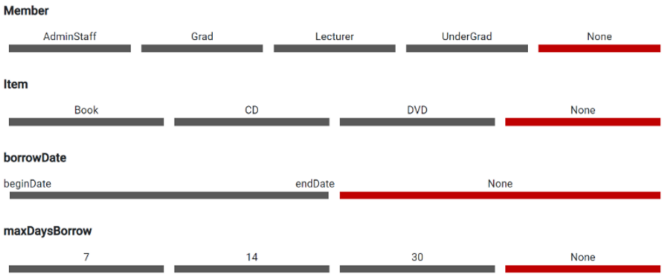


Fig. 5. Example of equivalence class partitioning for function *Borrow Item* of LMS.

C. Case Studies

To demonstrate the effectiveness of our proposed approach, case studies from the real world system are used. We consider two different case studies for this purpose. One is a Library Management System (LMS) deployed in Prince of Songkhla University, Phuket. The other is Kidney Failure Diagnosis (KFD) subsystem from Hospital Information System (HIS) replicated from [15]. The following sections describe the case study information together with the demonstration of how our approach manually works.

1) Library Management System (LMS): The LMS[2] is a system for managing various library resources. The members of the library can borrow or return resources such as books, CDs, or DVDs. Each type of member has different borrowing

[2]http://library.phuket.psu.ac.th/

conditions. To generate a test case, we considered the return function that contains fine calculation when the late return occurs. The detailed information as well as practical requirements were formulated from the LMS of Prince of Songkla University.

Requirements: The functional requirements of LMS is shown in Table II.

TABLE II. THE FUNCTIONAL REQUIREMENTS OF LMS

Req. ID	Reqiurements
LIB-FUN-01	Members can borrow item including books, CDs or DVDs.
LIB-FUN-02	Members are classified into admin staff, graduate student, lecturer and undergraduate.
LIB-CON-01	Books, CDs, DVDs must be disjointed.
LIB-CON-02	Admin, staff, graduate student, lecturer and undergrade must be disjointed.
LIB-CON-03	Maximum borrowing items and borrowing periods: 5 books per 7 days for admin staff and undergraduate students, 10 books per 14 days for graduate students, and 15 books per 30 days for lecturers.
LIB-CON-04	Maximum borrowing items and borrowing periods: 3 discs per 7 days for all members.

Ontology Engineering: From the functional requirements of LMS as shown in Table II, it can be used to design and develop an ontology which consists of classes, relationships, and data properties. The ontology syntax of LMS is shown in Fig. 6.

Req. ID	Roo-CNL Structure	OWL2 Functional Syntax
LIB-FUN-01	*Member* is a concept *Item* is a concept	Declaration(Class*(:Member)*) Declaration(Class*(:Item)*)
	Every *Book* is a kind of *Item* Every *CD* is a kind of *Item* Every *DVD* is a kind of *Item*	SubClassOf*(:Book :Item)* SubClassOf*(:CD :Item)* SubClassOf*(:DVD :Item)*
	hasBorrow is a relationship Every *Member hasBorrow Item*	Declaration(ObjectProperty*(:hasBorrow)*) ObjectPropertyDomain*(:hasBorrow :Member)* ObjectPropertyRange*(:hasBorrow :Item)*
LIB-FUN-02	Every *AdminStaff* is a kind of *Member* Every *Grad* is a kind of *Member* Every *Lecturer* is a kind of *Member* Every *UnderGrad* is a kind of *Member*	SubClassOf*(:AdminStaff :Member)* SubClassOf*(:Grad :Member)* SubClassOf*(:Lecturer :Member)* SubClassOf*(:UnderGrad :Member)*
LIB-CON-01	*DisjointClasses()*	DisjointClasses*(:Book :CD)* DisjointClasses*(:Book :DVD)* DisjointClasses*(:CD :DVD)*
LIB-CON-02	*DisjointClasses()*	DisjointClasses*(:AdminStaff :Grad)* DisjointClasses*(:AdminStaff :Lecturer)* DisjointClasses*(:AdminStaff :UnderGrad)* DisjointClasses*(:Grad :Lecturer)* DisjointClasses*(:Grad :UnderGrad)* DisjointClasses*(:Lecturer :UnderGrad)*
LIB-CON-03	*borrowDaysR1* is a *maxDaysBorrow*	Declaration(NameIndividual*(:borrowDaysR1)*) ClassAssertion*(:AdminStaff :borrowDaysR1)*
LIB-CON-04	And, configure data property assertion directly through GUI in the tool.	DataPropertyAssertion*(:maxDaysBorrow :borrowDaysR1 "7"^^xsd:integer)*
LIB-FUN-03	*hasReturn* is a relationship Every *Member hasReturn Item*	Declaration(ObjectProperty*(:hasReturn)*) ObjectPropertyDomain*(:hasReturn :Member)* ObjectPropertyRange*(:hasReturn :Item)*
LIB-FUN-04	*DataProperty()* *DataPropertyDomain()* *ObjectSomeValuesFrom()* *DataPropertyRange()*	Declaration(DataProperty*(:fine)*) DataPropertyDomain*(:fine* ObjectAllValuesFrom*(:hasReturn :Member))* DataPropertyRange*(:fine xsd:integer)*

Fig. 6. The ontology syntax of LMS.

