International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. 14, No.1 (Jun-24)

http://dx.doi.org/10.12785/ijcds/1501126

Classifying Critical Software Components Using Multi-Level Formalization and Knowledge Graphs

D.Jeya Mala[1] and A.Pradeep Reynold[2]

[1]*School of Computer Science and Engineering, Vellore Institute of Technology, Chennai, India*
[2]*Department of Safety, ASET College of Science and Technology, Chennai, India*

Received 3 May 2023, Revised 21 Jun. 2024, Accepted 22 Jun. 2024, Published 26 Jun. 2024

Abstract: When developing a high-quality software solution for industrial strength real-time systems, the critical software components are indispensable. For such kinds of systems, compromising the testing time due to market competition, delivery due date and critical time lines, will lead to hazardous impact to the cost and life of the end users. In particular, this overlooking of such crucial components will usually happen due to non-identification of them during the analysis and design phases of the software. These critical components are the ones that have high level of functionality and dependability when compared to other components. If these components are left untested during testing, the recurrent effects will lead to catastrophic failure. Therefore, it is necessary to devise a mechanism for discovering such critical components using information gathered early in the early phases of the development process. In this research work, the application of artificial intelligence techniques and mathematical formal specifications during analysis and design phases of the software development process are recommended. The identification of such critical components at the early phases of software development helps in devising rigorous testing process to evaluate such components to avoid field failure. By applying knowledge graphs that depict the software's design embedded with design metrics calculated using Object Constraint Language (OCL) based formal specification in order to classify the criticality level of the components is the major contribution of this proposed work. To achieve this, a rigorous methodology of multi-level formalization to generate a more precise system specification along with graph embedding using dependability and complexity metrics associated with each node in the knowledge graph is applied. Finally, a rigorous result analysis is conducted to ensure that, the proposed work provides promising results for industrial strength applications.

Keywords: Knowledge Graphs, Critical Components, Formal Specification, Object Constraint Language (OCL), Unified Modeling Language (UML), Metrics, Complexity, Criticality Index

1. Introduction

Generally, any system comprises of a number of components. And so, a component is one of the system's constituent parts. It may be a component of single functionality and may consist of other components too [1]. Critical software components are those components of a software system that may be responsible for major functionalities, errors, costs, and rework [2]. They are essential for the software system's global operation. Consequently, these components must be examined, created, and tested more carefully and completely than other components.
The objectives of this proposed work are threefold:

- To reduce the cost of failure due to non-identification of critical software components by means of applying formal specifications during the early design phase of SDLC
- To apply knowledge graphs to represent the real-time software system with metric values to identify the critical components
- To provide a novel mathematical and OCL specification based representation for metrics calculation to classify the components before actual development.

A. Motivation

Particularly, the real time systems that belong to healthcare, military and financial domains are highly complex in nature and have their impact on cost and life of the people who use them or being a part of them. Even a small flaw in detected these systems during field operation, will lead to hazardous impact to life and cost associated with them. As these systems have more number of critical components when compared to other software, if any of these critical components are left undetected, inadvertent effects will happen which will lead to erroneous operations, system failure, high cost and resource wastage etc.
In reality, these components include the majority of flaws uncovered during prerelease testing, and an even smaller number of components contain the majority of flaws discovered during operation [3].
This emphasizes the necessity for formal approaches to identify essential software components early in the software development process. According to Peter Bishop et al. [4],

E-mail address: jeyamala.d@vit.ac.in, pradeepreynold.a@gmail.com

the methodologies used to identify key software components are design documentation, expert opinions, and source code analysis.

When software is developed from specifications in a mathematically rigorous manner, development labor can be repurposed for program testing [5]. Therefore, in order to minimize design flaws and implementation failures, it is necessary to verify the important software components using a formal specification language such as OCL.

Cortelessa et al. [6] have presented a risk analysis technique for identifying essential components early in the software development process. The methodology uses design documents developed using UML to estimate the chance of failure due to performance issues and combines it with an estimate of severity of the failure derived from the Functional Failure Analysis (FFA). Moreover, essential software components would require rigorous analysis, design, and implementation, as well as additional testing efforts.

In a similar manner, the important software components are defined as those that have a greater impact on the overall quality of the product and hence require precise analysis, design, implementation, and testing. Ignorance of problematic or fault-prone components during the analysis and design phase results in costly, time-consuming, and substandard development and testing efforts. Diagnosis, for instance, is the key component of Patient Monitoring System. Incorrect analysis of this component results in erroneous diagnosis and a subpar medical report. Note that the criticality of the components estimated in this paper is not appropriate for hard real-time systems.

In this research work, a novel critical components identification framework is proposed. In which, the UML Meta model namely the class model is first defined using the OCL specification. It is followed by the creation of knowledge graph with each node represents the objects (subsystems, components, and classes etc.) in the Meta model and each node is being assigned with a metric value namely criticality index. This criticality index metric value is calculated using selected design complexity criteria with severity analysis on each component. This helps to identify the critical software components from the software to be developed.

B. Background

1) *Application of UML Class Model*

This proposed work focuses on the development of formal representation of the components to identify the criticality measure of each of them. In order to generate the OCL based formal representation, we need a static model and not the dynamic model that represents the behavioral aspects of the system [7]. As the behavioral models such as sequence, collaboration and activity diagrams provide the implementation guidelines for the developers to proceed in their software development and not the internal structure of the components and their relationships [8].

The objective here is to identify the criticality measure of each component based on the core functionality and dependability measures. To get this value, it is essential to depict the internal structure of each component and its relationship with other components. Hence, this research work uses UML class model which is the Meta model of the system to be developed.

The system's static structure can be represented via the most prevalent UML class model, which depicts classes, interfaces, collaborations, and relationships in a component-based system. However, a software system developed solely on the basis of class diagrams will be flawed. In addition, the informal definitions and unclear specifications of the majority of the available metrics [9] result in varying interpretations and a consequent lack of adoption.

To instill confidence in the final result, it is necessary to examine and certify all important software components using standard techniques and procedures. Modeling and describing design constraints are required to enhance the logic of software architectures and hence their quality [10]. OCL permits the explicit description of extra constraints on the objects and entities of a UML model. It is founded on mathematical set theory and predicate logic, as well as other concepts [11].

2) *Application of OO Metrics*

Complexity of software design is directly proportional to failure rate [12] In addition; object-oriented design metrics play a crucial role as early predictors of problematic classes and components in an object-oriented system [13]. Thus, software complexity measures can be used to determine which software components are crucial, since the component with the highest complexity value is likely to be vital. Therefore, the complexity metric values of the components are the essential data required to assess the component's criticality.

Our proposed methodology would correlate the overall metric value of a component, which is derived from the selected design complexity and dependability metric values, to the component's criticality index. In addition, software engineers can make software development decisions and priorities the testing process, defect identification, debugging, and testing efforts if they are aware of the criticality of components early on.

3) *Application of Formal Specification*

Habib [14] has identified that, the application of formal methods provide improved efficiency especially in the field of avionics during the design and verification processes. The formal methods help in providing the formal languages, compilers and adapted methods to establish the non-functional formal verification techniques.

4) *Application of Knowledge Graphs (KG)*

In recent years, the knowledge graphs gained their attention in representing real world problems as they can be embedded with semantics which are used for further analysis. Especially in Machine Learning, these constructs are playing a crucial role.

Knowledge Graphs (KG) are graphs which represent a

network of real-world entities. In the knowledge graph, entity descriptions are linked together. A knowledge graph is a knowledge base that uses graph structure and topology to display and integrate data. Compared to other forms of databases, this enables more effective querying and information retrieval. Additionally, by spotting connections and patterns in the data that would not be immediately obvious from individual data points, knowledge graphs can be utilized to develop fresh insights. In a variety of situations, this can enhance decision-making and spur creativity. [15], [16] In this research work, a novel methodology that combines Knowledge Graphs embedded with metric values of each object or component in the system. These metric values are calculated using formal specifications represented using OCL (Object Constraint Language) in order to identify whether a component or object is critical or not.

Further, the paper is divided into sections listed below: Section 2 gives the literature survey with a summary of the major contributions of the proposed work, Section 3 provides a summary of the proposed work with our recommended methodology. The fourth section includes a case study on E-Commerce application. The conclusion is presented in Section 5.

2. Related Work and Contributions of the Proposed Work

A. Literature Review

In this paper, a methodology that uses design documents, formal specifications, and design metrics to determine the critical software components is proposed. In this section, some of the research works relevant to the proposed work are summarized.

A number of authors [6], [17], [18] have developed a reliability-based risk assessment technique for determining which parts of software are indispensable. The researchers employed design complexity measurements to pinpoint potential points of failure.

To collect data on UML models, Goseva-Popstojanova et al. [17] used the commercial modeling software Rational Rose Real Time (RoseRT). In their method, a dynamic heuristic risk factor is obtained for each component and connector in software architecture, and the severity is evaluated using a hazard analysis. Next, a Markov model is built to get the potential dangers in different circumstances. The scenarios' risk factors are used to estimate the use case's risk factor and the system's risk factor as a whole.

Mitrabinda Ray and Durga Prasad Mohapatra [18] have proposed a reliability- based risk assessment method at the design level using the sequence and state chart diagram of UML. They have identified high risk components after taking into account the risks involved with the different stages of a component, the importance of the messages involved, and the risks to the business as a whole.

Some of the classification techniques that have been evaluated by Ebert Christof [2] for use in determining which parts of software are most important for fault prediction. Based on their findings, the fuzzy classification method is the most effective for identifying crucial components. And they concluded that the top 20% of the most important software parts can be easily identified using Pareto analysis (the "80:20 rule").

