UTM
UNIVERSITI TEKNOLOGI MALAYSIA
INTERNATIONAL JOURNAL OF
INNOVATIVE COMPUTING
ISSN 2180-4370
Journal Homepage : https://ijic.utm.my/

Ontology of Mutation Testing for Java Operators

Sherolwendy Sualim, Radziah Mohamad, Nor Azizah Saadon
Faculty of Computing, Universiti Teknologi Malaysia
81310, UTM Johor Bahru, Johor, Malaysia

Submitted: 12/01/2018. Revised edition: 16/05/2018. Accepted: 21/05/2018. Published online: 31/05/2018

Abstract—Operators are special characters within the Java language to manipulate primitive data type. Java operators can be classified as unary, binary and ternary. The design of Java operator sometimes becomes confusing when it comes to testing tools as they had the same function with different label in every testing tool. Therefore, in order to map the knowledge of operators correctly, this research has proposed ontology that is dedicated to mutation testing as a means to define the formal specification of concepts and documentation of knowledge of Java operators. Existing papers on ontology did not specify further on entities and properties of operators. Some papers only focus on mutation testing but not the operators. Thus, this paper will present the ontology clearly with the aim to ease end user to identify and understand every classes, properties and relations in Java operators.

Keywords — Mutation testing, ontology, Java operators

I. INTRODUCTION

For decades, knowledge representation has been the focus of interest as more methods and techniques have emerged during that time where ontologies currently are one of most popular and widespread [1]. The focus of modern information systems is moving from data processing towards concept processing, meaning that the basic unit for processing is being less and less an automatic piece of data and is becoming more like a semantic concept which carries an interpretation and exists in a context with other concepts. Ontology is defined as a formal, and explicit specification of a shared conceptualization [2], [3]. Building ontology especially for a specific domain can be started by scratching a new ontology [4] or just modifying an existing ontology [5]. There are three main components in ontology namely concepts, individuals and properties where concepts define aggregation of things, and properties link the instance of concepts, and individuals.

Designing ontology is very essential especially in capturing the concept, properties and interrelationship in a context. Modeling context ontology is possible because it can be considered as specific kind of knowledge [9]. Since ontology can be described as sharing comprehension of specific domain interest, it can be used as a basic structure to solve the problem in knowledge sharing [10]. Ontology also helps in improving communication between humans and computers. So, these can be further classified into several sections namely assisting communication among human agents, achieving interoperability, or improving the quality of tasks [11].

Mutation testing is basically a white-box technique that can generally be used in software testing to check syntax error in the programs. Mutation testing starts with injecting the original program with a fault to mutate it [15]. Then, a mutation operator is applied before checking whether the test identifies this fault [52]. These changes lead to a program variant which is called mutant. The fundamental aspect is to check whether the test suite is able to detect the mutant. It can be said that the mutant is detected or killed if the test run fails. Otherwise, the mutant is alive.The use of mutation testing is to improve a test suite by providing tests for undetected mutants. After mutations are applied to a program, then the instigator can check whether the test suite detects mutations or not. If the results show a set of undetected mutants, the programmer may attempt to add or modify existing tests until satisfactory results are attained.

The effectiveness of mutation testing depends on the types of faults that the mutation system is designed to represent. Since mutation testing uses mutation operators to implement faults, the quality of the mutation operators is crucial to the effectiveness of mutation testing. Two general types of mutation operators for Java namely

method operator and class operator. Method operators had been used in previous mutation tools for programming language besides Java [52]. These operators are applied to statements, operands and operators which perform actions such as modification, replacement and deletion. For class level operators, they are related to inheritance, polymorphism and Java-specific object oriented features [52]. Although mutation testing had a rich history, most of mutation operators have been developed for procedural programs.

The mutation operators are designed and expressed specifically for Java language [53]. This is important because mutation operators must take the semantics of a programming language into account. The current set of method and class operators are insufficient to evaluate concurrent Java source code [52]. To execute mutation testing with operators, they should be selected based on the characteristic of the program to be tested. Therefore, the quality of the mutation operators is the key to mutation testing. Mutation operators are classified by the language constructs they are created to alter. Traditionally, the scope of operators was limited to the method level [54]. Some previous mutation operators have been developed based on experience of testers. All behaviours of mutation operators fall under one of three categories namely delete, insert or change a target syntactic element [55].

The paper is organized as follow. Section 2 discusses the related work, and Section 3 discusses the methodology used to build ontology. Section 4 provides an overview of proposed ontology and Section 5 discusses about ontology consistency checking. The last two sections state the discussion and conclusion of the work.

II. RELATED WORK

Mutation testing is basically a white-box technique that can be used in software testing to ensure that programs are free from the syntax errors. Mutation testing is a software testing that is originally proposed by Hamlet [15]. Mutation testing is based upon seeding the implementation (original program) with a fault (mutating it), by applying a mutation operator, and determining whether the testing identifies this fault [16]. A program that is mutated is called a mutant and it is said to kill the mutant if any of the test case can distinguish between mutant and original program. But if there is no test case that can distinguish between mutant and original program then mutant is still alive.

Mutation testing process consists of three common phases which are mutant generation, test cases to mutant execution and results evaluation. Mutant is the version of a program that is generated from mutation operator that changed one or more source code line. A mutation operator is a group of rules that is used to select or manipulate the line of source code [17]. Mutation is carried out by applying a set of mutation operators to a ground string [18]. The ground string is actually expressed in grammar. Mutation operator is defined as a rule that specifies syntactic variations of string that are generated from a grammar [19]. These operators can also be applied directly to grammar without the existence of a ground string. Thus, mutation can be used to generate both valid and invalid string that differs from ground string. Both string cases are called mutants. Mutation testing has been applied to software code, particularly to Java [20][21]. Previous research has identified a set of operators for mutation [18]. However, in practice mutation is sensitive to the underlying mutants that it is using. In other words, the set of the realized operators can have major impact on both scalability and effectiveness of the technique. Therefore, it is mandatory to equip mutation testing tool with a comprehensive set of mutants that can adequately measure thoroughness and act as a practical test.

