ICIC Express Letters
Volume **14**, Number **5**, May **2020**

 ISSN 1881-803X
pp. **497–504**

SPECIFYING A PROGRAM CONTROL PATH USING FLOW DIAGRAM TRANSLATED INTO DESCRIPTION LOGIC OF ONTOLOGY ENGINEERING

Shimaa Ibrahim[1], Yasser Fouad Hassan[2,3] and Mohamed Hamed Kholief[4]

[1]Department of Mathematics
Faculty of Science
University of Damnhour
El-Gomhoreya Street, Damanhour 22516, Egypt
Shiamaa.ibrahim@yahoo.com

[2]Faculty of Computing and Artificial Intelligence
Pharos University
7 Sidi Gaber, Alexandria 21311, Egypt
y.fouad@pua.edu.eg

[3]Department of Mathematics and Computer Science
Faculty of Science
University of Alexandria
Baghdad Street, Alexandria 21568, Egypt

[4]Computers and Information Technology
Arab Academy for Science, Technology and Maritime Transport
9 Abou Qear Street, Alexandria 21937, Egypt

Received October 2019; accepted January 2020

Abstract. *Software testing plays a vital role in improving the performance of software by detecting and fixing bugs and faults which cause software failure. However, software testing is an expensive task, labor-intensive and time-consuming process in the software development life cycle. Every software product needs to be tested in order to make sure it achieves all of its goals and to detect any unexpected behaviour. One of the most critical features in structural testing is path testing which helps to find every possible executable path that helps to determine all faults lying within a piece of code. Any software program includes multiple entries and exit points. Testing each of these points is challenging as well as time-consuming. To reduce this complexity and time consuming, the use of ontologies might prove useful. Ontology is a technique used at one or more software lifecycle phases. Ontology allows for the definition of a common vocabulary and framework among users (either human or machines). Software development has benefited from this conceptual modelling, allowing a common understanding of the concepts involved in the software process. This paper is proposed to improve test path generation of control flow graph. The basic idea is to use OWL-DL ontology as the knowledge representation formalism, to model and specify control flow by adding semantics to control flow graph entities and adding semantics to the dynamic behaviour of control flow relations.*
Keywords: Software testing, Control flow graph (CFG), Basis path, Cyclomatic complexity (CC), Ontology, OWL-DL

1. **Introduction.** Software testing is the process of evaluating the developed software to ensure that it correctly implements a specific function and is traceable to customer requirements. Software testing is now treated as the most important part in software development life cycle. Black-box and white-box testing are the two major techniques for unit testing. In black-box testing, no information about the internal structure of the

DOI: 10.24507/icicel.14.05.497

program under testing is available. However, in white-box testing, a complete source code or the internal structure is available.

Basis path (or path) testing, a white-box testing technique, is one of the most important ways of testing the software code. Basis path testing uses a control flow graph to depict the logical control flow of the program under test. The main focus in this testing is to write test cases in such a way that it covers all possible feasible paths including all nodes and edges.

Control flow graphs (CFG) model is a program visualization technique commonly used when analyzing the possible flow of control between the basic blocks of a program during its execution. As such, many software analysis tools offer a feature for the automatic generation of control flow graphs for given program code. For large program functions, however, the resulting control flow graphs do become rather huge that sometimes it becomes difficult to examine a certain execution path [1]. Control flow information is used to recognize a set of paths from which test cases are generated. Test case creation is one of the most difficult steps in testing. Therefore, the creation of test cases is one of the important tasks in software testing [2]. In this context, knowledge management (KM) emerges as an important means to manage software testing knowledge and to improve the software testing process. The use of ontologies for testing has not been discussed as much as the use of ontologies in other stages of the software development process [3].

The approach in this paper is to apply semantic technologies and ontology engineering. The main purpose is to create a method for deriving test path from an ontology representing the specification and domain for control flow graph.

This paper is structured as follows: Section 2: a discussion of related works; Section 3: preliminaries of control flow graph and ontology; Section 4: the proposed approach; Section 5: a case study in this formalism; and finally Section 6: the conclusion of the paper.