To help pinpoint the most important pieces of code in a modular application, Suri. P K., and Kumar Sandeep [19] developed a simulator (CBS). Component Execution Graph (CEG) was used to portray the CBS as a network in their proposed study. Each execution link in this graph has a weight that represents the component's final destination. The total 'w[i]' of all execution linkages along a path is its weight, 'W'. Each heavily weighted execution path is presumed to be the Crucial Path with all components in that path being assumed to be critical software components.

Using a statistical and machine learning technique, Malhotra Ruchika and Jain Ankita [20] tested a methodology to evaluate the components' criticality level, CK metrics and QMOOD metrics are applied. In this study, they put 19 object-oriented metrics for class error prediction to the test. Out of the 19 metrics they examined, they found that only a subset of the metrics as reliable indicators of failure propensity.

Janes, A., et al. [21], have used a real-time telecommunication software system as an example, to demonstrate that early lifecycle data can be utilized to identify the most error prone class components. To evaluate the relative merits of the CK measure and the Lines of Code (LOC) metric, they have used statistical models. Their research demonstrated that inter-class communication-based measures, such as RFC and CBO, are more accurate indicators of critical software components than the LOC metric. In addition, their research revealed that the zero-inflated negative binomial regression model is found to be better than other applicable statistical models.

A number of software measures, including CBO, CTA, CTM, RFC, WMC, DIT, NOC, NOAO, NOOM, NOA, and NOO, were shown to be useful in locating the critical components by Shatnawi et al. [22].

Metrics used in Object-Oriented design have been analysed by Zhou et al. [23] to determine their ability to foretell bugs of varying degrees of severity. The NASA data set used to generate their findings is freely available to the public. The authors' results indicated that, CBO, WMC, RFC, and LCOM metrics used to predict the critical level of the components and these metrics are able to find the severity level too. They also claimed that the design metrics are more accurate predictors of class defects with low severity than of those with high severity.

Quality classification models for defect prediction in traditional and highly iterative, or agile, software development processes for both initial delivery and for subsequent, sequential releases are made possible by the CK and QMOOD OO class metrics suites, as shown by Olague et al. [24]. Class quality can be reliably predicted using CK-WMC, CK-RFC, QMOOD-CIS, and QMOOD-NOM (error-proneness). Studies have demonstrated that CK mea-

surements are more accurate and dependable predictors of fault-proneness than MOOD or QMOOD metrics.

Object-oriented design metrics have been validated on a commercial Java system by Emam et al. [25] to find bugs in the code early on. Their research confirms that indicators like OCAEC, OCMEC, OCMIC, and DIT all play an important role in identifying potentially faulty groups. They also demonstrated that the DIT and EC measures were significantly linked to the forecasting of fault prone classes. Fault prediction in open-source software like the web browser and e-mail client Mozilla was the subject of an empirical investigation by Gyimo' thy et al. [26]. They have looked at the CK metrics suite and LOC to inform their strategy. According to the results of their research, the CBO metric is the most effective in determining which classes are likely to have problems. They use logical and linear regression methods as well as machine learning techniques like decision trees and neural networks.

Khoshgoftaar et al. [27] did a case study analysis in order to classify error-prone software components. In their study, regression trees and a set of specified rules were used to segregate crucial error prone components.

According to Basili et al. [28], CK metrics seem to be helpful in predicting class fault-proneness early in the development process. The metrics such as WMC, DIT, RFC, NOC, LCOM, and CBO were found to be the suitable measures to find the error-prone components.

By building a Z language based meta-model, Meryem Lamrani et al. [29] proposed a method for defining software design quality measurements like the CK metric suite.

The formalisation effort of object-oriented design metric definitions is described by Baroni et al. [30], and some examples are presented, all of which are performed using UML meta-model with OCL. Using OCL, they constructed FLAME, an informal library meant to facilitate the extraction of metrics (OCL). Definitions of object-oriented design metrics are codified in their method using OCL. In order to verify the correctness of the UML and OCL models, Gogolla et al. [31] used the USE tool.

Zhou et.al [32] have applied Knowledge Graphs to simulate the warehouses in a supply chain so that complex information of warehouse resources have been easily summarized and provided as semantics in the graph. They have derived a unified resource allocation method so that, the decision making can be done in an optimized manner. Their model has provided an integration of semantic information in the workshops which are represented as nodes in the knowledge graph representation. Then they have applied a machine learning algorithm to find the resource information in order to update the graph dynamically.

Ramzy et.al [33] have proposed semantic integration of customer requests using Knowledge Graphs. They matched the actual transition time and the planned transition time using them as the mechanism.

Deng et. al [34], have applied Knowledge Graphs in supply chain management. They have proposed an event logic based knowledge graph to represent the events to maintain high accuracy.

The literature study shows that, there has been considerable work done on identifying critical software components and there is no work provided for critical component's identification though formal specification. And the current research works are focusing on applying Knowledge Graphs for complex problems that involve multiple parameters in decision making.

B. Major Contributions of the Proposed Research

Since the availability of architectural based models, formal specifications and design metrics, the criticality of the software components can be quantified. A precise program design requires the semantics of the software model. Since the well-formedness rules that are not expressible in the UML meta-model, class diagrams are expressed using OCL. Abreu et al. [35] study shows that formal definition is used to avoid subjectivity of measurement and thus allows replicability. This motivates the introduction of new concept which addresses the need for formal specification in failure prone components identification.

In this paper, we show that the OCL specification and dependability and complexity metrics are enough to estimate the criticality of the component. It is not our intent to detect design flaws, reducing component's complexity using redesign, perform risk analysis or define the consequences of critical software components failure in this paper. Instead, our proposed work ranks the components based on their severity which is estimated through formal specification and design complexity metrics. This early knowledge can aid the developers and testers in distributing efforts needed to produce quality software product.

The key contribution of the methodology presented here to the previous literature is to consider both the system's object model and its design metrics when determining which software components are mission-critical. In some cases, such as with safety-sensitive or complicated soft real-time systems, a purely criticality-based approach to identifying critical software components may not be adequate. Therefore, the potentially critical parts must be tested using either new testing tools or formal specifications and severity analysis.

To the best of our knowledge, critical software components identification based on OCL design documents and design metrics are not been found in the existing research works. The existing works have predicted failure prone components at the early stage software development based on risk assessment, simulation based techniques and design metrics etc., without addressing the elements for the cross verification of the identified critical software components using formal specification.

The work flow and the interaction of users with the entire system are given in Figure 1 using use case diagram.

The criticality analysis of the research is given in the Figure 2.

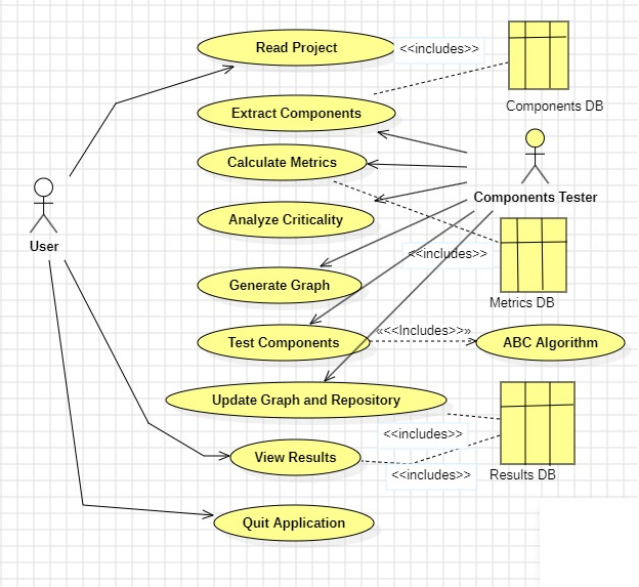


Figure 1. Use case diagram of the proposed work

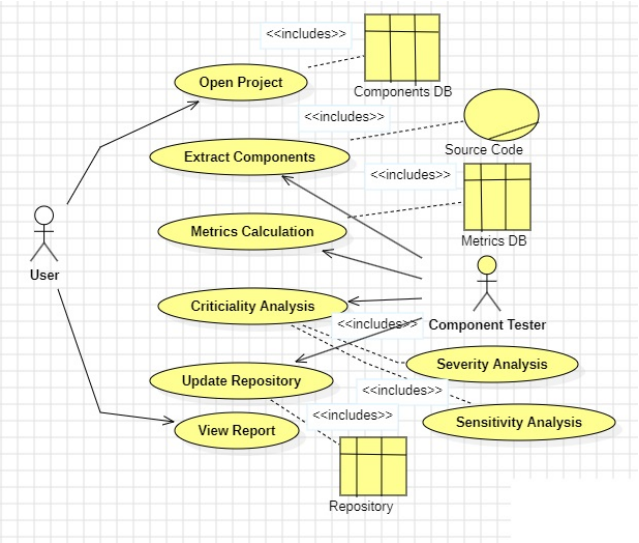


Figure 2. Use Case Diagram – Criticality Analysis

3. Proposed Methodology for the Critical Software Components Identification

In the proposed methodology given in Figure 3, the criticality estimates for UML class diagram-modeled software components are proposed. Using OCL specification, the UML class model is formalized and an analysis of the resulting object model is conducted. Following formalization, criticality estimation and severity analysis are performed on the object model, to locate the most likely sources of failure. It is possible to gauge the component's seriousness by looking at its normalized complexity.
This paper demonstrates that estimating the component's

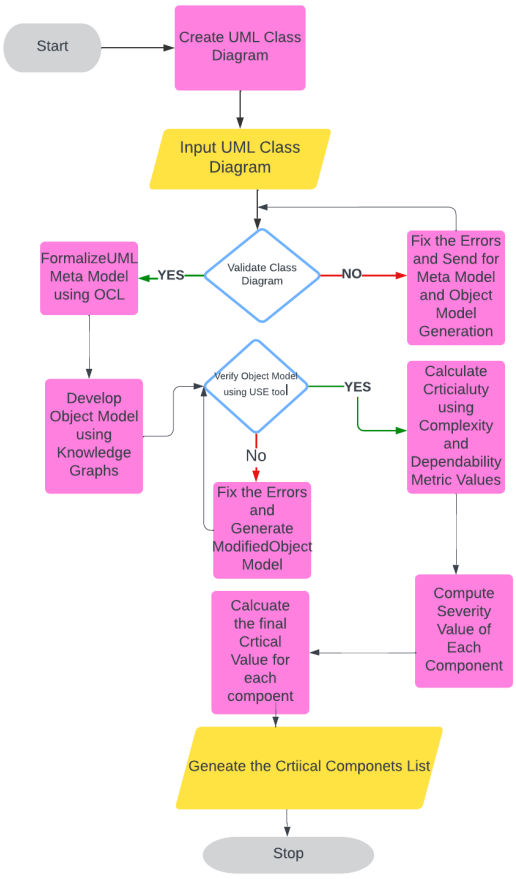


Figure 3. Methodology for critical software components identification

criticality is possible using only the OCL specification and complexity measurements. In Figure 3, the stages of the proposed methodology and the process for locating the critical software components are given.