Fig. 7 shows the ontology structure of LMS generated by ROO tool. It consists of two classes that are related to each other. The *Member* class is a member of the library including *AdminStaff*, *Grad*, *Lecturer*, and *UnderGrad*. The *Item* class is a library resource that can be borrowed including *Book*, *CD*, and *DVD*. The *ObjectProperty* between the *Member* and *Item* classes represents the relationship in which members can borrow (*hasBorrow*) library resources. Another relationship, *hasReturn* is a relationship where members can return library resources after they have been borrowed. Furthermore, the *DataProperty* is also an entity of data, the domain is a class, and the data type is a range of data properties. For example, *borrowDate* has class *Member* to be a domain and *xsd:dateTime* to be a range. Moreover, an individual or instance of value such as *borrowDayR1 "7"* is the condition for the maximum of days to borrow the *Book* of *UnderGrad* member type.

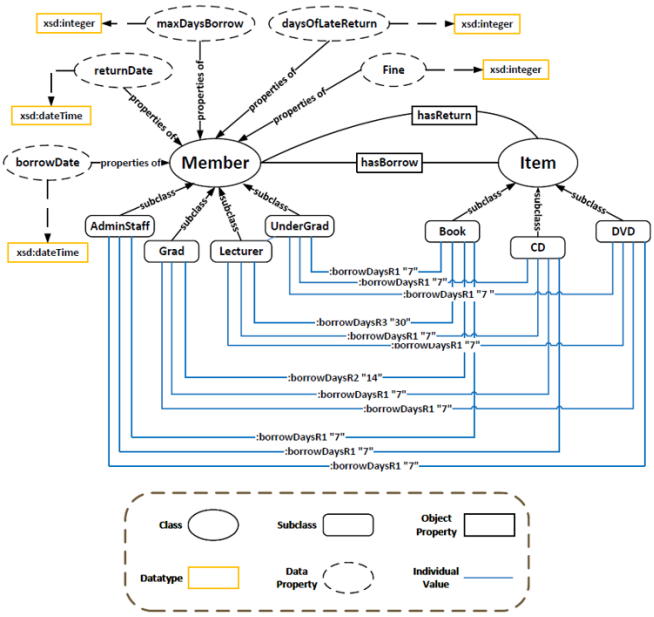


Fig. 7. The ontology structure of LMS.

Test Case Generation: Fig. 8 demonstrates the classification tree resulted from CTM technique. In the tree, *Library Management System* as a system name is considered to be a root node, before the subsystem *Return Item* is defined as a terminal classification in the next level. Variables *Member*, *Item*, *borrowDate*, *returnDate*, *maxDaysBorrow*, *daysOfLateReturn*, and *fine* related to this function are defined in the below level in the tree. These variables are considered to generate test cases by using ECP in the later step. The terminal class of each terminal classification defining the possible range of value is used to be a partition for generating test cases and test data in ECP.

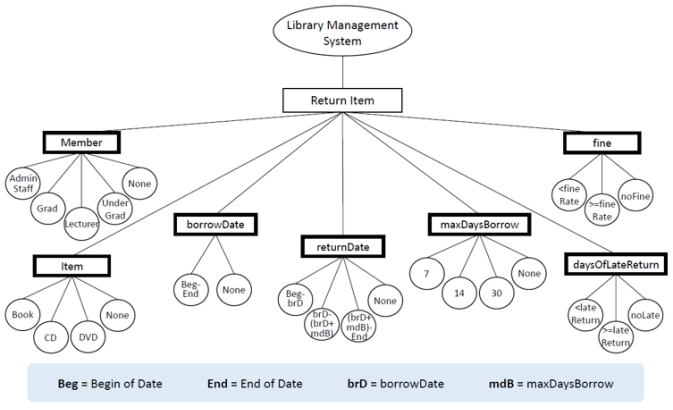


Fig. 8. The classification tree of LMS.

Test cases are generated from the Cartesian product of all

equivalence classes defined in four input variables (*Member*, *Item*, *borrowDate*, *returnDate*). There are a total of 160 (5*4*2*4) test cases to be generated. An example of test cases and test data is shown in Table III.

TABLE III. EXAMPLE OF TEST CASES AND TEST DATA FOR TESTING RETURN OPERATION

TC#	Member	Item	borrow Date	return Date	maxDays Borrow	daysOf LateReturn	fine	Comments
1	AdminStaff	Book	1/7/2019	2/7/2019	7	0	0	Valid
2	AdminStaff	Book	5/7/2019	15/7/2019	7	3	9	Valid
...	...	...	...	...	...	...	...	...
67	Lecturer	Book	10/7/2019	9/6/2019	30	-1	-3	Invalid
68	Lecturer	Book	10/7/2019	None	30	None	None	Invalid
...	...	...	...	...	...	...	...	...
160	None	None	None	None	None	None	None	Invalid

Table III is the test case generation result for testing return operation. Consider test case #1, it is a normal test case in which there is no late return for *AdminStaff*. This test case is different from test case #2. This results in the fine of 9 (3*3) to be calculated. Furthermore, the generated test cases cover in the case of invalid. In invalid test case #67, for example, it defines the return date before the borrowing date.