Traditional mutation testing introduced error in the code when operated at the syntax level. But, traditional mutation operators are not sufficient for testing Object-Oriented (OO) programming languages like Java [22][20]. This is because the faults represented by the traditional mutation operators are different from OO environment due to the differences in OO programming structure. Besides, there are new faults introduced by OO-specific features, inheritance and polymorphism. The design of Java operators are not strongly influenced by previous work as the first design operators [23] using Hazard and Operability Studies (HAZOP). Based on these plausible faults, 20 Java mutation operators are designed with six groups. Then, Class Mutation is introduced to OO programs that targets faults related to OO-specific features [24]. In Class Mutation, the first three mutation operators are selected to represent Java OO-features and later ten mutation operators are added [25]. The Class mutation operators are extended to 15 which are grouped into four types [26]: polymorphism, overloading, information hiding and exception handling.

To increase the effectiveness of all mutation operators, 24 comprehensive Java mutation operators are introduced and they are classified into six groups: information hiding, inheritance, polymorphism, overloading, Java specific features and common programming mistakes [27]. There are also alternative approaches to define mutation operators for Java, which is to inject faults into Java utility libraries especially container library and iterator library [28]. All of these approaches keep growing due to concurrent Java environment. In general process of mutation testing, the mutant is generated by executing mutation operators to the source code program. Many mutation operators are already being defined, for instance, Mothra has defined 22

operators [29], while latest research has compiled 24 operators [30][31] under statement level, class level and method level. The generation of operators will also increase as more operators will be invented. Hence, changes in operators lead to new tools invention or upgrade.

In 2006, Ma *et al*. invents a mutation system for java called MuJava [56]. It is an automated tool that uses two types of mutation operators, method level and class level. This tool supports the entire mutation process of Java program. However, this tool is relatively slow when it generates and runs lots of mutants. It is an ongoing project that still needs some improvement to be a more effective tool. In 2009, Schuler and Zeller introduce an efficient mutation testing for Java called Javalance [59]. This tool is built for efficiency and effectively addresses the problem of equivalent mutants. Then, Madeyski and Radyk (2010) invent a mutation testing tool called Judy [57]. Their objective was to speed up mutation testing. This testing tool encounters problems when the operators are extended to inter-method, intra-class and inter-class operators as it requires the combination of meta-mutant. The tool that needs to offer an even wider set of mutation operators is under active development. Just and Schweiggert in 2011 introduce an efficient and extensible tool for mutation analysis in a Java compiler [58]. It is a fault seeding and mutation analysis system integrated into Java compiler. But, they plan to implement new mutation operators and enhance domain specific language. This tool is not ready for industrial practice. The latest tool is PIT, invented by Coles *et al*. (2016) as a practical mutation testing tool [43]. Since the recent years, many mutation testing tools have been developed mainly to support research in this area. MuJava and Major are the most popular among others. Unfortunately they were built to support research projects thus, their practical use was limited [60].

Ontologies, on the other hand, have a number of uses where primarily they describe some domain of knowledge from a specific perspective. They act more likely to be a vocabulary as similar to database. Ontologies have become a basis of knowledge representation in many application fields, from web searches to medical and local domains [6]. Besides, they also are used for decision support [7], therefore it is important to be completed without any errors as possible. It is widely known that there is no proper way of defining ontology. The definition really depends on the domain where the purpose of which the ontology is intended [8]. The application of ontology in mutation testing were first applied for Web Service which were targeting specific XML-based language features, for example in OWL-S specification language [12][13]. OWL-S introduces a semantic workflow specification using an ontology specification language. Last time, OWL-s was analyzed by other researcher to composite Web service fault patterns [12] and then came out with OWL-s input type mutant operators and OWL ontology mutation operator. Then, another approach is proposed based on OWL-s requirement model [14]. Hence, ontology is actually an important constituent in semantic web layered architecture. Problem-solving methods, domain-independent applications, and software agents, all of them are using ontologies and knowledge bases built from ontologies as data. Without ontology, it is impossible to maintain relationships among the real world entities as various operations can be performed on the ontology. With its formal nature and philosophical aspects of handling real world scenarios, ontology also acts as a linking medium between human and machine.

Ontology is not a new element in mutation testing. Nonetheless, this paper introduced ontology for mutation testing as a solution to the problem of mutation operators. The distinguished abilities of ontology such as sharing common understanding of information structure among end user, enabling reuse of domain knowledge, making domain assumptions explicit, separating domain knowledge from operational knowledge, and analyzing knowledge, [32] are among the reasons it is chosen for a better solution. Subsequently, ontology will synchronize and ease the definition of operator for understanding. Ontology is also very flexible and it is totally suitable for future generation of mutation testing operators or any changing due to future research output. Hard-coding in programming language code makes implementations not only hard to find and understand but also hard to change as well. Ontology has made the implementation easier as explicit specification of domain knowledge are very useful for new user to learn domain mean. In contrast, this ontology may solve other limitation issues by previous methods or techniques of mutation testing operators.