A. *OCL Transformation*

Defining the software model's semantics is crucial to achieving a clear design for the software. An OCL expression is typically used to describe it. Class diagrams benefit from OCL expressions, a specification language used to specify formal expressions in the diagrams drawn using UML [36]]. This language helps to improve the diagrams' semantic qualities, precision, documentation, and readability. Because of its declarative and typed nature, OCL does not make use of variables or state. Additionally, expression validity can be checked when modelling. It permits the expression of invariants, preconditions, and post-conditions, three types of constraints on components of object-oriented models.
Using OCL syntax, the UML meta model can be transformed into an object model. The UML object diagrams' OCL notation is used to label objects and their relationships, which is why this transformation was developed. To probe

internal organization, object diagrams are drawn. A UML object diagram may be the best choice when attempting to describe intricate inter-class interactions. Whenever there are operations and Invariants in the system under study that cannot be represented graphically using the class diagram, an OCL specification is utilized to represent them. In a class, each operation has its own set of requirements, such as initial and final states. Object-oriented modelling tools have the following parts.

i. **Class hierarchy** - It represents the relationship between the classes, methods invocation between classes, properties or attributes of a class being shared with other classes given in terms of association, inheritance, aggregation and composition.

ii. **Class** -The basic component of UML meta-model. It provides a common template that can be used to create a collection of similar objects. The definition of a class in UML diagram includes the specification of its attributes as the properties and responsibilities as operations.

iii. **Attributes** – It represents the property of a class and has a variable name and a data type.

iv. **Operations** - Class definition includes the specification for operations. They represent the functionalities of the objects. OCL specification is used to represent operations and invariants related to the system under study. Invariants indicate the structure of a system. Each operation in a class is associated with a specification such as pre and post conditions. Pre and post conditions denote the system behavior. An operation can have parameters or non-parameters and can have return types or no return type.

v. **Associations** - The structural relationship between two classes can be described using association. An association indicates one class is related to another class by means of either 1 to 1 or 1 to many or many to many kind of relationship.

vi. **Role Names** - When a class is associated with more than two classes it plays different roles in each class. Thus there is a need to assign a role to each class in an association. They are used in navigation expression of OCL. All role names must be distinct.

vii. **Generalization** - Generalization relationship is applied when one class property is based upon another class property. It indicates inheritance relationship between two or more classes.

B. Proposed Algorithm

The proposed algorithm to achieve the goals of the proposed work is given below:

Step 1: Formalize UML Meta model using OCL specification, build an object diagram as a Knowledge Graph (KG)

Object-oriented analysis and design relies on the UML class diagrams. It serves as an example of the system's structural view. To illustrate the system's classes, their properties, operations, and relationships (such as inheritance, aggregation, and association), a UML class diagram is drawn. The static nature of the system is also emphasized. However, the complexity of integrated OCL expressions should be considered when evaluating UML models because it has a direct impact on the complexity required to manage the UML models, for example, for verification, validation, and code generation [37].
Following is a representation of the backbone of the UML meta-model, as extracted and reconstructed as a knowledge graph with each node holds the criticality metric value associated with that node. It is given in Figure 4. **Step**

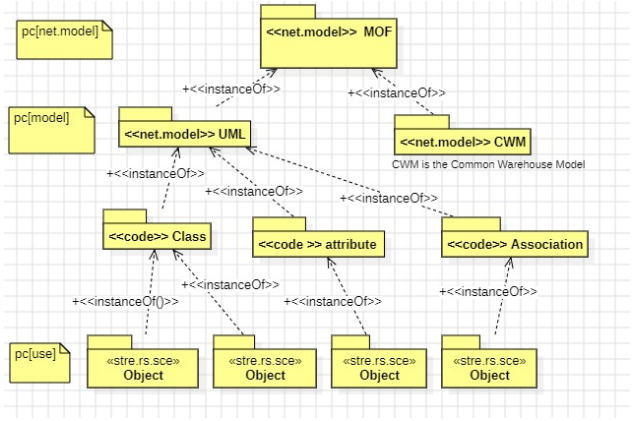


Figure 4. UML Meta Model

2: Select a set of complexity metrics M = m1, m2... which contribute to identify critical components in a system

When compared to hardware failure, it is much more difficult to predict a software component failure. Components prone to failure must be quantified in order to determine their severity. The complexity of software is a reflection of how challenging it is to learn and use [37], [38]. Moreover, the more complicated a module is, the more likely it is to have bugs [39]. As a result, design metrics are essential in creating a cost-effective software product, as the design phase is a significant cost driver in the software development life cycle.
The proposed methodology relies on four sets of object-oriented design complexity metrics. The complexity and reliability of a programme are both affected by these metrics, which are also used to determine the criticality of individual components during the design phase. Chidamber and Kemerer (CK) [40] proposed three of these measures (DIT, NOC, WMC), Bansiya et al [41], [42] proposed two (DCC, and DAM), and Fan-In and Fan-Out metrics are used to find the dependability between classes. The information flow metric is calculated as (Fan-In * Fan-Out)2 [43].

Zimmermann et al. [44] mentioned NOCL, NOI, NOF, NOP, as complexity measures that are associated with field failures and dependencies.
The NSMF metric is suitable for Java-based applications. There is no foolproof way to determine the coupling metric using class diagrams alone. Whenever a method in one class calls a method in another class, it means there is a coupling between them [44].
In the subsequent discussions, the metrics chosen for the proposed study are defined together with their appropriate OCL syntax along with mathematical notation that will be utilized to extract complexity and dependability measurements from the object diagram.

- **NOCL metric (Number of Classes) Definition 1:**
  It indicates the number of classes in the given software system which includes the current class and sub classes.

  Goal: NOC metric measures the complexity of the component. Higher the number of classes defined; larger is the complexity of the class.

  Let a system contains classes C=Ci……Cn and each class contains subclasses S=Sj….Sn then the number of classes is given as NOCL. This NOCL finds the total number of sub classes along with its contained class. So, it will have the total number of sub classes + 1. Thus, NOCL is given by,

Number of Classes (NOCL)	
Mathematical Notation	**OCL Formal Specification**
NOCL= $\sum_{j=0}^{n}$ Sj(Ci) +1	class classname1 end class derivedclass < baseclass end

- **NOI metric (Number of Interfaces) Definition 2:**
  It provides the number of interfaces.

  Goal: NOI metric is a complexity measure. If the number of interfaces defined are more, the complexity of the class will also be higher. Let a class Ci contains interfaces F=Fj…Fn then number of interfaces in a class is given by,

Number of Interfaces (NOI)	
Mathematical Notation	**OCL Syntax**
NOI= $\sum_{j=0}^{n}$ Fj(Ci)	context interfacename self.isAbstract = true and self.allInstances-> isFinal(attribute)

- **NOF metric (Number of Fields) Definition 3:**
  This metric count all attributes defined in a class.

  Goal: NOF metric is a complexity metric. If the number of attributes defined in a class is higher, the criticality of the class will also be higher.
  Let a system contains classes C=Ci……Cn and each class contains attributes A=A1…An, then the number of attributes in a class is given by,

Number of Fields (NOF)	
Mathematical Notation	**OCL Syntax**
NOF = $\sum_{j=0}^{n}$ Aj(Ci)	context classname:functioname(para… type) pre: attribute.isDefined()

- **WMC metric (Weighted method per class) Definition 4:**
  It gives the number of methods defined in a class.

  Goal: This is used to measure the complexity of a class, since the larger the number of methods defined, the higher is the criticality of it.
  Let a system contains classes C=Ci…Cn and each class contains methods M=M1…..Mn, then number of method per class is given by,

Weighted Method per Class (WMC)	
Mathematical Notation	**OCL Syntax**
NOM= $\sum_{j=0}^{n}$ Mj(Ci)	Context Typename :: operationName (param1 : Type1, ...) : ReturnType pre : param1 > ... post: result = ...

- **NOP metric(Number of Parameters) Definition 5:**
  This metric count the total number of parameters defined in each methods of a given class.

  Goal: It is the measure of class complexity. If the number of parameters for a given class is high, it in turn maximizes the complexity of the class.
  Let a system contains classes C=Ci…,Cn and each class contains methods M=Mj. .. Mn, and each method contains attributes A=Ak…An then the number of parameters in a class is given by,

- **DOI metric (Depth of Inheritance) Definition 6:**
  This metric is used to determine the extent to which a certain class is in the system of inherited properties. Accordingly, the class name to which we navigate

Number of Parameters (NOP)	
Mathematical Notation	**OCL Syntax**
$NOP = \sum_{k=0}^{n} Ak(Mj(Ci))$	context Typename::operationName(param 1 : Type1, ...): ReturnType

allows us to derive a tree to represent navigation. The name of the class itself represents the node's starting point.
Each expression always begins with the self variable, and then each path results in a new branch in which the classes visited are individual nodes. Roles which are used for navigation, link nodes together.
The number of predecessors of node i is counted to determine the depth of the inheritance tree if there exists a path from node i to node j and node j is the successor of node i.
Goal: DOI metric is the measure of dependability and complexity of the class. If the depth of inheritance tree is higher, the class dependability will also be higher.