2) Kidney Failure Diagnosis (KFD) Subsystem: The KFD subsystem is a system for recommending the treatment appropriately to physicians for patients that have kidney dysfunction. It is calculated from the *Glomerular Filtration Rate* (*GFR*) result, consisting of *sex*, *age*, and *creatinine result* (*SCr*). The *GFR* and *Urine Creatinine (UO)* results are paired to interpret the stage of kidney failure. This case study is based on [15]. It is an open-source system and is part of the Hospital Information System called HospitalOS[3]. It is a system that is installed and used in community hospitals and more than 100 clinics in Thailand.

Requirements: The functional requirements of KFD that design and develop an ontology comprise a total of four requirements as shown in Table IV.

TABLE IV. THE FUNCTIONAL REQUIREMENTS OF KFD

Req. ID	Reqiurements
KFD-FUN-01	Stage is paired with GFR and UO.
KFD-FUN-02	Stage of GFR includes ESRD, Loss, Failure, Injury and Risk.
KFD-CON-01	ESRD, Loss, Failure, Injury, Risk must be disjointed.
KFD-CON-02	GFR is calculated with sex, age, height and SCr.

Ontology Engineering: From the functional requirements of KFD in Table IV, it can be used to design and develop an ontology which consists of classes, relationships, and data properties. The ontology syntax of KFD is shown in Fig. 9.

Fig. 10 shows the ontology structure of KFD resulted from ROO tool. It consists of three classes: *Stage*, *GFR*, and *UO*. The stage of kidney failure includes *ESRD*, *Loss*, *Failure*, *Injury*, and *Risk*. The *ObjectProperty* is the relationship between classes. For example, *hasPair* is a relationship between *GFR* and *UO* class to represent a pair to interpret the stage of kidney failure. Furthermore, the *DataProperty* is also an entity of data. As *GFR* contains *Scr*, *Height*, *Age* and *Sex*, they are defined as a data property. In the data property, the domain is a class and the data type is a range. For example, *Height* has class *GFR*

[3]http://www.opensource-technology.com

Req. ID	Roo-CNL Structure	OWL2 Functional Syntax
KID-FUN-01	*Stage* is a concept *GFR* is a concept UO is a concept	Declaration(Class*(:Stage)*) Declaration(Class*(:GFR)*) Declaration(Class*(:UO)*)
	hasGFR is a relationship Every *Stage hasGFR GFR*	Declaration(ObjectProperty*(:hasGFR)*) ObjectPropertyDomain*(:hasGFR :Stage)* ObjectPropertyRange*(:hasGFR :GFR)*
KID-FUN-02	Every *ESRD* is a kind of *Stage* Every *Loss* is a kind of *Stage* Every *Failure* is a kind of *Stage* Every *Injury* is a kind of *Stage* Every *Risk* is a kind of *Stage*	SubClassOf*(:ESRD :Stage)* SubClassOf*(:Loss :Stage)* SubClassOf*(:Failure :Stage)* SubClassOf*(:Injury :Stage)* SubClassOf*(:Risk :Stage)*
KID-CON-01	*DisjointClasses()*	DisjointClasses*(:ESRD :Loss)* DisjointClasses*(:ESRD :Failure)* DisjointClasses*(:ESRD :Injury)* DisjointClasses*(:ESRD :Risk)* DisjointClasses*(:Loss :Failure)* DisjointClasses*(:Loss :Injury)* DisjointClasses*(:Loss :Risk)* DisjointClasses*(:Failure :Injury)* DisjointClasses*(:Failure :Risk)* DisjointClasses*(:Injury :Risk)*
KID-FUN-03	*DataProperty()* *DataPropertyDomain()* *DataPropertyRange()*	Declaration(DataProperty*(:Sex)*) DataPropertyDomain*(:Sex :GFR)* DataPropertyRange*(:Sex xsd:integer)*
	sex1 is a *Sex* And, configure data property assertion directly through GUI in the tool.	Declaration(NameIndividual*(:sex1)*) DataPropertyAssertion*(:* *Sex :sex1 "Female"^^xsd:string)*

Fig. 9. The ontology syntax of KFD.

to be a domain and *xsd:integer* to be a class range including the restriction of data property is 0-300 (*0-300^^ xsd:integer*). Another example is *Sex* which has a domain to be class *GFR* and a range to be *xsd:string*. For this property, two individuals or instances are defined *Female* and *Male* to represent the gender of the patient.

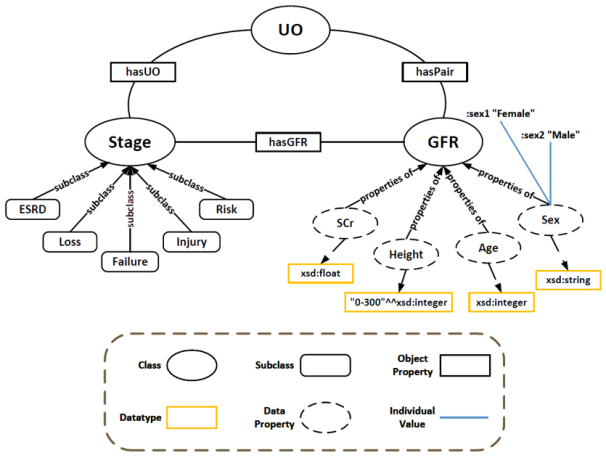


Fig. 10. The ontology structure of KFD.