2. **Related Works.**

The use of ontologies for testing has not been discussed as much as the use of ontologies in other stages of the software development process. Researchers focus on generating the executable test cases based on the ontologies and reusing the already built test cases. Nasser et al. [4] presented a framework to automatically generate the executable test cases based on the ontologies on different domains with custom defined coverage rules. The framework uses various ontologies such as behavioural model ontology, domain ontology, and implementation ontology to generate the custom defined coverage criteria in order to generate the executable test suite.

Dalal et al. [5] also described an ontology-based approach for test case reuse which enhances flexibility and reusability during test case generation and handles users' queries better. Roy et al. [6] proposed a formalization of business processes which merges OWL-DL with semantic web rule language (SWRL). The main goal of his approach is linking control flow relations of a process and its sub-processes using rules as well as providing a consistent way of modelling and designing ontologies. The framework is successfully integrated with different applications involving requirement engineering.

3. **Preliminaries.**

3.1. **Control flow graph.**

A control flow graph (CFG) is a directed graph that visualizes all possible execution paths in a program. Each node corresponds to one of the basic blocks, which the program is composed of. The edges of the control flow graph represent the flow of control between these basic blocks. There are two types of basic blocks that pose exceptions, namely the entry blocks and the exit blocks and most of the programs are constructed with the three types of constructs, namely sequence, selection and iteration (see Figure 1) [7].

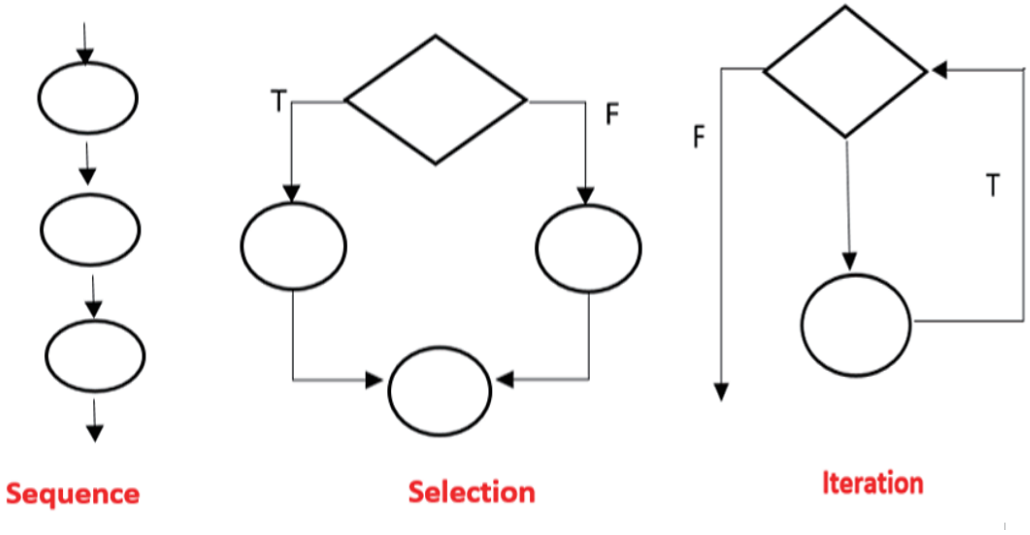

FIGURE 1. Basic types of control structures

Control flow graph (CFG) diagrams are used to generate optimal or efficient path for software under test (SUT). In other words, a control flow graph describes how the control flows throughout the program. CFG based testing provides all statement coverage, branch nodes coverage, event coverage and provides all path coverage. This is the most effective technique for software testing.

3.2. **Cyclomatic complexity.** Cyclomatic complexity is used to generate a number of linearly independent paths in the graph. Cyclomatic complexity is also denoted as V(G) while V means the Cyclomatic number in graph theory. The formula V(G) = P + 1, where P stands for number of predicate nodes in control flow graph.

3.3. **Ontology.** An Ontology is a specification of the conceptualization and corresponding vocabulary used to describe a domain. It represents a domain knowledge in an understandable way for both human and computer. It is formed by a set of concepts which are organized hierarchically and defined by properties [8]. OWL (Ontology Web Language), recommended by the W3C, is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL1 offers three sublanguages with increasing expression intended for specific communities of developers and users: OWL Lite, OWL DL, and OWL Full [8]. OWL DL supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL is so named due to its correspondence with description logics.