Depth of Inheritance (DOI)	
Mathematical Notation	**OCL Syntax**
$DOI = \sum_{k=0}^{n} PREDES(Ci) + 1$	class classname1 end class derivedclass < baseclass end

- **NOC metric (Number of Dependent classes as Children) Definition 7:**
Using this metric, we can determine how many other classes a target class has as children.

The OCL technique of association specification can be used to infer this. The number of classes that come after Ci can be used as a rough estimate.

Goal: It is a Measure of Class Complexity and Depdenability. The complexity and dependability of a class increases as its dependent children classes grow larger. Let us consider a class Ci has 'n' immediate successor classes and is represented as ImmSucc=j....n.

- **DAM metric (Data Access Metric) Definition 8:**
The ratio of private (protected) attributes to all attributes specified in the class constitutes this measure.

Goal: DAM metric is the complexity metric. Higher the call by the defined function to other function in a system more is the dependability of the given class.

If this metric value is lesser, it indicates that the amount of dependability will be lesser. In that case, the class is more of a self resilient class.
Let a system contains classes C=Ci.....Cn and each class contains attributes A=Aj...An which includes private attributes Ap=P1....Pn, then the ratio of private (protected) attribute to the total number of attribute is given by,

Number of Children (NOC)	
Mathematical Notation	**OCL Syntax**
$NOC = \sum_{j=0}^{n} ImmSucc(Ci) + 1$	class classname1 end class classname2 < classname1 end

Data Access Metric (DAM)	
Mathematical Notation	**OCL Syntax**
$DAM = \frac{\sum_{p=0}^{n} Ap(Ci)}{\sum_{j=0}^{n} Aj(Ci)}$	context classname invinv1:self.allInstances-> isPrivate(attributename)

- **NSMF metric (Number of Static Methods and Fields) Definition 9:**

It provides the number of static methods and fields in a class.

Goal: NSMF metric is the dependability metric. Higher the call by the defined function to other function in a system more is the dependability of the given class.
Let us consider a class that contains static methods SM= SM1...SMn and static fields SF= SF1....SFn, then total number static methods and fields in a class is given by,

Number Static methods and Fields (NSMF)	
Mathematical Notation	**OCL Syntax**
NSMF=NSM+NSF where $NSM= \sum_{j=0}^{n} SMj(Ci)$ $NSF=\sum_{k=0}^{n} SFkj(Ci)$	context stock::sid() pre: self.sid > 0 post: self.sid = self.sid@pre + sid

- **DCC metric (Direct Class Coupling) Definition 10:**

It is a measure of how many other classes are referred to by a given class.
In addition, it's a tally of the classes to which the class belongs but which are not related by inheritance.

The class itself, an instance of the class, or a variable associated with the class are all valid points of reference for a class.
The OCL technique of association specification can be used to infer this.

Goal: DCC metric is both dependency and complexity metrics.

Excessive coupling increases sensitivity to changes in other parts of the design and makes a module more critical.
Let us consider a class Ci associated with other classes Cj=Cj…Cn then class coupling is given by,

Direct Class Coupling (DCC)	
Mathematical Notation	**OCL Syntax**
$DCC = \sum_{j=0}^{n} Cj(Ci)$	Context classname Inv:self.selfclassattribute-condition self.Asscoiatedclassname -condition Context classname Inv: self.associatedclassattribute. rolename condition

- **Fan-In metric Definition 11:**
It is used to measure the number of classes referring the given class.

Goal: Fan-In metric is dependability metric. Higher the call to function in class by other functions, more is the dependability and complexity of the given class.

Let us consider 'a' is the number of class components that calls A, 'b' is the number of parameters passed to A from component higher in the hierarchy, 'c' is the number of parameters passed to A from component lower in the hierarchy, and 'd' is the number of elements read by component A. Then the value of Fan-In of component 'A' is a+b+c+d. If number of classes R=Rj…Rn references a Class Ci then summation of all these measures have to be calculated to provide the Fan-In metric value of Ci.

- **Fan-Out metric Definition 12:**
It is used to find the number of classes referenced by a class.

Fan-In	
Mathematical Notation	**OCL Syntax**
Fan-In= $\sum_{j=0}^{n}$ Rj(Ci)	Context classname Inv:self.selfclassattribute-condition self.Asscoiatedclassname

Goal: Fan out metric is the dependability metric. Higher the call by the defined function to other function in a system, more is the dependability of the given class. Let us consider 'e' is the number of components called by 'A', 'f' is the number of arguments passed from 'A' to component higher in the hierarchy, 'g' is the number of arguments passed from 'A' to component lower in the hierarchy and 'h' is the number of data elements return by 'A'. Then Fan-Out of component 'A' is calculated as e+f+g+h. If the class Ci refers other classes R=Rj…Rn , then this metric is given as below:

Fan-Out	
Mathematical Notation	**OCL Syntax**
Fan-Out= $\sum_{j=0}^{n}$ Ci(Rj)	Context classname Inv: self.associatedclassattribute. rolename condition

Step 3: Estimate each design metric (m € M) value for each component in object model and analyze its criticality

The UML object diagram investigates both the objects in a system and their connections to one another. Object models given in OCL, a formal specification language, can be mined for design complexity metrics that can be used to assess a component's criticality. This allows for a holistic understanding of the module's complexity, from which error-prone components can be measured and analyzed. By estimating the design complexity metre for each component in an OO system, we are able to derive the criticality index for the entire system. When the criticality analysis is complete, the data is captured in a table with columns for the component list and the complexity metrics associated with each component. It is calculated using formulas given in eqn. (1) to (4) as given below:

$$InformationFlow(Ci) = (FANIN * FANOUT) \quad (1)$$

$$ComplexityMetric1 = NOCL + NOI + NOF + NOM + NOP + NSMF \quad (2)$$

$$ComplexityMetric2 = DOI + DAM + DCC \quad (3)$$

$$CriticalityIndex = ComplexityMetric1 + ComplexityMetric2 + InformationFlow \quad (4)$$

Step 4: Conduct severity analysis. Identify high risk components

The most crucial component poses the most risk in the event of a failure. The recommended methodology we've laid out helps pinpoint a group of high-threat parts that need extra attention during the design, development, and testing phases. It is possible to determine the relative severity of each component in a component-based system by considering the two criteria below:

- Analyze component's criticality which is estimated in step 3.
- Estimate normalized complexity of each component using the formula given in eqn. (5).

Normalized complexity of Ci is given by,

$$NormalizedComplexity(Ci) = \frac{Criticalityindex(Ci) - Min(Criticalityindex(Ci))}{Max(Criticalityindex(Ci)) - Min(Criticalityindex(Ci))} \quad (5)$$

Where, i=1 to number of components in the given software. The normalization enables us to assign severity ranking for the components, that is, to identify fault-prone software components from real time software systems.
The severity classification suggested in [45], such as critical, major, and minor, to rank the severity of components is applied in this research work. The output of severity analysis is documented in a tabular column containing a list of components, their normalized complexity, and their severity ranking.

Step 5: Methodology verification using USE tool

In this case, meta-models are generated with the help of USE [46], an information system specification system. It is useful for analyzing requirements. It borrows concepts from the Object Constraint Language (OCL) and a subset of the Unified Modeling Language (UML). In the early phases of software development, the USE tool's [31] primary goal is to animate, test, and validate a UML class diagram and its OCL constraints. From the OCL spec, a '.use' file will be generated. This file will be run and checked by the USE tool.

4. Results and Discussion

The case study is implemented in real time and the results derived are given below along with the discussion on the outcome received.

A. Case Study Implementation

In this section, an E-Commerce application is taken as a case study to show the critical component identification process. The objective of E-Commerce application is to facilitate online interactions between customers and vendors. The case study mostly includes information on order, inventory, billing, shopping, authorization, product and customer information, etc. In these types of systems, a relatively small number of faulty components, as a result of bad specification, analysis, or design, may cause application failure or alter the behaviour of the entire system. In addition, testing components such as inventory, order, and product has a high failure rate and is the most challenging. Obviously, the severity of a failed critical component depends on the nature of the programme.
Generally speaking, an E-Commerce application enables a customer to browse among the many catalogues provided by the seller, choose the required item, and place an order. The order is validated by making sure the client has a contract with the supplier and one or more bank accounts through which payments can be made. If the goods are in stock, the supplier confirms their availability and ships them. Following the receipt of the package, the customer replies with an acknowledgment. An electronic transfer of money from the customer's bank account to the supplier's bank account completes the processing of the invoice [47]. The methodology described in Section 3 is gradually applied to the system under study in the remaining paragraphs of this section. Then, based on our findings, we come to a conclusion.

Step 1: Formalize UML meta-model using OCL specification, build an object diagram Od

Figure 5 is a class diagram of the E-Commerce application. The system is broken down into thirty-four components as indicated in table II. In Figure 5, we see the E-Commerce application class diagram revised as an object diagram. The transition was motivated by the fact that OCL specifications give a more detailed and accurate description of the semantics of the UML model. Object diagrams can also be used to visually verify that an OCL standard has been met. When investigating the framework of a model, object diagrams are a useful tool. In an object diagram, each individual component is depicted by a square box, and the connections between them are shown by connecting lines.