Test Case Generation: Fig. 11 demonstrates the classification tree resulted from CTM technique. In the tree, *GFR Module* as a system name is considered to be a root node, before the subsystem *GFR Interpreted* is defined as a terminal classification in the next level. Variables *Sex*, *Age*, *Height*, *SCr*, *GFR*, *UO*, and *Stage* related to this function are defined in the below level in the tree. These variables are considered to generate test cases by using ECP in the later step. The terminal class of each terminal classification defining the possible range of value is used to be a partition for generating test cases and test data in ECP.

Table V is the test case generation result for testing *GFR interpreted* operation. Consider test case #1, it is a valid test case for GFR calculation of a female patient less than 18 years old. This result of the stage of kidney failure interpreted as the Injury. Furthermore, the generated test cases cover in the case

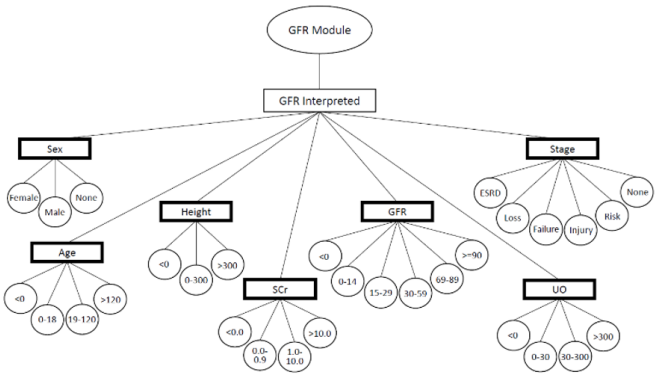


Fig. 11. The classification tree of KFD.

of invalid. In test case #144, it is an invalid test case because the data value is out of the range of interest.

TABLE V. EXAMPLE OF TEST CASES AND TEST DATA FOR TESTING RETURN OPERATION

TC#	Sex	Age	Height	Scr	GFR	UO	Stage	Comments
1	Female	10	142	0.8	73	450	Injury	Valid
2	Female	18	165	4.5	15	27	Loss	Valid
...	...	...	...	...	...	...	...	...
61	Male	177	0.3	356	30	555	Risk	Valid
62	Male	65	104	7.2	7	10	ESRD	Valid
...	...	...	...	...	...	...	...	...
144	None	200	1140	110.2	11558	65487	None	Invalid

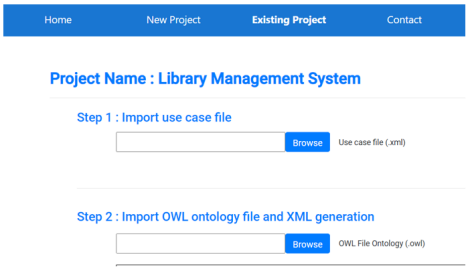
IV. PROOF OF CONCEPT

A. Tool Development

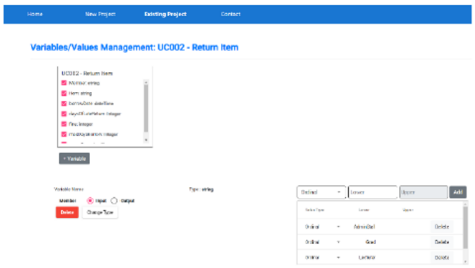
To demonstrate the effectiveness of *ReqOntoTesGen* approach, a prototype tool was developed. The developed tool is a Java based web application using Node.js 16.14.0[4] JavaScript runtime environment that is well known and widely used.

Fig. 12 demonstrates an example of our developed tool. Fig. 12a shows the screen for importing the necessary XML file. Two types of XML files are imported into the tool (1) the XML file of use cases indicating functionality from SRS documents and (2) the OWL file of requirement specifications created by the ROO tool. Then, the XML file of data dictionary is automatically generated from the OWL file. All extracted variables and their range value from the XML file of data dictionary are analysed. This includes variable name, variable type and variable range value as shown in Fig. 12b. The next step is the decision tree creation in the case that the system uses the condition for decision making on the operation process. The condition and decision of the decision tree can be adjusted as necessary as demonstrated in Fig. 12c. This decision tree is considered with the transformed data dictionary to generate test cases by the CCTM technique in the tool. The classification tree and equivalence partition of related variables resulted from CCTM are shown on the screen as demonstrated in Fig. 4 and 5 respectively. Test cases are automatically generated from this classification tree and equivalence partition. The result of test case generation is shown on the screen as in Fig. 12d.