III. METHODOLOGY

For the development of ontology, this paper uses METHONTOLOGY [49] as the methodology. METHONTOLOGY is among the most comprehensive ontology engineering methodology as it is building ontologies either from scratch, reusing other ontology as they are, or by process of re-engineering them. This framework enables the construction of ontologies at the knowledge level like the conceptual level, as opposed to the implementation level. This framework consists of several processes namely identification of the ontology development process, a life cycle based on evolving prototypes, and specification steps by methodology itself. So, generally this method described the process in detail to build ontology for centralized ontology based systems.

There are many advantages of using METHONTOLOGY in building ontologies. This methodology emphasizes the reuse of existing domain and upper-level ontologies and proposed to use for formalization, a set of intermediate representations that later can be automatically transformed into formal languages [48]. This includes specification, conceptualization, formalization, implementation and maintenance.

Therefore, this methodology is suitable for developing ontologies at knowledge level.

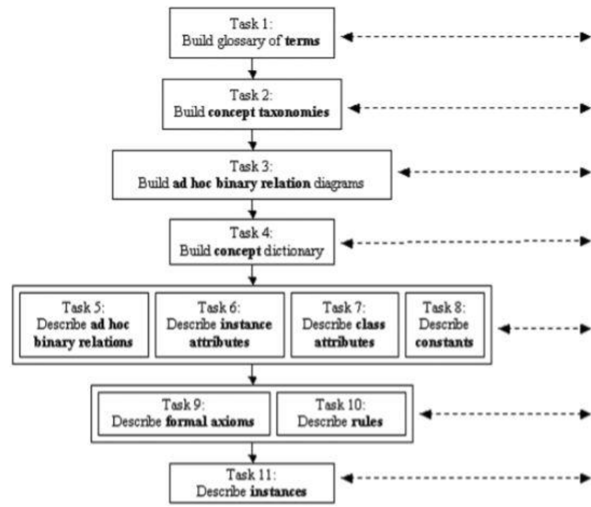

Fig. 1. Tasks of the conceptualization activity according to METHONTOLOGY [49]

The figure above emphasizes ontology components that are built inside each task. There are concepts, attributes, relations, constants, formal axioms, rules and instances. Besides, the figure illustrates the steps that this methodology has proposed for creating such component during conceptualization activity. This is not the sequential modeling process but some order must be followed to ensure the consistency and completeness of the represented knowledge [50].

IV.ONTOLOGY CONSTRUCTION

This proposed ontology explained about Java operators for mutation testing. Mutation operator is important in this case. All operators from previous researchers had been collected and analyzed. The development of ontology was using web ontology language (OWL). OWL is one of the recommendations from World Wide Web Consortium (W3C) to develop ontologies. OWL makes it possible to describe concepts as it has a richer set of operators like intersection, union and negation. Fig. 2 shows the diagram with direct relationships among the concepts in the ontology.

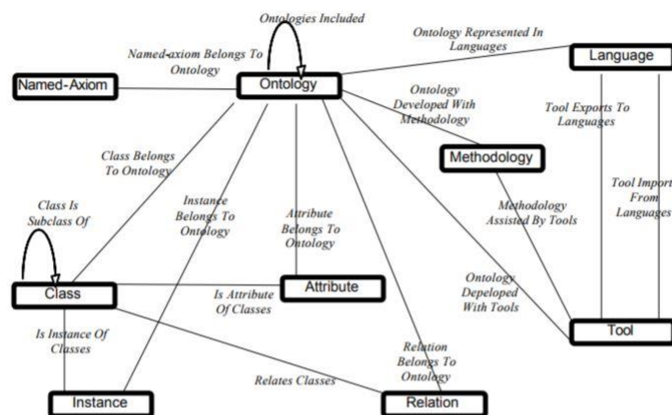

Fig. 2. Direct relationship among concepts of ontology [47]

Ontology is a formal explicit description that consists of individuals, properties and classes. Individuals represent objects in the domain in which raise interest. Properties are binary relations on individuals where two individuals are linked together. Properties of each concept also describe various features and attributes of the concept. Classes are interpreted as sets that contain individuals and they are the focus of most ontology. Classes describe concept in the domain. A class can have subclasses that represent concepts that are more specific than the superclass. There are several steps in developing ontology. The steps include defining classes, arranging the classes in a taxonomic hierarchy, defining properties and describing allowed values for these properties, and filling in the values for properties for instance.

A. Entity Extraction

In this proposed ontology, the concepts were extracted first as starting the development of ontology by determining the domain and scope. This proposed ontology are used for the application of Java operators in mutation testing. Concepts can be perceived differently depending on the domain. For example, in context of mutation operators, the concepts can include operator type, operator category, operator characteristic, etc. These concepts are generally organized in taxonomy where inheritance is usually involved. Specific type of annotations and metadata were added later to the document. Table 1 shows the concepts listed in the proposed ontology as well as the description of main concept.

Table 1. Description of Concepts

Name	**Description**
Operator Type	Operator type is defined as the type of operators. There are four main types as mentioned, method level [33][39], class level[33][34][35], traditional[33] and general. General for operator type means the operator falls in the group differ from method, class and traditional.
Operator Category	Operator category is defined as the category of every operator and totally related to operator type. Every operator type has their own category. Method level has five category namely arithmetic, relational, conditional, shift logical, and assignment [33][40]. While class level has five categories called as encapsulation, inheritance, polymorphism, Java specific features, and overloading [33][34]. Traditional has sub traditional and general has others.
Operator Characteristic	Operator characteristic involves all the characteristic related to Java operator in mutation testing. These characteristics include redundant [29], non-redundant [35] and deletion mutation [36]. These three characteristic are highly selected and most reviewed by other researchers and are considered an important element in mutation testing area.
Operator Equivalency	Operator equivalency is actually one of the important elements in mutation testing. Some operator may generate similar data as the original [37] during mutation testing, also known as equivalent.
Operator Example	Operator example is the list of all operators that is already defined for Java program. After reviewing several papers, there are around 60 operators involved [33][34][38].