Step 1.1: Convert the object model into a Knowledge Graph (KG) to represent each object as a vertex or node in the graph and the relationship between them as links or edges. Each node is then associated with a weight value calculated from steps 2 to 5

Step 2: Select a set of complexity metrics M = m1, m2... which contribute to filter risky component in a system

Suites of design complexity metrics used for the proposed methodology are enumerated in Table I. The WMC, DAM, NOCL, NOI, NOP, NOF, and NSFM metrics address the component's design complexity, whereas the DOI,

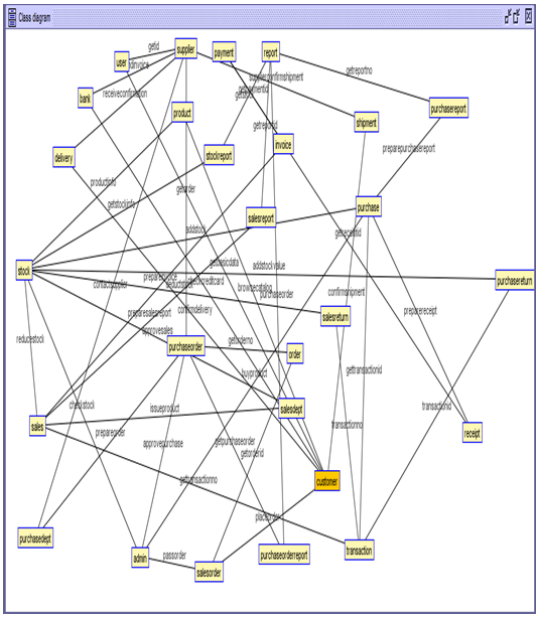


Figure 5. UML class diagram as Knowledge graph for E-Commerce application

NOC, DCC, Fan-in, and Fan-out metrics examine the component's dependability in an object-oriented system. Both complexity and dependability metrics are adequate measures of the components' criticality. The criticality of a component Ci in an object-oriented system is determined by eqn. (6).

$$CriticallityIndex(Ci) = TotalComplexityValue(Ci) + Totaldependabilityvalue(Ci) \quad (6)$$

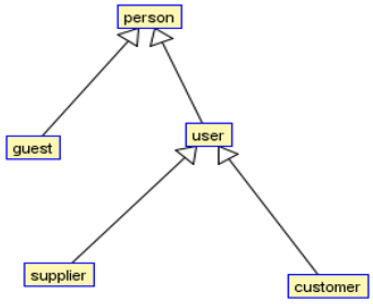
TABLE I. Design complexity metrics selected for the proposed methodology

Metric Suite	Design Metrics (Class Level)	Measure Of	
		Complexity	Dependability
CK Metric	Depth of Inheritance (DOI)		
	Number of Children (NOC)		
	Weighted method per class (WMC)		X
QMOOD	Direct Class Computing (DCC)		X
	Data Access Metric (DAM)	X	
Hendry et al.	Fan-In		X
	Fan-Out		X
Zimmer mann et al. [ref]	Number of classes (NOC)	X	
	Number of Interfaces (NOI)	X	
	Number of fields (NOF)	X	
	Numbef of Parameters (NOP)	X	
	Number Static Fields and Methods (NSFM)	X	

Step 3: Estimate each metric (m € M) value for each component in object model and analyse its criticality

Using the section 3-discussed metric definitions, OCL syntax, and mathematical notations, the complexity and dependability metric values for each component in the E-Commerce case study are extracted from the object diagram. Sample is given in Figure 6.

Figure 6 depicts the OCL expression for E-Commerce application components such as person, user, guest, customer, and supplier. It indicates that the class PERSON has two immediate successors, namely GUEST and USER. Therefore, its NOC and NOCL are 2 and 3. Similarly, the class USER has two direct descendants, namely CUSTOMER and SUPPLIER. NOC of class USER is therefore 1 and NOLC equals 3. Because classes like GUEST, CUSTOMER, and SUPPLIER have no direct successors, their NOC is 0 and their NOCL is 1. Class




```
class person end
class guest <person end
class user <person end
class customer < user end
class supplier < user end
```

Figure 6. Example of DOI, NOC, NOCL metrics extraction

CUSTOMER and SUPPLIER have two forerunners, namely USER and PERSON, as described by the preceding OCL expression, hence their DOI is 2.

Class USER and GUEST are descended from PERSON, hence their DOI is 1. The PERSON root class does not have a predecessor, hence its DOI is 0. Sample is given in Figure 7.

According to the above OCL syntax, the variable sid is incremented with each call to the sid function in class stock. The attribute sid is therefore considered a static variable. Also, static variables can only be utilized by static methods; hence, method sid() and attribute sid are deemed static. So NSMF of class stock equals 2.

The syntax above defines seven characteristics for the class stock, including sid, opstock, date, receivedqty, issueqty, minimumqty, and itemcost. Therefore, its NOA is 7 In addition, two of the nine attributes defined for the class stock are private, including minqty and opstock. Therefore, its DAM of class stock is 4.5. Its representation is given in Figure 8.

The above OCL expression shows that reorderlevel is the

```
abstract class reorderlevel
end
class stock < reorderlevel end
context reorderlevel
inv inv1:  self.reorderqty > 0
invinvr2:self.allInstances->
isFinal(reorderqty
context stock::sid()
pre: self.sid > 0
post: self.sid = self.sid@pre + sid

context stock::stockinfo(sid:Integer)
pre: sid.isDefined()

context stock::stockinfo(sid:Integer)
pre: Date.isDefined()
pre: receivedqty.isDefined()
pre: issueqty.isDefined()
pre: minqty.isDefined()
pre: Itemcost.isDefined()

context stock
invinv1:stock.allInstances->
isPrivate(minqty)
invinv2:stock.allInstances->
isPrivate(opstock
```

Figure 7. Example of NOF, NOP, NSFM metrics extraction

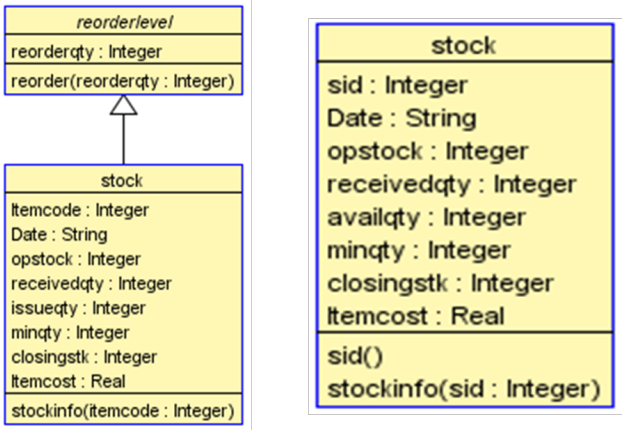
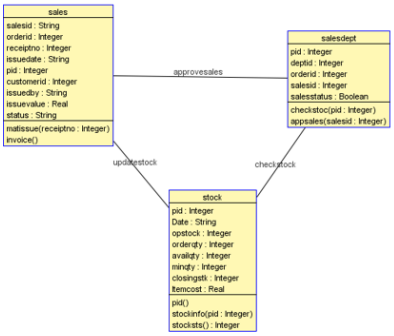


Figure 8. Example of NOI metric extraction

abstract class and contains final variable reorderqty hence, class reorderlevel is considered to be an interface which is implemented by class stock. Hence NOI of class stock is 1. An example is given in Figure 9.

The above OCL expression illustrates the relationship between the class stock and the classes salesdept and sales by defining the role names checkstock and updatestock with approvesales being used between sales and salesdept. This means the DCC value of stock class is 2+2+1=5. And the DCC for sales and saledept is also 5.

Now, the information passing through class component (Ci) is defined by the formula given in eqn. (7).




```
context salesdept
inv.inv1:
self.checkstock.availqty>=0 implies
self.salesstats= true

inv inv2: self.salesstatus = true
impliesself.sales->size()>0

context sales
inv inv2: self.updatestock.availqty=
availqty@pre+1

context stock
invinv3:  self.checkstock.pid-
>includesAll(self.pid)
invinv:   self.updatestock.pid-
>includesAll(self.pid)
```

Figure 9. Example of DCC, Fan-In and Fan-out metrics extraction

$$InformationFlow(Ci) = (FANIN * FANOUT)^2 \qquad (7)$$

OCL syntax and Figure 9 suggests that, for the class component 'stock', the classes 'salesdept' and 'sales' are both the calling components and thus a =2. The two descendant classes namely, salesdept and sales, each passes a single argument named 'pid' to stock class and thus b=2. Consequently, the total number of parameters passed from 'stock' class to the above components is also 1 for each and so, c=2. The two referenced classes each take in one of two parameters, such as availqty or pid. So, we have d=2. Hence Class stock has a Fan-In metric value of 8.

To compute Fan-Out, as discussed in section 3, the value e=2 refers to the fact that the classes stock and sales are both referenced by the salesdept class. Class Stock receives the parameter name 'pid' with a a value of f=2, because it is higher in the hierarchy than class Sales, which receives the parameter name 'pid' with a value of g=2, since it is lower in the hierarchy. Here, the value of h=2 that indicates the return of two separate pieces of information (in this case, 'pid' and 'salesstatus'), and as there are two such items, the Fan-Out metric value is 8.

Class 'stock' is related with 'salesdepartment' and sales. For this reason, we know that a=e=1. As there is only one component higher in the hierarchy that receives the 'pid' argument, the value of b=f=1. When separating 'sales'

by product category, only one piece of information is requested and returned (ie, pid). Therefore, d=h=1 is the correct value. As a result, the class component 'sales', has the fan-in and fan-out metric values as 5.

Consistent with what has been said thus far, we provide predictions about the value of the information flow measure across the given case study in Table II. The values of the extracted metrics and the relative criticality index of each component of the E-Commerce case study are summarized in Table II, which was generated in light of the preceding discussion.