Description logic (DL) is the most recent family of formal languages of knowledge representation based on first-order logic. It consists of basic descriptions of the application domain, i.e., atomic concepts for the classes or group of individuals with similar characteristics; atomic roles for properties of these concepts or binary relations between individuals, and from them, other complex descriptions can be constructed as axioms using a set of logical operators called concepts constructors [9]. A knowledge base that refers to an application domain, formalized through description logic, comprises two fundamental components as shown in Figure 2.

TBox, a terminological component, represents the intentional knowledge or the knowledge about the characteristics of the concepts.

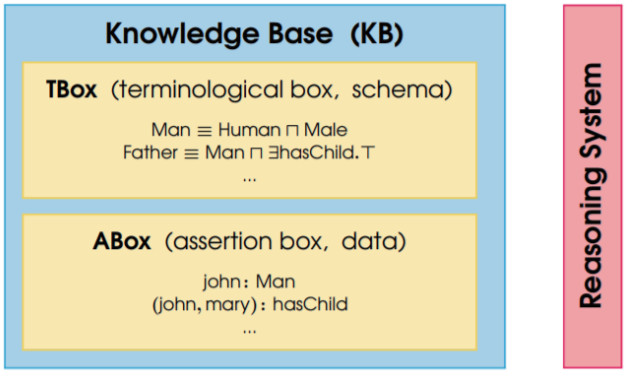

FIGURE 2. DL architecture

ABox, an assertional component, represents the extensional knowledge or the specific knowledge about the individuals (instances) and their relationships within the same abstraction level.

3.4. **Behavioural model ontology.** Behavioural models describe the internal behaviour of a system. Behavioural model ontology is an ontology-based representation of the software artifact based on which tests are generated. It describes concepts corresponding to the software artifacts structural elements, the relationships between them, and their instances. For code-based artifacts, behavioural model ontology can describe the concepts in a programming language (such as methods, and function call) and be automatically populated from the existing code. Also, for GUI testing, it can describe the GUI elements and be automatically populated from the existing GUIs [11]. On the other side, application logic ontology can be used to define logic (reason) behind the application behaviour. It defines the reasons for a particular behaviour performed by the software system. While system behaviour ontology treats the system as 'Black Box', application, logic ontology treats it as 'White Box', i.e., the latter deals with the reason behind a particular behaviour and knows how and why the system is behaving in such a way. Finally, behavioural model ontology can define and model diagrams, such as data flow diagram, flow chart diagram, activity diagram, sequence diagram, class diagram state chart diagram, object diagram, component, and deployment diagram [10].

4. **Proposed Approach.** This paper proposes to improve test path generation of control flow graph. The basic idea is to use OWL-DL ontology, to model and specify control flow by adding semantics to specify the meaning of control flow graph entities and adding semantics to specify the dynamic behaviour of control flow relation. OWL-DL seems to perfect choice for semantically annotating flow diagram and dynamic behaviour of control flow relation.

The approach decomposes control flow structure diagram into atomic patterns (see Figure 3) and generates an independent OWL concept for each of these patterns and then merges concepts to get the complex concept for the whole diagram. Complex concept is generated from the start node to an end node in a diagram. In the following, we will focus on defining concepts and properties we have chosen.

Axiom.

Start node. Start node is defined for the corresponding first node in the diagram. The expression B_0 denotes the complex concept for the whole block. The concept expression for the pattern in (Figure 3(a)) is modeled as a conjunction of Start class and the existential restriction of concept class for block A with followed_by role.