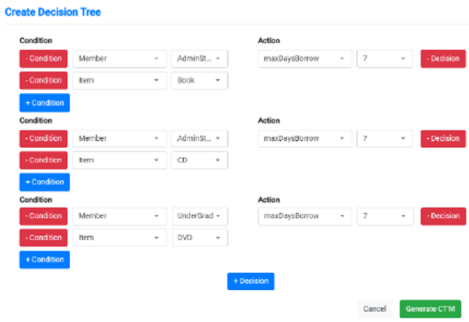
[4]https://nodejs.org/en/about/



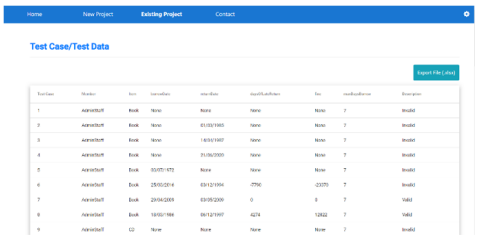
(a) The import file management screen



(b) The variables and values management screen



(c) The screen for adding decision tree



(d) The test case / test data generation screen

Fig. 12. Example of the tool screens.

B. Tool Validation

To validate the developed tool whether all functionalities of the tool perform correctly according to the *ReqOntoTestGen* framework proposed in Section III. Three test scenarios corresponding to three steps of the framework were conducted as shown in Table VI. This includes 1) *TS-01 Validate OWL transformation to XML function* with 13 relevant designed transformation rules. 2) *TS-02 Validate variable and decision tree management function* to validate the correctness of extracted variables from the XML file of use case and data dictionary together with the decision tree information. 3) *TS-03 Validate test case generation function* that validates the correctness of test case generation with CCTM techniques.

TABLE VI. THE RESULT OF TOOL TESTING

Test Scenario	Result	Revision
TS-01 Validate OWL transformation to XML function	***Fail***	Pass
TS-02 Validate variable and decision tree management function	Pass	-
TS-03 Validate test case generation function	Pass	-

Table VI demonstrates the validation result. This led us to reveal an error that occurred in *TS-01*. The validation result of *TS-01* was *Fail* because the individual value transformation rule generated the wrong order in the XML element as shown in Fig. 13a This resulted in the data property element e.g., *maxDaysBorrow* to be generated outside the class element. This violated our designed transformation rules in which the data property needs to be inside the class. This led us to restructure the rule according to the design. Fig. 13b shows the corrected version of this error that resulted in this testing scenario to be *Pass*.

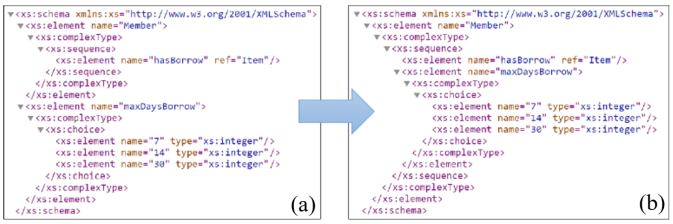


Fig. 13. The error of *TS-01*.

C. Tool Evaluation

We evaluated the effectiveness of our proposed approach with two real case studies, Library Management System (LMS) and Kidney Failure Diagnosis (KFD) subsystem. This evaluation is divided into two parts that are 1) effectiveness evaluation and 2) satisfaction evaluation.

1) Effectiveness evaluation: The precision, recall and F-measure computation were calculated by comparing the result produced by the manual operation and automated tool. The computation metrics were adapted from [35] as follows.

$$Precision = \frac{|\{Expert\ Identified\} \cap \{Tool\ Identified\}|}{|\{Tool\ Identified\}|} \times 100 \quad (1)$$

$$Recall = \frac{|\{Expert\ Identified\} \cap \{Tool\ Identified\}|}{|\{Expert\ Identified\}|} \times 100 \quad (2)$$

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (3)$$

TABLE VII. THE RESULT OF IMPACT ANALYSIS

System	# Test cases		Precision	Recall	F-measure
	identified by an expert	*identified by the tool*			
LMS	300	160	100%	53.33%	69.56%
KFD	144	144	100%	100%	100%

Table VII demonstrates the comparison results between the expected test case manually created by experts and the actual test case automatically generated by the tool. Considering the calculated F-measure with precision and recall of KFD case study, the accuracy of the automatic tool performing with this case study is very high. This is because KFD case study is not a complex case study compared to LMS case study. However, considering the calculated F-measure, with precision and recall of LMS, they are quite low. We have found that in the manual design of test cases by experts, the out of range of variable *borrowDate* and *returnDate* was identified as invalid partition. This led to 300 (5*4*3*5) test cases to be created. However, after we revealed this case, we discovered that this type of *dateTime* variable has the range of time from *"Begin of Date"* to *"End of Date"* that can be selected at any time for testing. Therefore, it is impossible to be *"Out of Range"*. This led us to recalculate the number of created test case after cutting these *"Out of Range"* partition (partitions 12 and 13 in Fig. 14) and resulted in this recalculation to be the same as calculated by the tool.