According to the data in the Table II, class 'sales' has a criticality index of 27225 in the E-Commerce case study, followed by Payment at 17424. The next part provides an explanation of how this criticality index value can be utilized to determine the normalized complexity. The criticality index of the E-Commerce case study's components is depicted in a bar chart shown in Figure 10. In Table II, it is observed that, the NOI metric is 0 for almost all the components as this particular case study didn't have more number of interfaces associated with each component. It is purely based on the design decisions taken by the analyst and designer of the software. Also, it is observed that, the Fan-out metric of some of the components are higher which indicates that, these components invoke other components' functionality based on the need of their service.

Step 4: Conduct severity analysis to identify high risk components

Using normalized criticality analysis, the severity of the component can be determined. In the aforementioned illustration, we first identify the related components of each class and then aggregate their metric value. A system's normalized complexity is calculated using the formula given in eqn. (5). Normalized complexity values for the preceding example are given in Table III.

When evaluating the probable consequences of a failure, the components were classified as either "critical," "major," or "minor" [45].

In this work, we use a linear scale to rate the severity of potential failures; more specifically, we assume that components with a normalized complexity greater than or equal to 0.9 are to be treated as mission-critical. If the normalized complexity of a component is less than 0.9 but greater than or equal to 0.19 (approx. 0.2), then we classify it as having a major level of severity and the severity of a component will be deemed low or minor if its normalized complexity is less than 0.19. The normalized complexity is calculated as per eqn. (5) given in section 3.

From Figures 11 and 12, we can identify high-risk components of the given system. This information is valuable for managing error prone components identification and prioritization. By identifying components that are crucial for software development can help distribution of effort by prioritizing the components based on their criticality index value.

Step 5: Methodology verification using USE tool

The object diagram of a given system can be realized using OCL specification with USE tool. The next section deals with the OCL syntax described in the USE tool

Step 5.1: Define class, associations and constraint using OCL expression and derive class diagram in USE Tool:

The OCL specification contains definitions for each class. The definition of a class includes the pattern of its attributes and operations. The following is the sample OCL specification of class Order of E-Commerce application.

– Class definition

model e-commerce

class order

attributes

orderid:Integer

productId: Integer

qty:Integer

date: String

deliveryterms: String

paymentterms: String

deliverymode:String

status: String

povalue: Real

operations

orderid(orderid:Integer)

orderinfo(orderid:Integer)

end

TABLE II. Criticality estimation for components of E-Commerce Case Study

S.No	Component Name	DOI	NOC	WMC	NOCL	NOF	NOP	NOI	NSFM	DAM	DCC	Information flow $(Fan-in * Fan-out)^2$	Criticality Index
1	Admin	0	0	4	1	2	5	0	0	0	4	64	80
2	User	1	2	2	3	4	1	0	2	0	0	576	591
3	Customer	2	0	1	1	2	1	0	0	0	8	256	271
4	Supplier	2	0	1	1	2	1	0	0	0	5	81	93
5	Bank	0	0	2	1	5	1	0	2	0	1	1296	1308
6	Order	0	2	2	3	10	1	0	2	0	0	17424	17448
7	Purchase Order	1	0	1	1	2	2	0	0	0	2	1764	1773
8	Salesorder	1	0	1	1	2	2	0	0	0	2	4900	4909
9	Transaction	0	4	1	5	3	0	0	2	0	0	64	79
10	Purchase	1	0	2	1	7	2	0	2	0	5	12100	12120
11	Sales	1	0	2	1	9	4	0	2	0	5	27225	27245
12	Purchasereturn	1	0	2	1	7	3	0	0	0	1	5184	5199
13	Salesreturn	1	0	2	1	7	3	0	0	0	1	5184	5199
14	Stock	0	0	2	1	9	1	1	2	4.5	5	12100	12125.5
15	Shipment	0	0	1	1	6	2	0	2	0	2	12100	12114
16	Confirmshipment	0	0	2	1	4	5	0	0	0	2	3969	3983
17	Product	0	0	2	1	7	1	0	2	0	2	8100	8115
18	Payment	0	2	2	3	8	2	0	2	0	0	17424	17443
19	Invoice	1	0	1	1	4	2	0	2	0	2	2401	2414
20	Receipt	1	0	1	1	4	2	0	2	0	2	900	913
21	Report	0	5	1	6	1	0	0	2	0	0	16	31
22	Purchasereport	1	0	1	1	4	1	0	0	0	1	2025	2034
23	Salesreport	1	0	1	1	4	1	0	0	0	1	2025	2034
24	Purchaseorderreport	1	0	1	1	6	1	0	0	0	1	5929	5940
25	Stockreport	1	0	1	1	3	1	0	0	0	1	2304	2312
26	Salesorderreport	1	0	1	1	6	1	0	0	0	1	5184	5195
27	Department	0	2	1	3	2	1	0	2	0	0	9	20
28	Purchasedept	1	0	2	1	3	2	0	0	0	6	576	591
29	Salesdept	1	0	3	1	3	3	0	0	0	6	576	593
30	Accessbank	0	0	2	1	7	3	0	0	0	6	14641	14660
31	Guest	1	0	1	1	0	0	0	0	0	0	0	3
32	Person	0	2	1	3	4	0	0	0	0	0	16	26
33	Reorderlevel	0	0	1	1	2	1	0	0	0	1	0	6
34	Employee	0	0	2	1	7	1	0	2	0	1	0	14

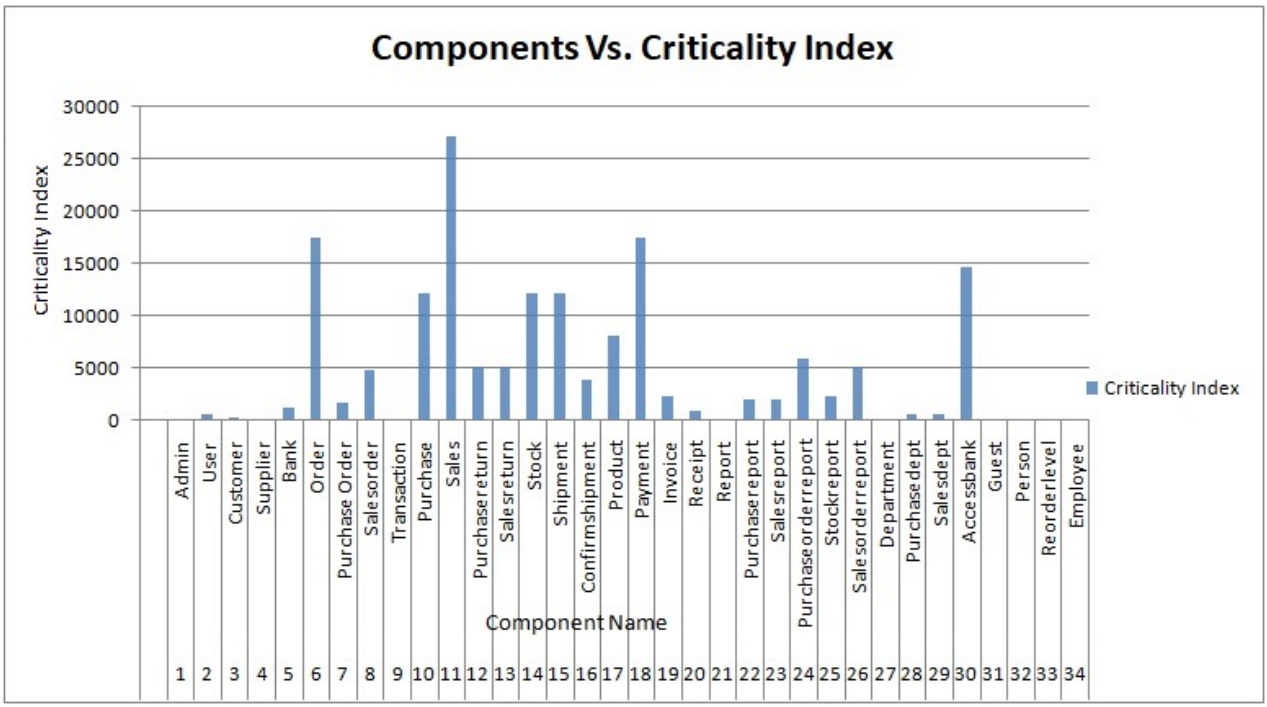


Figure 10. Criticality analysis of E-Commerce application

TABLE III. Normalized complexity estimation and severity estimation for components of E-Commerce Application

Components	Normalized Complexity	Severity	Severity value in Numeric Form
Admin	0.002827	Minor	0
User	0.021584	Minor	0
Customer	0.009838	Minor	0
Supplier	0.003304	Minor	0
Bank	0.047904	Minor	0
Order	0.640371	Major	4
Purchase Order	0.064973	Minor	0
Salesorder	0.18009	Minor	0
Transaction	0.00279	Minor	0
Purchase	0.444791	Major	4
Sales	1	Critical	5
Purchasereturn	0.190735	Major	4
Salesreturn	0.190735	Major	4
Stock	0.444993	Major	4
Shipment	0.444571	Major	4
Confirmshipment	0.146098	Minor	0
Product	0.297775	Major	4
Payment	0.640188	Major	4
Invoice	0.088503	Minor	0
Receipt	0.033404	Minor	0
Report	0.001028	Minor	0
Purchasereport	0.074554	Minor	0
Salesreport	0.074554	Minor	0
Purchaseorderreport	0.217936	Major	4
Stockreport	0.084759	Minor	0
Salesorderreport	0.190588	Major	4
Department	0.000624	Minor	0
Purchasedept	0.021584	Minor	0
Salesdept	0.021658	Minor	0
Accessbank	0.53803	Major	4
Guest	0	Minor	0
Person	0.000844	Minor	0
Reorderlevel	0.00011	Minor	0
Employee	0.000404	Minor	0

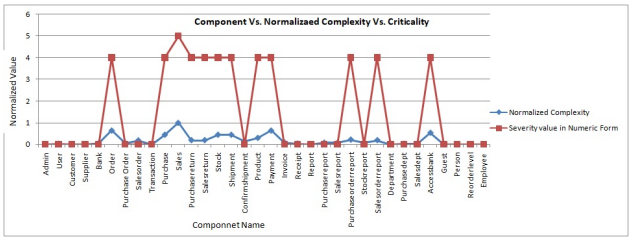


Figure 11. Normalized Criticality Analysis of E-Commerce application

Step 5.2: Representation of dependency between components:

The relation between the components can be represented using OCL as follows.