B. Taxonomy Formation

After the entities extraction, the next step is taxonomy formation. The concepts were arranged in the taxonomic hierarchy and turned to classes of ontology later. Forming taxonomy is important as it helps human to understand the ontology better. Besides, it also acts as a reference for future use of ontology. The taxonomy of proposed ontology for operators in mutation testing is shown in Fig. 3. For the taxonomy, there are five elements sub-classes of Operator02 presented. These five elements are Operator type, Operator characteristic, Operator category, Operator equivalency, and Operator example. These five elements are also called child to Operator02.

Fig. 4 shows the sub-classes for Operator type. There are four types of operator namely Method level, Class level, Traditional, and General. There are six categories of method operators: Arithmetic operator, Relational operator, Conditional operator, Shift operator, Logical operator, and lastly Assignment. There are also five categories listed under class level: Encapsulation, Inheritance, Polymorphism, Java specific features and Overloading. Traditional operator type only has one sub-class called Sub-traditional as there is no specific category for traditional Java operator. Same goes to general type operator with one sub-class called Others. Others mean operators that did not belong to any method, class or traditional types. All these categories are generally the sub-classes of Operator category and shown in Fig. 5.

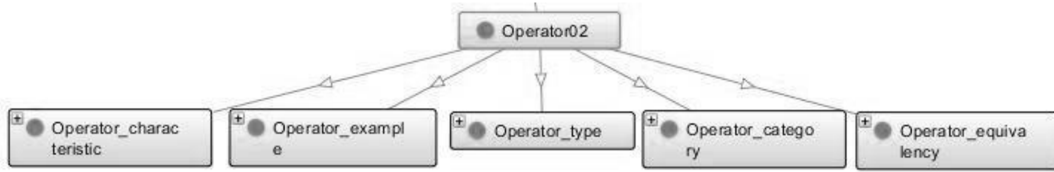

Fig. 3. Taxonomy of ontology for operators

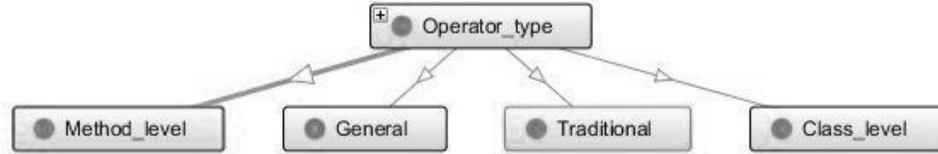

Fig. 4. Sub-classes of Operator type

After listing all types and categories of operators, another part is Operator characteristic and Operator equivalency. As shown in Fig. 6, Operator characteristic focuses on three main classes: Redundant, Non-redundant and Deletion mutation. These three elements are important in operators' classification and every operator belongs to any one of them. Figure 7 shows the Operator equivalency with one sub-class, Equivalent. There are only several operators that have that equivalency where they tend to produce equivalent mutant during mutation testing.

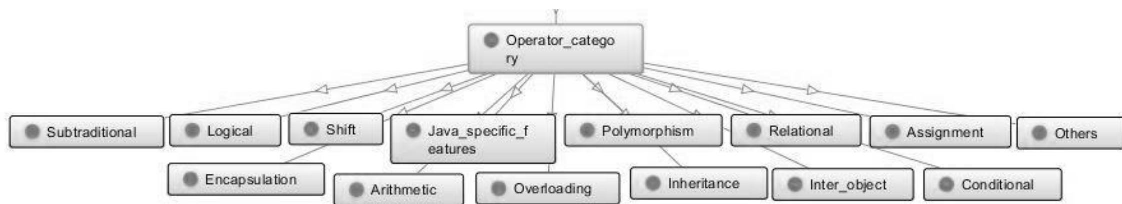

Fig. 5. Sub-classes of Operator category

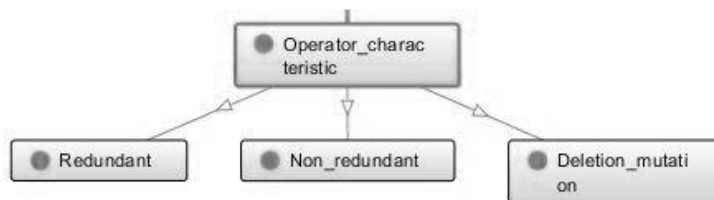

Fig. 6. Sub-classes of Operator characteristic

Fig. 8 shows several sub-classes of Operator example. Operator example consists of all operators from every category. Each of the categories has their own members. For example, Arithmetic has three members namely AOR, AOI, and AOD. While for Class level example, Inheritance contains IHI, IHD, IOD, IOP, IOR, ISI, ISD, and IPC. There are all 61 operators classified and listed under Operator example but only several can be shown in Fig. 8.