$$B_0 \equiv (\text{Start} \sqcap = 1 \text{ followed\_by}.B_A) \tag{1}$$

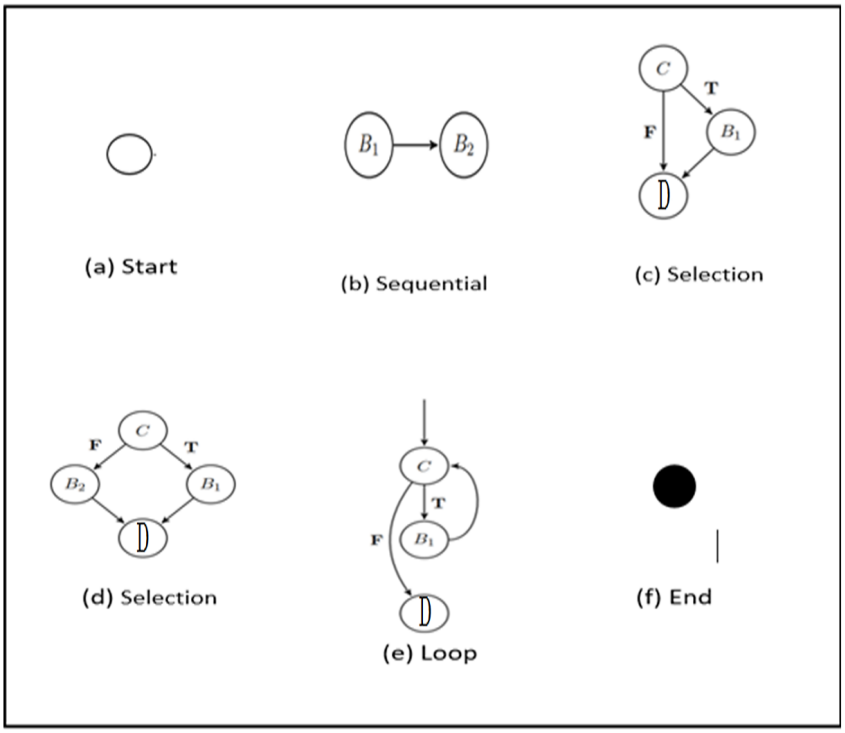

FIGURE 3. Basic pattern of CFG

Sequential pattern. In Figure 3(b) block B_1 is followed by B_2.

$$B_1 \equiv (\text{Block} \sqcap = 1 \text{ followed\_by}.B_2) \tag{2}$$

Selection pattern. This section is divided in two parts.

If statement. Figure 3(c) block C is followed by block B_1 or block D depending on certain condition. The condition associated with the choice is captured as string values for the data type property pairWith.

$$C \equiv (\text{Block} \sqcap = 1 \text{ followed\_by}.(B_1 \sqcup D)) \tag{3}$$

$$B_1 \equiv (\text{Block} \sqcap \exists \text{pairWith } \text{“xsd : T”} \sqcap \ldots) \tag{4}$$

$$D \equiv (\text{Block} \sqcap \exists \text{pairWith } \text{“xsd : F”} \sqcap \ldots) \tag{5}$$

If-else statement. In Figure 3(d) block C is followed by block B_1 or block B_2 depending on certain condition. The conditions associated with the choice are captured as string values for the data type property pairWith.

$$C \equiv (\text{Block} \sqcap = 1 \text{ followed\_by}.(B_1 \sqcup B_2)) \tag{6}$$

$$B_1 \equiv (\text{Block} \sqcap \exists \text{pairWith } \text{“xsd : T”} \sqcap D) \tag{7}$$

$$B_2 \equiv (\text{Block} \sqcap \exists \text{pairWith } \text{“xsd : F”} \sqcap D) \tag{8}$$

Iteration pattern. This construct is a special decision with a self-reference. Loop can be specified by breaking it into patterns and writing the concepts accordingly. Figure 3(e) begins with “C” choice node if condition evaluates to true, node B_1 occurs and then return to choice node again. Otherwise “D” node occurs if the condition is false. Write concepts corresponding to these patterns as formulated before. This may lead to cyclic terminology which need not be definitorial [9]. However, argue that this terminology will have an interpretation which is a fixpoint and hence will have a model. The cycle (due to the loop) of this terminology contains zero negative arc and so, it will have a fix point interpretation [9].

$$C \equiv (\text{Block} \sqcap = 1 \text{ followed\_by}.(B_1 \sqcup D)) \tag{9}$$

$$B_1 \equiv (\text{Block} \sqcap \exists \text{pairWith “xsd : true”} \sqcap = 1 \text{ followed\_by.C}) \quad (10)$$

$$D \equiv (\text{Block} \sqcap \exists \text{pairWith “xsd : false”} \sqcap \ldots) \quad (11)$$

End node. In Figure 3(f) the graph terminates with an end node which is preceded by a node F.

$$B_F \equiv (\text{Block} \sqcap = 1 \text{ followed\_by.End}) \quad (12)$$

Table 1 lists sample axioms corresponding to flow patterns in Figure 3. Here $\perp$ denotes contradiction, similar to owl: “Nothing”.