– Associations

association getorderid between
order[*]
salesorder[*]
end

Step 5.3: Representation of invariants and operations:

Invariants indicate the system structure whereas pre and post conditions denote the system behavior.

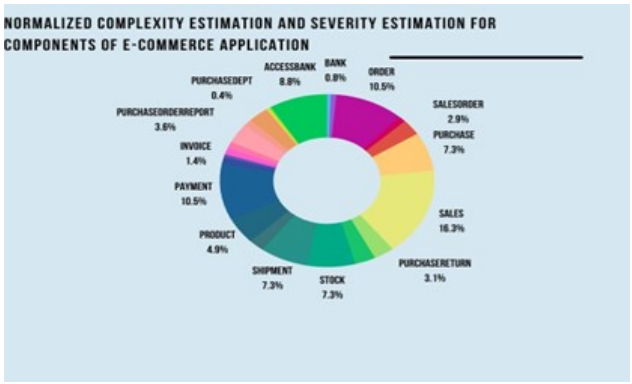


Figure 12. Visualization in terms of normalized Complexity

- Constraints

$$context order$$

$$invinv1 : self.qty > 0$$

$$invinvr2 : order.allInstances- > isUnique(orderid)$$

-Pre and post condition

$$context order :: orderid(orderid : Integer)$$

$$pre : orderId > 0$$

$$post : self.orderId = self.orderId@pre + orderId$$

Step 5.4: Derive the object diagram: The following USE command is used to derive object diagram in Figure 13 based on the OCL syntax discussed in section 3
!insert (customer,order) into orderproduct
!insert(customer,product) into viewproduct

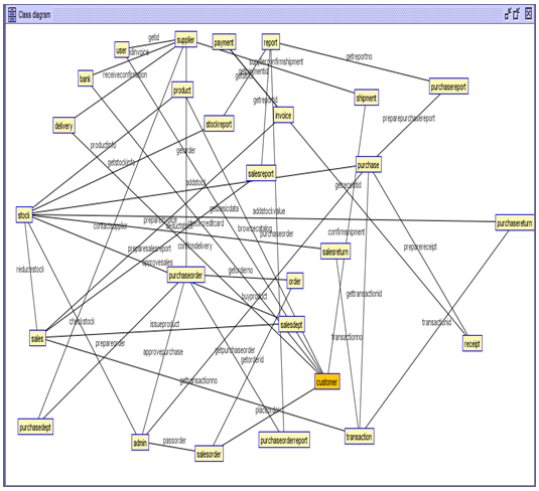


Figure 13. Object diagram of E-Commerce application

B. Result Analysis

1) *Artificial Fault Injection based Critical Components Identification- Ground Truth Verification*
Here, for ground truth verification, this work has applied an artificial fault injection based analysis on the extracted components. The affected components list due to faults injected in the components is collected using dynamic code execution based analysis. The impact level of each of the components is calculated and is given in the range of 1 to 10. It is shown in Table IV. Based on dynamic execution of all the mutated/ artificially fault injected components the impact value of each component over the other components is calculated and is shown in Table IV. The higher impact components are extracted and showcased in figure 14. From the Tables III and IV, it is inferred that, the proposed formal specification based metric analysis and the artificial fault injection based dynamic code execution based impact analysis are yielding the same result of the components' criticality level. Hence, it is concluded that, even before coding, the proposed technique can be applied to extract all the critical components from the given software so that, they can be rigorously tested during the testing phase.

TABLE IV. Components with affected components list with the invoked method

Pid	Component Name	Component Affected	Impact Level
Ecom	Admin	Purchase, Stock, Sales, Purchase Order	4
Ecom	User	Supplier, Salesdept	2
Ecom	Customer	Product, Purchase, Purchase Return, Order, Transaction, Sales Order	6
Ecom	Supplier	User, Product, Bank, Delivery, Shipment, Purchasedept	6
Ecom	Bank	Salesdept, Supplier, Transaction	3
Ecom	Order	Purchaseorder, Salesorder, Product, Purchase, Purchase Return, Transaction, Shipment	8
Ecom	Purchase Order	Product, Order, Purchasedept, stock, admin, purchaseorderreport	6
Ecom	Salesorder	Order, Customer, admin, Product, Purchase, Purchase Return, Order, Transaction, Sales Order, Bank, Transaction, Receipt	10
Ecom	Transaction	Sales, Purchase, Salesreturn, Purchasereturn	3
Ecom	Purchase	Stock, Purchasedept, Purchaseorderreport, receipt, transaction, admin, Purchase Return, Order, Transaction	8
Ecom	Sales	Stock, transaction, Salesdept, Invoice, Salesreport, Order, Customer, admin, Product, Purchase, Purchase Return, Order, Transaction, Salesorder, Bank, Transaction, Receipt	10
Ecom	Purchasereturn	Stock, Purchase, Transaction, Order,, Purchaseorder, Bank, Transaction, Receipt	8
Ecom	Salesreturn	Stock, Sales, Order, Transaction, Salesorder, Bank, Transaction, Receipt	8
Ecom	Stock	Product, Sales, Purchase, Salesreturn, Purchasereturn, stockreport, salesorder, purchaseorderorder	8
Ecom	Shipment	Customer, Supplier, Purchaseorder, Salesorder, Salesreturn, Purchasereturn, Invoice, Receipt	8
Ecom	Confirmshipment	Supplier	1
Ecom	Product	Stock, Customer, Purchaseorder, Salesorder	4
Ecom	Payment	Customer, Supplier, Receipt, Sales, Purchase, Salesreturn, Purchasereturn, Product, Order, Transaction, Salesorder	9
Ecom	Invoice	Sales, Receipt	2
Ecom	Receipt	Purchase, Invoice	2
Ecom	Report	Stockreport, Salesreport, Purchaseorderreport	3
Ecom	Purchasereport	Purchase, Report	1
Ecom	Salesreport	Sales, Stock, Report	1
Ecom	Purchaseorderreport	Purchase, Stock, Purchasereturns, Product, Order, Transaction, Bank, Report	8
Ecom	Stockreport	Stock, Report	2
Ecom	Salesorderreport	Sales, Stock, Salesreturn, Product, Order, Transaction, Bank, Report	8
Ecom	Department	Purchasedept, Salesdept	2
Ecom	Purchasedept	Purchaseorder, Supplier	2
Ecom	Salesdept	Sales, Customer	2
Ecom	Accessbank	Purchase, Sales, Customer, Purchaseorder, Salesorder, Product, Sales, Purchase, Transaction, Admin	8
Ecom	Guest	User, Person	2
Ecom	Person	User, Guest, Customer, Supplier	4
Ecom	Reorderlevel	Stock, Admin	2
Ecom	Employee	Person, Admin, User	3

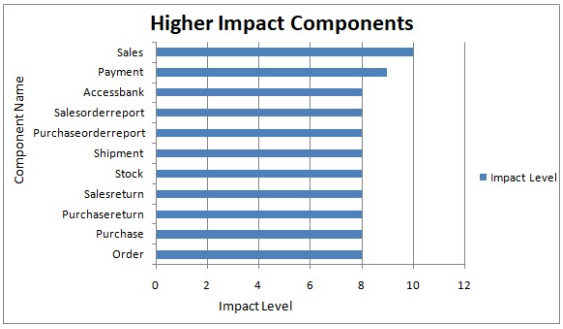


Figure 14. Extracted High Impact Components

2) *Statistical Analysis - Ground Truth Verification* To examine the correlation between the complexity metric calculated for each component and the actual critical software components identified by means of artificial fault injection based analysis is done using the statistical analysis. This correlation based analysis indicates that, the complexity value associated with each component is correlated with the classification of the actual critical components. There is a positive correlation between these two and hence, the proposed work rightly finds out the critical components from the given set of components based on the severity value. The same is plotted in Figure 15. The same is given in Table V.

TABLE V. Statistical Value based Analysis

Parameter	Value
Pearson correlation coefficient (r)	0.8077
P-value	7.744e-9
Covariance	1.9891
Sample size	(n) 34
Statistic	7.7502

The 'r' value is r = 0.8077 ↑

r = Sxy / SxSy = 1.9891 / (1.259)*(1.956) = 0.8077 **p-test**

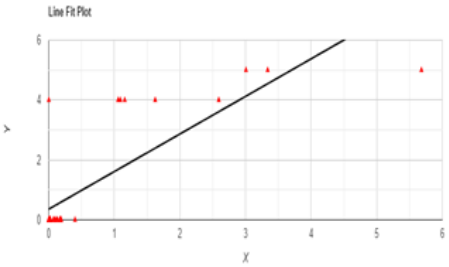


Figure 15. Line Fit Plot on Complexity Value (x) Vs. Component Classification (y)

stat = 0.8077-0/0.1042 = 7.7502
p = p (x ≤ 7.7502) = 1

$p-value = 2*Min(p, 1-p) = 7.744e^{-}9 = 0.0000000007744$

The distribution test is shown in Figures 16(a),(b) and (c).