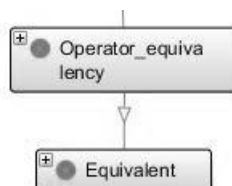

Fig. 7. Sub-classes of Operator Equivalency

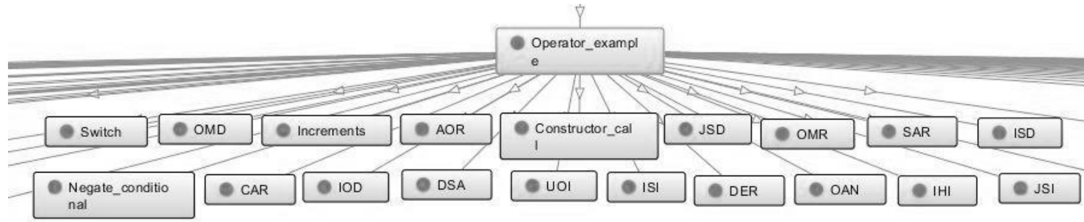

Fig. 8. Sub-classes of Operator example

C. Relationships

As mentioned in previous paragraph, it is possible to define entities, attributes and relationship for ontology. Semantic relationship is a direct relationship that exists in ontology. While, for indirect relationship in ontology it is called semantic association. Relations shows the type of connection among the predetermined concepts. Based on Fig. 9, it shows all the related relations for the operator and the relation between domain and range. *isValueOf* relation is the inverse of *hasValue* relation. Here are some examples of relations:

- Method level *hasCategory* Relational
- Overloading *isCategoryOf* Class level
- Arithmetic *hasExample* AOR
- COR *isExampleOf* Conditional
- ROR *hasCharacteristic* Redundant
- Deletion mutation *isCharacterisitcOf* CDL
- ABS *possible* Equivalent
- Equivalent *isPossibleFor* LCR

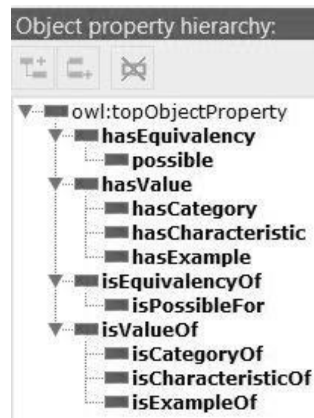

Fig. 9. Properties of ontology

D. Axioms

Axiom is the assertion in logical form that consists of overall theory described in the ontology in application domain. Axioms include statements asserted as a deductive knowledge. Generally, ontology has their own line of axiom and these axioms are actually a statement that is assumed as true. Axioms are used to associate class and property identifier either partial or complete specification as well as giving other information about classes and properties. Table 2, 3 and 4 show the axioms for the context ontology including definition and logical expression.

Table 2. Logical Table

Operator Category		
Concept name	**Axiom description**	**Logical expression**
Method_level	An operator type that has category Relational	$Method_level \cap \ni hasCategory.Relational$
Class_level	An operator type that has category Inheritance	$Class_level \cap \ni hasCategory.Inheritance$
Traditional	An operator type that has category Subtraditional	$Traditional \cap \ni hasCategory.Subtraditional$
General	An operator type that has category Others	$General \cap \ni hasCategory.Others$

Table 3. Logical Table

Operator Example		
Concept name	**Axiom description**	**Logical expression**
Arithmetic	An operator category who is a member of method level has example AOR	$Arithmetic \in Method_level \cap \exists\ hasExample.AOR$
Relational	An operator category who is a member of method level has example ROR	$Relational \in Method_level \cap \exists\ hasExample.ROR$
Conditional	An operator category who is a member of method level has example COI	$Conditional \in Method_level \cap \exists\ hasExample.COI$
Shift	An operator category who is a member of method level has example SOR	$Shift \in Method_level \cap \exists\ hasExample.SOR$
Logical	An operator category who is a member of method level has example LOI	$Logical \in Method_level \cap \exists\ hasExample.LOI$
Assignment	An operator category who is a member of method level has example ASR	$Assignment \in Method_level \cap \exists\ hasExample.ASR$
Encapsulation	An operator category who is a member of class level has example AMC	$Encapsulation \in Class_level \cap \exists\ hasExample.AMC$
Inheritance	An operator category who is a member of class level has example OMC	$Inheritance \in Class_level \cap \exists\ hasExample.OMC$
Polymorphism	An operator category who is a member of class level has example PPD	$Polymorphism \in Class_level \cap \exists\ hasExample.PPD$
Overloading	An operator category who is a member of class level has example OMR	$Overloading \in Class_level \cap \exists\ hasExample.OMR$

Table 4. Logical Table

Operator Characteristic		
Concept name	**Axiom description**	**Logical expression**
Conditional_bound	An operator example who is a member of Conditional has characteristic non redundant	$Conditional_bound \in Conditional \cap \exists\ hasCharacteristic.Nonredundant$
AOI	An operator example who is a member of Arithmetic possible equivalent	$AOI \in Arithmetic \cap \exists\ possible.Equivalent$
COI	An operator example who is a member of Conditional possible equivalent	$COI \in Conditional \cap \exists\ possible.Equivalent$
ROR	An operator example who is a member of Relational has characteristic redundant	$ROR \in Relational \cap \exists\ hasCharacteristic.Redundant$
IHI	An operator example who is a member of Inheritance has characteristic redundant	$IHI \in Inheritance \cap \exists\ hasCharacteristic.Redundant$
Return_values	An operator example who is a member of Others has characteristic nonredundant	$Return_value \in Others \cap \exists\ hasCharacteristic.Nonredundant$
Member_variable	An operator example who is a member of Others has characteristic nonredundant	$Member_variable \in Others \cap \exists\ hasCharacteristic.Nonredundant$
CRP	An operator example who is a member of Subtraditional has characteristic redundant	$CRP \in Subtraditional \cap \exists\ hasCharacteristic.Redundant$
SDL	An operator example who is a member of Subtraditional has characteristic redundant	$SDL \in Subtraditional \cap \exists\ hasCharacteristic.Deletion_mutation$