TABLE 1. Axiomatization for diagram

Statement	***OWL-DL axioms***
Start, End, Block, Loop and Path subclasses of class Node	Start, End, Block, Loop, Path $\sqsubseteq$ Node
precede is the super role of followed_by	followed_by o followed_by $\sqsubseteq$ precede
Start node has no predecessor	(Node $\sqcap\forall$ containedIn. Diagram $\sqcap = 1$ followed_by.Start) $\sqsubseteq \perp$
End node has no follower	(End $\sqcap = 1$ followed_by.Node) $\sqsubseteq \perp$
A Diagram contains some nodes	Diagram $\sqsubseteq \exists$ contains.Node
A Diagram begins with a start node	Diagram $\sqsubseteq= 1$ beginsWith.Start
A Diagram ends on an end node	Diagram $\sqsubseteq=$ endsWith.End

5. **A Case Study.** This section proposes an approach to generate basic paths of the graph.

STEP 1 Generate the CFG from source code.

```
procedure avg

1. int count = 0
2. fread(fileptr,n)
3. while (not EOF) do
4.     if (n<0)
5.         return(error)
       else
6.         numarray[count] = n
7.         count++
       endif
8.     fread(fileptr,n)
   endwhile
9. avg = calcavg(numarray,count)
10. return(avg)
```

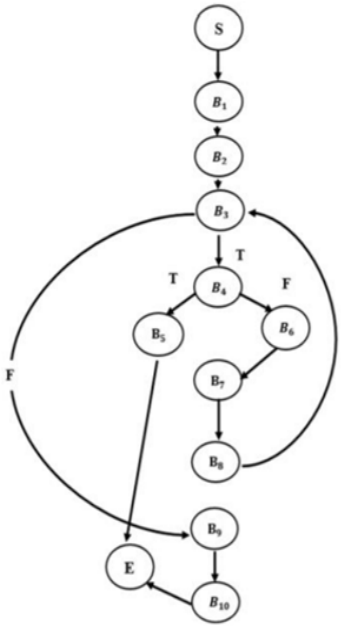

FIGURE 4. Control-flow graph of program

STEP 2 Calculate Cyclomatic complexity in order to get all the possible paths in CFG from start to end node: V(G) = 2 predicate nodes + 1 = 3.

STEP 3 According to TBox axioms and ABox axioms specified in Table 1 translate CFG (see Figure 4) to OWL-DL complex concepts as shown below:

$$B_0 \equiv \text{Start} \sqcap = 1 \text{ followed\_by.} B_1 \tag{13}$$
$$B_1 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.} B_2 \tag{14}$$
$$B_2 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.} B_3 \tag{15}$$
$$B_3 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.} (B_4 \sqcup B_9) \tag{16}$$
$$B_4 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.} (B_5 \sqcup B_6) \tag{17}$$
$$B_6 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.} B_7 \tag{18}$$
$$B_7 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.} B_8 \tag{19}$$
$$B_8 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.} B_3 \tag{20}$$
$$B_9 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.} B_{10} \tag{21}$$
$$B_{10} \equiv \text{Block} \sqcap = 1 \text{ followed\_by.E} \tag{22}$$
$$B_5 \equiv \text{Block} \sqcap = 1 \text{ followed\_by.E} \tag{23}$$
$$\text{DDG} \sqsubseteq = 1 \text{beginsWith.S.} \tag{24}$$
$$\text{DDG} \sqsubseteq = \text{endsWith.E.} \tag{25}$$

STEP 4 Set of paths from above specify:

Path 1: S-B_1-B_2-B_3-B_9-B_{10}-E,
Path 2: S-B_1-B_2-B_3-B_4-B_5-E,
Path 3: S-B_1-B_2-B_3-B_4-B_6-B_7-B_8.