Partition	Num of Partition	Valid Data				Invalid Data		
Member (input)	5	AdminStaff (1)	Grad (2)	Lecturer (3)	UnderGrad (4)	None (5)		
Item (input)	4	Book (6)	CD (7)	DVD (8)		None (9)		
borrowDate (input)	3	Begin of Date – End of Date (10)				None (11)		Out of Range (12)
returnDate (input)	5	hasBorrowDate – (13) (hasBorrowDate + hasMaxDaysBook)		(hasBorrowDate + (14) hasMaxDaysBook) – End of Date		Begin of Date – (15) hasBorrowDate	None (16)	Out of Range (17)
maxDaysBorrow (fix rate)	4	7 (18)	14 (19)	30 (20)		None (21)		
daysOfLateReturn (output)	3	1 – Late Return (22)		None (23)		Smallest Number – 1 (24)		
fine (output)	3	1 x Fine Rate – (25) daysOfLateReturn x Fine Rate		None (26)		Smallest number – Fine Rate (27)		

Fig. 14. The partition of variable change.

2) Satisfaction evaluation: The satisfaction of our proposed approach and tool was evaluated with a wide range of experts that have at least five years in software engineering and software testing. This included two programmers and three testers. We designed questions for satisfactory evaluation, it consists of four categories for evaluation, *Q1) Functionality*, *Q2) Efficiency and reliability*, *Q3) Usability*, and *Q4) Ability and applicability* that is shown in Table VIII.

TABLE VIII. SATISFACTION QUESTIONS

Questions	Average
Q1. Functionality	
Q1.1 The function can operate accurately and appropriately.	Likert scale (Mandatory)
Q1.2 The function can operate with each other.	Likert scale (Mandatory)
Q1.3 The function can operate according to the users' requirements.	Likert scale (Mandatory)
Q2. Efficiency and reliability	
Q2.1 The prototype can appropriately process the test cases.	Likert scale (Mandatory)
Q2.2 The prototype can increase the structure of functional requirements.	Likert scale (Mandatory)
Q2.3 The prototype can reduce errors caused by functional requirements.	Likert scale (Mandatory)
Q2.4 The prototype can work completely.	Likert scale (Mandatory)
Q3. Usability	
Q3.1 The prototype is easy to learn and understand.	Likert scale (Mandatory)
Q3.2 The prototype is easy to use, and the function is not complicated.	Likert scale (Mandatory)
Q4. Ability and applicability	
Q4.1 The prototype can be applied in the system or other case studies.	Likert scale (Mandatory)
Q4.2 The prototype can be easily installed and used.	Likert scale (Mandatory)

The Likert scale was used to design the levels of satisfactory for each question including Strongly Agree (5), Agree (4),

Neutral (3), Disagree (2), and Strong Disagree (1) respectively. The evaluation result of the satisfactory is shown in Fig. 15.

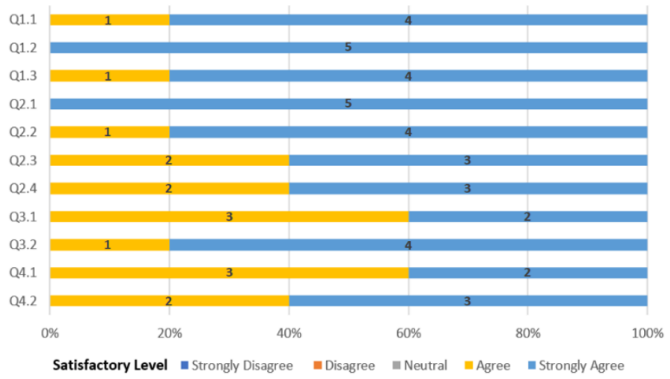


Fig. 15. Results of the four Likert scale questions.

As can be seen in Fig. 15, most of the specialists strongly agreed that our developed tool provides an accurate and appropriate functionality and interoperation ability (with an average of Q1.1-1.3, 86.66% of them strongly agree). They also strongly agreed that the tool is efficient and reliable (with an average of Q2.1-2.4, 75% of them strongly agree). Considering the usability (Q3), 60% of specialists strongly agreed that the functions provided by the tool are not complicated and easy to use and understand. Furthermore, the specialists satisfied the prototype in terms of its ability and applicability (Q4) from the agree level (with half of them satisfied at the strongly agree level). Overall, we can conclude that the specialists were mostly satisfied our *ReqOntoTestGen* approach and its corresponding tool.

V. Lesson Learned and Discussion

In this section, we discuss the benefits of the proposed *ReqOntoTestGen* approach for generating test cases with the software requirement ontology. This section also shares lessons learned achieved from our implications of practical implementation and experiment. The approach influences the benefits according to our research questions as follows.

- It provides a systematic mechanism and framework to generate test cases from a very clear structure of functional requirements encoded in the form of ontology. The application of ROO tool in the framework enables the unstructured requirement to be transformed into more structured and clearer requirements before generating ontology. This results in the complexity of requirement structure to be reduced and the ambiguity of the terminology used in the ontology to be eliminated as discussed in [8][11][16][36][37], This also guarantees that the main causes of errors in software testing that are mainly from requirements to be eliminated and the correct test cases that satisfied user requirements to be generated.
- CCTM test case generation technique implemented in the framework to construct test cases influences benefits that the number of test cases is reduced with maximizing testing coverage. As claimed in [14][15] we have discovered from our implemented experiences that CTM technique in CCTM enables the redundant test cases to be eliminated, On the other hand, ECP technique in CCTM expands the possible range value both valid and invalid cases. This led to the testing coverage to be increased.
- *ReqOntoTestGen* approach provides a semi-automatic prototype tool that implemented the algorithm to generate test cases from well-defined ontology. The results of the experiment by comparing the manual test case generation and automatic test case generation by the tool with two case studies: LMS and KFD can guarantee the correctness, effectiveness, and accuracy of the proposed approach and tool. Furthermore, the efficiency and potential use in the future are confirmed by the evaluation result from experts.