Table 5. Logical Table

Operator Equivalency		
Concept name	**Axiom description**	**Logical expression**
OAN	An operator example who is a member of Overloading possible equivalent	$OAN \in Overloading \cap \ni possible.Equivalent$
JSI	An operator example who is a member of Java_specific_features possible equivalent	$JSI \in Java_specific_features \cap \ni possible.Equivalent$

V. CONSISTENCY

Redundancy in instances could lead to inconsistent ontology which could reduce the practicality of the ontology itself. For this context ontology, Protégé is used for ontology development [40]. There are some reasoners provided by Protégé to determine the class inconsistencies and discovering implicit information. Pellet [41] and FACT++ [42] reasoner are used for consistency checking. The ontology is cautiously checked first before starting the reasoner. Any inconsistencies in concepts, relationships, and labeling were removed. Then, first open source reasoner, Pellet [41] were applied for reasoning. This open source reasoner is able to handle the growing of OWL ontologies. The validation for context ontology is using Protégé tools. It will send out error messages if the query detected any inconsistent relationship. To make sure the ontology is well checked, this experiment also used FACT++ for evaluating the ontology. The result shows no error which mean there are no redundancies in the concepts. DL expressivity includes attribute language, full existential qualification, inverse properties, and functional properties. Fig. 11 shows the result of consistency checking and it is proved that proposed ontology is coherent and consistent. The context ontology only incorporates five main classes and is fully focused on Java operator context, thus, explained the reason only consistency checking for validation is chosen.

Fig. 10. Consistency checking

VI. ONTOLOGY QUALITY

There is no single best or preferred approach to evaluate ontology. The choice of a suitable approach to ontology must depend on the purpose of evaluation, the application in which the ontology is to be used, and on the aspect of the ontology that needs to be evaluated [61].

For this ontology, the evaluation is based on ten simple criteria proposed by Burton-Jones *et al*. (2005): lawfulness, richness, interpretability, consistency, clarity, comprehensiveness, accuracy, relevance, authority and history [60]. In evaluating the correctness of the ontology, the finding revealed that it satisfied six out of the 10 characteristics identified in Burton-Jones *et al*., although only eight characteristics (lawfulness, richness, interpretability, consistency, clarity, comprehensiveness, accuracy and relevance) were relevant to the evaluation given its recentness. As the ontology is further refined, improvements will be made to address those characteristics that were not satisfied to strengthen the ontology structure and content.

VII.DISCUSSION AND FUTURE WORK

In the first phase, the ontology for Java operators in mutation testing was implemented. There is plenty of Java operators defined nowadays and that value keep increasing due to changes made in Java system. The testing tools invented cannot afford new Java system and some of them only focused on several operators. Just like PIT [43], [44], Mujava [45] [46], these two tools only used selected operators during Java program testing.

The context ontology in this paper focused on Java operators' characteristic and specification. This proposed ontology went into details of each subdomain, explaining the concepts, relationship, and axioms in Java operators. This research contributes to consistency, accuracy, and relationship of context ontology. The ontology has defined every axioms involved, correct hierarchies level, and table to relate one concept with another correctly. This ontology has explained the information of the operators from general to specific characteristic. General information in this perspective means the main characteristics such as

traditional operator, general operator, method level operator and class level operator. These four main characteristics could be found in all Java tools. On the other hand, the specific characteristics include redundant operators, non-redundant operators, deletion operators and equivalency. This ontology has specified and listed all the possible operators within specific characteristic and most important thing is equivalency. Till now, there are still no Java tools that automatically detect equivalency. Hence, with the help of ontology, the development of Java tool for mutation testing would be easy and well improved by adding equivalency.

In short, this proposed ontology is able to help the developers to develop testing tool that specifically focus on Java. Besides, it also helps future researcher defining operators as well as their characteristic. This proposed ontology can be then be used as a basis for some applications in a suite of Java testing tools. That is why this ontology carefully defined all the operators from various testing tools and standardized all the operators' name. So, standardization is important for future use of Java operators.

VIII.CONCLUSION

Thus this paper presents a step-by-step construction of context ontology for Java operators including concepts, axioms, and relations between concepts. The main purpose of this proposed ontology is to standardize the name and the characteristic of operators as well as documenting all the context knowledge for end user. This will help the development of mutation testing tool for Java program in the future. Besides, this documentation will help in many ways for Java tool improvement especially in equivalency detection, redundant and non-redundant detection, and deletion operator detection. There are a lot of improvement that will be made in mutation testing area with regards to this proposed ontology. The changes in Java programming language had forced researcher to implement a new testing tool in order to reach full testing inspection.

This proposed ontology is a new finding in Java operators but not in mutation testing. Some of the researchers have already implemented ontology in mutation testing. But, for Java operators this will ease the tool construction to be more efficient in testing area. So, it is possible for this ontology to become more developed and complex in the future as end users update new knowledge.

ACKNOWLEDGEMENTS

The authors would like to thank Ministry of Higher Education Malaysia for sponsoring the research through the Fundamental Research Grant entitled Search-Based Mobile Applications Test Data Generation Model and Universiti Teknologi Malaysia for providing the facilities and support for the research.

REFERENCES

[1] A. Gomez-Perez, M. Fernandez-Lopez and O. Corcho. (2004). *Ontological Engineering: With Examples from the Areas of Knowledge Management, Ecommerce and the Semantic Web*. Springer.

[2] T. R. Gruber. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition,* 5, 199-220.

[3] T. R. Gruber. (1993). Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In Guarino, N., Poli, R., (Eds.). Formal Ontology in Conceptual Analysis and Knowledge Representation, Deventer. The Netherlands, Kluwer Academic Publishers.

[4] M. Cristani and R. Cuel. (2005). A Survey on Ontology Creation Methodologies. *International Journal on Semantic Web and Information Systems (IJSWIS),* 1(2), 49-69.