6. **Conclusions.** With the increase in the size of the software, the number of execution paths also increases, thereby degrading the effectiveness of path testing. The vast number of test cases, would lead to a very time consuming and costly testing process, which is impossible to perform. So, this work attempted to show how to integrate semantic technology with software engineering, specifically software testing to produce higher-quality software in a time-effective manner at a lower cost.

One of the helpful goals of using Semantic Web technologies in software engineering is a uniform description of software entities on different levels of abstraction. In light of the foregoing, ontology is used, specifically OWL-DL ontology, to model and specify control flow graph by adding semantics to the meaning of control flow graph entities and to the dynamic behaviour of control flow relations.

Entities and relations of control flow graph are formalized based on OWL-DL to represent it in such a way that it can easily be used to extra test paths in order to map test scenarios on them. OWL-DL seems to be the perfect choice for semantically annotating flow diagram and dynamic behaviour of control flow relations because OWL-DL is the most well known and most investigated species of OWL which can be seen as an alternate notation for the Description Logic language SHOIN (D).

Semantic approach is achieved by following these steps: first, the transformation of the basic control flow patterns to corresponding OWL-DL concepts, and the dynamic behaviour of control flow relations into OWL-DL object properties. Second, the implementation of the ontology using Protégé 5.0 software and using built-in reasons to evaluate the illustrations on it.

REFERENCES

[1] Toprak, Sibel, S. Schupp, A. Wichmann and E. Erklärung, *Sibel Toprak Intraprocedural Control Flow Visualization Based on Regular Expressions*, 2014.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, New York, NY, USA, 2008.
[3] H. J. Happel and S. Seedorf, Applications of ontologies in software engineering, *Proc. of Workshop on Sematic Web Enabled Software Engineering (SWESE) on the ISWC*, pp.5-9, 2006.
[4] V. H. Nasser, W. Du and D. MacIsaac, An ontology-based software test generation framework, *InSEKE*, pp.192-197, 2010.
[5] S. Dalal, S. Kumar and N. Baliyan, An ontology-based approach for test case reuse, *Intelligent Computing, Communication and Devices*, pp.361-366, 2015.
[6] S. Roy, G. S. Dayan and V. D. Holla, Modeling industrial business processes for querying and retrieving using OWL+SWRL, *OTM Conferences*, pp.516-536, 2018.
[7] R. Gold, Control flow graph and code coverage, *Int. J. Appl. Math. Computer Sci.*, vol.20, no.4, pp.739-749, 2010.
[8] D. L. McGuinness and F. van Harmelen, *OWL Web Ontology Language Overview*, W3C Recommendation, http://www.w3.org/TR/2004/REC-owlfeatures20040210/, 2004.
[9] F. Baader and Nutt, Basic description logics, *Description Logic Handbook*, pp.43-95, 2003.
[10] M. P. S. Bhatia, A. Kumar and R. Beniwal, Ontologies for software engineering: Past, present and future, *Indian Journal of Science and Technology*, vol.9, no.9, 2014.
[11] H. Li, F. Chen, H. Yang, H. Guo, W. C. Chu and Y. Yang, An ontology-based approach for GUI testing, *The 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, pp.632-633, 2009.
[12] I. Horrocks, *Description Logics in Ontology Applications*, Springer-Verlag Berlin Heidelberg, 2005.
[13] H. N. Anjalika, M. T. Y. Salgado and P. I. Siriwardhana, *An Ontology-Based Test Case Generation Framework*, 2017.
[14] E. F. Souza, R. A. Falbo and N. L. Vijaykumar, ROoST: Reference ontology on software testing, *Appl. Ontol. J.*, vol.12, no.1, pp.1-30, 2017.
[15] S. Vasanthapriyan, J. Tian and J. Xiang, An ontology-based knowledge framework for software testing, in *Communications in Computer and Information Science*, J. Chen, T. Theeramunkong, T. Supnithi and X. Tang (eds.), Singapore, Springer, 2017.