However, as suggested by the practical specialists from the satisfaction evaluation, there are limitations of the approach. Firstly, the proposed approach provides the semi-automated prototype tool in which the conceptual ontologies from the requirement specification resulted from the ROO tool need to be input manually into the prototype for test case generation. Furthermore, the experiment for the prototype validation and evaluation is based on two real case studies. It needs to be evaluated with other different domain of case studies.

VI. Conclusion and Future Work

This paper presents a novel approach, *ReqOntoTestGen*, to enhance the efficiency of traditional testing techniques. It provides a semi-automatic framework that integrates ontology engineering with software testing for test case generation. The effectiveness and efficiency of our *ReqOntoTestGen* approach and framework is demonstrated by the developed prototype tool. The experiment results with the implementation of two case studies have shown that the Control Natural Language (CNL) from the ROO tool used in our tool enables the unstructured functional requirements that may lead the generated test cases to be inconsistent to the users' needs to be more structured and clearer, before transforming them into the OWL conceptual ontology. This OWL file is, then, transformed automatically into the XML file of data dictionary. CCTM technique implemented in the tool creates the automatic test case generation environment in which test cases are generated automatically from the transformed XML file of data dictionary with the decision tree. This influences the benefits that the redundant test cases to be eliminated and the coverage of the test case generation to be increased. Furthermore, the evaluation result has shown that our developed tool has a high degree of validity, accuracy and satisfaction level from the practical specialist perspective. As a result of this, it can be confirmed that our proposed approach contributes a hybrid test case generation technique with a software requirement ontology engineering that both meets the users' need and covers all possible testing scenarios.

For the future work, to increase the capability and reliability of the developed prototype, it needs to link with the ROO tool which can automatically input the conceptual ontology resulted from the ROO to the prototype. Furthermore, the evaluation of the prototype with different domain of case studies is still open as another research issue.

DEPLOYMENT AND AVAILABILITY

The developed tool with the user guide document and source of example case studies is available at https://sites.google.com/phuket.psu.ac.th/reqontotestgen/.

REFERENCES

[1] R. Patton, *Software Testing (2nd Edition)*. USA: Sams, 2005.

[2] Z. Liu and R. Kang, "Imperfect debugging software belief reliability growth model based on uncertain differential equation," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 735–746, 2022.

[3] D. Dermeval, J. Vilela, I. I. Bittencourt, J. Castro, S. Isotani, P. Brito, and A. Silva, "Applications of ontologies in requirements engineering: A systematic review of the literature," *Requirements Engineering*, vol. 21, no. 4, p. 405–437, nov 2016. [Online]. Available: https://doi.org/10.1007/s00766-015-0222-6

[4] K. Thongglin, S. Cardey, and P. Greenfield, "Thai software requirements specification pattern," in *2013 IEEE 12th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT)*, 2013, pp. 179–184.

[5] G. Liargkovas, A. Papadopoulou, Z. Kotti, and D. Spinellis, "Software engineering education knowledge versus industrial needs," *IEEE Transactions on Education*, vol. 65, no. 3, pp. 419–427, 2022.

[6] P. Jorgensen, *Software Testing: A Craftsman's Approach*, 3rd ed. Boca Raton, NY: Auerbach Publications, 5 2013.

[7] K. Mokos, T. Nestoridis, P. Katsaros, and N. Bassiliades, "Semantic modeling and analysis of natural language system requirements," *IEEE Access*, vol. 10, pp. 84 094–84 119, 2022.

[8] C. Keet. (2020) An introduction to ontology engineering. [Online]. Available: https://people.cs.uct.ac.za/ mkeet/OEbook/

[9] W. W. Sim and P. Brouse, "Towards an ontology-based persona-driven requirements and knowledge engineering," *Procedia Computer Science*, vol. 36, pp. 314–321, 2014, complex Adaptive Systems Philadelphia, PA November 3-5, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050914013489

[10] K. Siegemund, U. Assmann, J. Pan, E. Thomas, and Y. Zhao, "Towards ontology-driven requirements engineering," in *Proceeding of the 10th International Semantic Web Conference (ISWC)*, 10 2011.

[11] N. S. Harsha, C. N. Kumar, V. K. Sonthi, and K. Amarendra, "Lexical ambiguity in natural language processing applications," in *2022 International Conference on Electronics and Renewable Systems (ICEARS)*, 2022, pp. 1550–1555.

[12] S. Popereshnyak and A. Vecherkovskaya, "Modeling ontologies in software testing," in *2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT)*, vol. 3, 2019, pp. 236–239.