[5] M. F. López, A. Gómez-Pérez, J. P. Sierra and A. P. Sierra. (2013). Building a Chemical Ontology Using Methontology and the Ontology Design Environment. *IEEE Intelligent Systems and Their Applications*, 14(1), 37-46. Horrocks, I. (1999). What are Ontologies Good For? In Evolution of Semantic Systems (pp. 175-188). Springer Berlin Heidelberg.

[6] M. Rospocher and L. Serafini. (2012). An Ontological Framework for Decision Support. *Joint International Semantic Technology Conference* (pp. 239-254). Springer, Berlin, Heidelberg.

[7] C. Bartolini. (2016). Mutating OWLs: Semantic Mutation Testing for Ontologies.

[8] R. M. Hierons, M. Harman and S. Danicic. (1999). Using Program Slicing to Assist in the Detection of Equivalent Mutants. Software Testing. *Verication and Reliability,* 9(4): 233-262.

[9] M. Uschold and M. Gruninger. (1996). Ontologies: Principles, Methods and Applications. *The Knowledge Engineering Review,* 11(2), 93-136.

[10] M. Poveda Villalon, M. C. Suárez-Figueroa and A. Gómez-Pérez. (20100. A Double Classification of Common Pitfalls in Ontologies.

[11] R. Jasper and M. Uschold. (1999). A Framework for Understanding and Classifying Ontology Applications. *Proceedings 12th Int. Workshop on Knowledge Acquisition, Modelling, and Management KAW,* 99, 16-21.

[12] S. Lee, X. Bai and Y. Chen. (2008). Automatic Mutation Testing and Simulation on OWL-S Specified Web Services. *Simulation Symposium, 2008. ANSS 2008. 41st Annual* (pp. 149-156). IEEE.

[13] R. Wang and N. Huang. (2008). Requirement Model-based Mutation Testing for Web Service. Next Generation Web Services Practices, 2008. NWESP'08. *4th International Conference on* (pp. 71-76). IEEE.

[14] X. Wang, N. Huan and R. Wang. (2009). Mutation Test based on OWL-S Requirement Model. Web Services, 2009. ICWS 2009. *IEEE International Conference on* (pp. 1006-1007). IEEE.

[15] D. Hamlet. (1992). Are We Testing for True Reliability? *IEEE Software,* 9(4), 21-27.

[16] D. Schuler, V. Dallmeier and A. Zeller. (2009). Ecient Mutation Testing by Checking Invariant Violations. Technique Report, Saarland University, Saarbrucken, Telefon,

[17] M. S. Tuloli, B. Sitohang and B. Hendradjaya. (2016). Regex Based Mutation Testing Operator Implementation. Data and Software Engineering (ICoDSE). *2016 International Conference on* (pp. 1-6). IEEE.

[18] J. Offutt and P. Ammann. (2008*). Introduction to Software Testing* (p. 27). Cambridge: Cambridge University Press.

[19] A. G. Fabio and D. Dipankar. (2004). Anomaly Detection Using Real-Valued Negative Selection. Division of Computer Science, University of Memphis, Memphis, TN.

[20] Y. S. Ma and J. Offutt. (2005). Description of Class Mutation Mutation Operators for Java. Electronics and Telecommunications Research Institute, Korea.

[21] D. Schuler and A. Zeller. Javalanche. (2009). Efficient Mutation Testing for Java. *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 297-298). ACM.

[22] S. W. Kim, J. A. Clark, and J. A. McDermid. (2001). Investigating the Effectiveness of Object-oriented Testing Strategies Using the Mutation Method. *Software Testing, Verification and Reliability,* 11(4), 207-225.

[23] S. Kim, J. Clark, and J. McDermid. (1999). The Rigorous Generation of Java Mutation Operators using HAZOP. *Informe técnico*, The University of York.

[24] S. W. Kim, J. Clark and J. McDermid. (1999). Assessing Test Set Adequacy for Object Oriented Programs Using Class Mutation. *28 JAIIO: Symposium on Software Technology.*

[25] S. Kim, J. A. Clark and J. A. McDermid. (2000). Class Mutation: Mutation Testing for Object-oriented Programs. *Proc. Net. ObjectDays* (pp. 9-12).

[26] S. W. Kim, J. A. Clark and J. A. McDermid. (2001). Investigating the Effectiveness of Object-oriented Testing Strategies Using the Mutation Method. *Software Testing, Verification and Reliability*, 11(4), 207-225.

[27] Y. S Ma, Y. R. Kwon and J. Offutt. (2002). Inter-class Mutation Operators for Java. Software Reliability Engineering, 2002. ISSRE 2003. *Proceedings. 13th International Symposium on* (pp. 352-363). IEEE.

[28] R. T. Alexander, J. M. Bieman, S. Ghosh and B. Ji. (2002). Mutation of Java Objects. Software Reliability Engineering, 2002. ISSRE 2003. *Proceedings 13th International Symposium on* (pp. 341-351). IEEE.

[29] Y. Jia and M. Harman. (2011). An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering,* 37(5), 649-678.

[30] S. Kim, J. A. Clark and J. A. McDermid. (2000). Class Mutation: Mutation Testing for Object-oriented Programs. *Proc. Net. ObjectDays* (pp. 9-12).

[31] Y. S. Ma, Y. R. Kwon and J. Offutt. (2002). Inter-class Mutation Operators for Java. Software Reliability Engineering, 2002. ISSRE 2003. *Proceedings. 13th International Symposium on* (pp. 352-363). IEEE.

[32] N. F. Noy and D. L. McGuinness. (2001). Ontology Development 101: A Guide to Creating Your First Ontology.

[33] M. Umar. (2006). An Evaluation of Mutation Operators for Equivalent Mutants. Project Report, MSc in Advanced Software Engineering, Department of Computer Science, King's College London, London, UK.

[34] Y. S. Ma and J. Offutt. (2005). Description of Class Mutation Mutation Operators for Java. Electronics and Telecommunications Research Institute, Korea.

[35] R. Just, G. M. Kapfhammer and F. Schweiggert. (2012). Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis. *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on* (pp. 11-20). IEEE.

[36] M. E. Delamaro, J. Offutt and P. Ammann. (2014). Designing Deletion Mutation Operators. *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on* (pp. 11-20). IEEE.

[37] L. Madeyski, W. Orzeszyna, R. Torkar and M. Jozala. (2014). Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and A Comparative Experiment Of Second Order Mutation. *IEEE Transactions on Software Engineering,* 40(1), 23-42.

[38] D. Klischies and K. Fögen. (2016). An Analysis of Current Mutation Testing Techniques Applied to Real World Examples. *Full-scale Software Engineering/Current Trends in Release Engineering*, 13.

[39] L. Deng, J. Offutt, P. Ammann and N. Mirzaei. 2017. Mutation Operators for Testing Android Apps. *Information and Software Technology*, 81, 154-168.

[40] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson and M. A. Musen. (2001). Creating Semantic Web Contents with Protege-2000. *IEEE intelligent Systems,* 16(2), 60-71.

[41] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur and Y. Katz. Pellet. (2007). A Practical Owl-dl Reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web,* 5(2), 51-53.

[42] T. Huang, W. Li and C. Yang. (2008). Comparison of Ontology Reasoners: Racer, Pellet, Fact++. In AGU Fall Meeting Abstracts. Dec.

[43] H. Coles, T. Laurent, C. Henard, M. Papadakis and A. Ventresque. (2016). PIT: A Practical Mutation Testing Tool for Java. *Proceedings of the 25th International Symposium on Software Testing and Analysis* (pp. 449-452). ACM.

[44] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. Le Traon and A. Ventresque. (2017). Assessing and Improving the Mutation Testing Practice of PIT. *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on* (pp. 430-435). IEEE.

[45] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis and N. Malevris. (2016). Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Study. *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on* (pp. 147-156). IEEE.

[46] M. Delahaye and L. Du Bousquet. (2013). A Comparison of Mutation Analysis Tools for Java. *Quality Software (QSIC), 2013 13th International Conference on* (pp. 187-195). IEEE.

[47] A. Lozano-Tello and A. Gómez-Pérez. (2004). Ontometric: A Method to Choose the Appropriate Ontology. *Journal Of Database Management,* 2(15), 1-18.

[48] F. J. López-Pellicer, L. M. Vilches-Blázquez, J. Nogueras-Iso, O. Corcho, M. A. Bernabé and A. F. Rodríguez. (2008). Using a Hybrid Approach for the Development of an Ontology in the Hydrographical Domain.

[49] O. Corcho, M. Fernández-López and A. Gómez-Pérez. (2003). Methodologies, Tools and Languages for Building Ontologies. Where is Their Meeting Point? *Data & Knowledge Engineering,* 46(1), 41-64.

[50] O. Corcho, M. Fernandez-Lopez, A. Gómez-Perez and A. Lopez-Cima. (2005). Building Legal Ontologies with METHONTOLOGY and WebODE. *Law and the Semantic Web* (pp. 142-157). Springer Berlin Heidelberg.

[51] T. A. Budd and A. S. Gopal. (1985). Program Testing by Specification Mutation, *Computer Languages.* 10, 63-73.

[52] J. S. Bradbury, J. R. Cordy, and J. Dingel. (2006). Mutation Operators for Concurrent Java (J2SE 5.0). *Mutation Analysis, 2006. Second Workshop on* (pp. 11-11). IEEE.

[53] Y. S. Ma, Y. R. Kwon and J. Offutt. (2002). Inter-class Mutation Operators for Java. Software Reliability

Engineering, 2002. *ISSRE 2003. Proceedings. 13th International Symposium on* (pp. 352-363). IEEE.

[54] R. T. Alexander, J. M. Bieman, S. Ghosh, and J. Bixia. (2002). Mutation of Java Objects. *13th International Symposium on Software Reliability Engineering.* pp. 341-351. Fort Collins. CO USA.

[55] Y. S. Ma and J. Offutt. (2005). Description of Method-level Mutation Operators for Java. Electronics and Telecommunications Research Institute, Korea, Tech. Rep.

[56] Y. S. Ma, J. Offutt and Y. R. Kwon. (2006). MuJava: A Mutation System for Java. *Proceedings of the 28th International Conference on Software Engineering* (pp. 827-830). ACM.

[57] L. Madeyski and N. Radyk. Judy. (2010). A Mutation Testing Tool for Java. *IET Software,* 4(1), 32-42.

[58] R. Just, F. Schweiggert and G. M. Kapfhammer. MAJOR. (2011). An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler. *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on* (pp. 612-615). IEEE.

[59] D. Schuler and A. Zeller. Javalanche. (2009). Efficient Mutation Testing for Java. *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 297-298). ACM.

[60] A. Burton-Jones, V. C. Storey, V. Sugumaran and P. Ahluwalia. (2005). A Semiotic Metrics Suite for Assessing the Quality of Ontologies. *Data & Knowledge Engineering,* 55(1), 84-102.

[61] J. Brank, M. Grobelnik and D. Mladenić. (2005). A Survey of Ontology Evaluation Techniques.