[13] S. Charoenreh and A. Intana, "Enhancing software testing with ontology engineering approach," in *2019 23rd International Computer Science and Engineering Conference (ICSEC)*, 2019, pp. 186–191.

[14] B. Ramadoss, P. Prema, and S. R. Balasundaram, "Combined classification tree method for test suite reduction," in *Proceedings on International Conference and workshop on Emerging Trends in Technology (ICWET, 2011)*, no. 11, 2011, pp. 27–33.

[15] A. Intana, K. Laosen, and T. Sriraksa, "An automated impact analysis approach for test cases based on changes of use case based requirement specifications," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 1, 2023. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2023.01401105

[16] R. Denaux, "Intuitive ontology authoring using controlled natural language," Ph.D. dissertation, School of Computing, University of Leeds, 2013.

[17] K. Athiththan, S. Rovinsan, S. Sathveegan, N. Gunasekaran, K. S. A. W. Gunawardena, and D. Kasthurirathna, "An ontology-based approach to automate the software development process," *2018 IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*, pp. 1–6, 2018.

[18] D. Tsoukalas, M. Siavvas, M. Mathioudaki, and D. Kehagias, "An ontology-based approach for automatic specification, verification, and validation of software security requirements: Preliminary results," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2021, pp. 83–91.

[19] V. Castañeda, L. Ballejos, M. Caliusco, and M. Galli, "The use of ontologies in requirements engineering," *Journal of Researches in Engineering*, vol. 10, pp. 2–8, 01 2010.

[20] A. A. Alsanad, A. Chikh, and A. Mirza, "A domain ontology for software requirements change management in global software development environment," *IEEE Access*, vol. 7, pp. 49 352–49 361, 2019.

[21] H. Alrumaih, A. Mirza, and H. Alsalamah, "Domain ontology for requirements classification in requirements engineering context," *IEEE Access*, vol. 8, pp. 89 899–89 908, 2020.

[22] L. Yang, K. Cormican, and M. Yu, "Ontology learning for systems engineering body of knowledge," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 2, pp. 1039–1047, 2021.

[23] E. Souza, R. Falbo, and N. Vijaykumar, "Using ontology patterns for building a reference software testing ontology," in *2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops*, 2013, pp. 21–30.

[24] E. F. Barbosa, E. Y. Nakagawa, and J. C. Maldonado, "Towards the establishment of an ontology of software testing," in *International Conference on Software Engineering and Knowledge Engineering*, 2006.

[25] P. Chen and A. Xi, "Research on industrial software testing knowledge database based on ontology," in *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, 2020, pp. 425–429.

[26] L. Olsina, G. Tebes, D. Peppino, and P. Becker, "Approaches used to verify and validate a software testing ontology as an artifact," in *2020 IEEE Congreso Bienal de Argentina (ARGENCON)*, 2020, pp. 1–8.

[27] S. Banerjee, N. C. Debnath, and A. Sarkar, "An ontology-based approach to automated test case generation," *SN Computer Science*, vol. 2, no. 1, pp. 1–12, 2021.

[28] Y. Wang, X. Bai, J. Li, and R. Huang, "Ontology-based test case generation for testing web services," in *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*, 2007, pp. 43–50.

[29] A. W. Crapo and A. Moitra, "Using owl ontologies as a domain-specific language for capturing requirements for formal analysis and test case generation," in *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, 2019, pp. 361–366.

[30] S. Ul Haq and U. Qamar, "Ontology based test case generation for black box testing," in *Proceedings of the 2019 8th International Conference on Educational and Information Technology*, ser. ICEIT 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 236–241. [Online]. Available: https://doi.org/10.1145/3318396.3318442

[31] A. Jounaidi and M. Bahaj, "Designing and implementing xml schema inside owl ontology," in *2017 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, 2017, pp. 1–7.

[32] O. E. Hajjamy, L. Alaoui, and M. Bahaj, "Xsd2owl2 : Automatic mapping from xml schema into owl 2 ontology," *Journal of Theoretical and Applied Information Technology*, vol. 95, no. 8, pp. 1781–1796, 2017.

[33] N. Yahia, S. Mokhtar, and A. Ahmed, "Automatic generation of owl ontology from xml data source," *International Journal of Computer Science Issues*, vol. 9, 06 2012.

[34] D. Pilone and N. Pitman, *UML 2.0 in a Nutshell*. OŔeilly Media, Inc., 2005.

[35] K. M. Ting, *Precision and Recall*. Boston, MA: Springer US, 2010, pp. 781–781. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_652

[36] J. Henarejos-Blasco, J. A. García-Díaz, O. Apolinario-Arzube, and R. Valencia-García, "Cnl-rdf-query: A controlled natural language interface for querying ontologies and relational databases," in *Proceedings of the 10th Euro-American Conference on Telematics and Information Systems*, ser. EATIS '20. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3401895.3402064

[37] R. Denaux, V. Dimitrova, A. G. Cohn, C. Dolbear, and G. Hart, "Rabbit to owl: Ontology authoring with a cnl-based tool," in *Controlled Natural Language*, N. E. Fuchs, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 246–264.