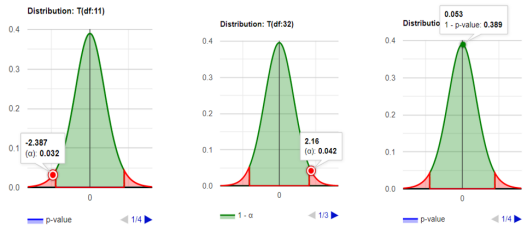


Figure 16. (a), (b), (c) – t-Distribution test for Correlation Analysis

a) H_0 Hypothesis
From the p-test conducted on the case study application, it is observed that the value of p-value is less than α (0.01), and so, the hypothesis H0 is rejected and hypothesis H1 is accepted. That is, there is a close correlation between the severity value and the normalized complexity value calculated.
The correlation analysis of the given case study application states that, the difference between the correlation of the case study and the expected correlation value is bigger to be considered as statistically significant.

b) p(x≤7.7502) = 1; the p-value is 7.744e-9. It indicates that there is a low probability of making a type I error : 7.744e-9 (7.7e-7%). As the p-value is very less, it indicates that, the hypothesis H1 is strongly supported.

Residual's Normality

The Shapiro-Wilk Test, with a value of as 0.05, is used to test the hypothesis of the relationship between the normalised complexity metric value generated using formal specification and the severity value calculated using dynamic code execution based effect analysis.

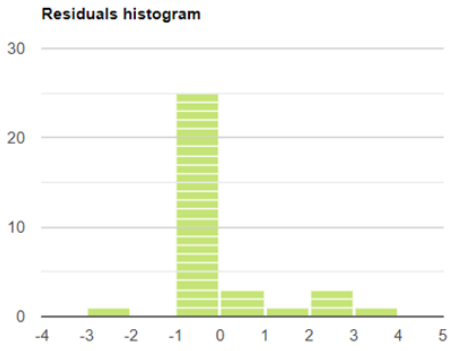


Figure 17. Residual Histogram

Given the p-value, it is assumed that the distribution of the residuals follows a normal distribution. The graphs below demonstrate that we cannot disprove the normalcy assumption. The residuals' normality may be a sign of a bivariate normal distribution. This indicates that the model's inference such as confidence intervals, model predictions, etc. should be accurate given that the assumption is true. Figures 17 and 18 illustrate it.

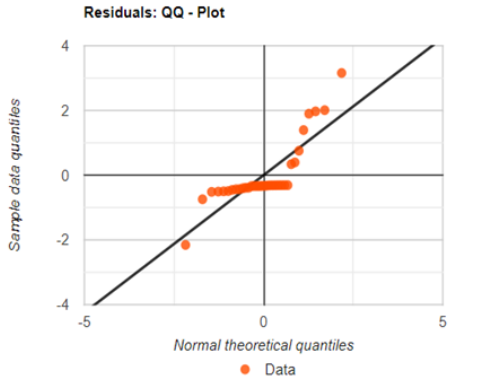


Figure 18. Residual QQ- Plot

C. Discussion and Limitations

The above results have indicated that, the proposed work on the application of Knowledge Graphs and OCL based formal specification to identify the critical components from the given software has yielded promising results. The application of mathematical formal specification will definitely be an opportunity for the development team to identify the critical software components early in the SDLC. This will help them to focus more during the development as well as testing phases so that, the impact of failure due to the improper testing of these components will be reduced. However, it has to be understood that, if the design documents are not prepared with complete analysis of the requirements, the accuracy in the identification of critical components will be reduced as this work heavily depends on the design documents.
Also, the OCL formal specifications are the key to the major success of this proposed work. If the OCL specifications are prepared with less care and omitted some of the crucial dependability and complexity aspects of the requirements, the approach may not provide the expected outcome.
Hence, it is advisable to have the Software Requirements Specification (SRS) must have been prepared with at most care so that, the proposed work will provide the expected outcome with high level of accuracy.

5. Conclusion and Future Work

In this research, a novel approach to identify the critical software components by combining mathematical formal specification with knowledge graphs representation derived from UML design documents with design metrics based analysis is proposed. Here, the criticality of each component in the software was analyzed at the early stages of development through the formal definition of UML design document utilizing OCL and design metrics encoded in Knowledge Graphs (KG). Novel and effective approaches for identifying important software components in a real-time, component-based system, as well as supplementary data for severity analysis, are provided by this study. The

validity and ground truth verification are also done by conducting statistical analysis with artificial fault injection based dynamic impact analysis.

Therefore, this paper relies solely on an analytical approach to identify critical components using formal specification documents with design metrics. Starting with a UML class diagram and an OCL-based formal specification, Knowledge Graphs are used to estimate the complexity value associated with the components. The normalized complexity value is then used to evaluate the severity of each component. This allows for more accurate critical component prediction and sensitivity analysis, as well as the development of a tool for automating the identification of essential components.

The limitation of this research work is that, it has the preliminary assumption that, the UML meta model has no design flaws and is thoroughly validated for its correctness. As a future research work, the proposed approach will be automated to create a tool that automatically calculate metrics based on formal expressions, analyze the findings for complexity and severity, and automate the formal OCL representation of design models in accordance with the UML meta model.

References

[1] I. Terminology, "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[2] C. Ebert, "Metrics for identifying critical components in software projects," in *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*. World Scientific, 2001, pp. 401–436.

[3] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[4] P. Bishop, R. Bloomfield, T. Clement, and S. Guerra, "Software criticality analysis of cots/soup," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2002, pp. 198–211.

[5] I. J. Hayes, "Specification directed module testing," *IEEE transactions on Software Engineering*, no. 1, pp. 124–133, 1986.

[6] V. Cortellessa, K. Goseva-Popstojanova, K. Appukkutty, A. R. Guedem, A. Hassan, R. Elnaggar, W. Abdelmoez, and H. H. Ammar, "Model-based performance risk analysis," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 3–20, 2005.

[7] S. A. El Hayat, F. Toufik, and M. Bahaj, "Uml/ocl based design and the transition towards temporal object relational database with bitemporal data," *Journal of King Saud University-Computer and Information Sciences*, vol. 32, no. 4, pp. 398–407, 2020.

[8] M. Gogolla and T. Stüber, "Metrics for ocl expressions: development, realization, and applications for validation," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020, pp. 1–10.

[9] A. L. Baroni and F. Abreu, "An ocl-based formalization of the moose metric suite," in *Proc. 7th ECOOP Workshop on Quantitative Approaches in Object-Orietend Software Engineering*, 2003.

[10] A. Tang and H. Van Vliet, "Modeling constraints improves software architecture design reasoning," in *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. IEEE, 2009, pp. 253–256.

[11] C. Avila, A. Sarcar, Y. Cheon, and C. Yeep, "Runtime constraint checking approaches for ocl, a critical comparison," 2010.

[12] S. Luke, "Failure mode, effects and criticality analysis (fmeca) for software," in *5th Fleet Maintenance Symposium*, 1995, pp. 731–735.

[13] C. H. Loh and S. P. Lee, "Towards cohesion-based metrics as early quality indicators of faulty classes and components," in *Proceedings of International Symposium on Computing, Communication, and Control (ISCCC 2009)*, 2009.

[14] A. Brahmi, M.-J. Carolus, D. Delmas, M. H. Essoussi, P. Lacabanne, V. M. Lamiel, F. Randimbivololona, J. Souyris, and A. O. SAS, "Industrial use of a safe and efficient formal method based software engineering process in avionics," *Embedded Real Time Software and Systems (ERTS 2020)*, 2020.

[15] Y. Zou and Y. Liu, "The implementation knowledge graph of air crash data based on neo4j," in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, vol. 1. IEEE, 2020, pp. 1699–1702.

[16] B. Abu-Salih, "Domain-specific knowledge graphs: A survey," *Journal of Network and Computer Applications*, vol. 185, p. 103076, 2021.

[17] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D. E. M. Nassar, H. Ammar, and A. Mili, "Architectural-level risk analysis using uml," *IEEE transactions on software engineering*, vol. 29, no. 10, pp. 946–960, 2003.

[18] M. Ray and D. P. Mohapatra, "A novel methodology for software risk assessment at architectural level using uml diagrams," *SETLabs Briefings*, vol. 9, no. 4, pp. 41–60, 2011.

[19] P. Suri and S. Kumar, "Simulator for identifying critical components for testing in a component based software system," *IJCSNS International Journal of Computer Science and Network Security*, vol. 10, no. 6, pp. 250–257, 2010.

[20] R. Malhotra and A. Jain, "Fault prediction using statistical and machine learning methods for improving software quality," *Journal of Information Processing Systems*, vol. 8, no. 2, pp. 241–262, 2012.

[21] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, and

G. Succi, "Identification of defect-prone classes in telecommunication software systems using design metrics," *Information sciences*, vol. 176, no. 24, pp. 3711–3734, 2006.

[22] R. Shatnawi and W. Li, "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process," *Journal of systems and software*, vol. 81, no. 11, pp. 1868–1882, 2008.

[23] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on software engineering*, vol. 32, no. 10, pp. 771–789, 2006.

[24] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," *IEEE Transactions on software Engineering*, vol. 33, no. 6, pp. 402–419, 2007.

[25] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of systems and software*, vol. 56, no. 1, pp. 63–75, 2001.

[26] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[27] T. M. Khoshgoftaar, E. B. Allen, and J. Deng, "Using regression trees to classify fault-prone software modules," *IEEE Transactions on reliability*, vol. 51, no. 4, pp. 455–462, 2002.

[28] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[29] M. Lamrani, Y. Amrani, and Y. Ettouhami, "Formal specification of software design metrics," in *In Proceedings of the 6th International Conference on Software Engineering Advances*, 2011, pp. 348–355.

[30] A. L. Baroni and F. B. Abreu, "A formal library for aiding metrics extraction," in *International Workshop on Object-Oriented Re-Engineering at ECOOP*, 2003.

[31] M. Gogolla, J. Bohling, and M. Richters, "Validating uml and ocl models in use by automatic snapshot generation," *Software & Systems Modeling*, vol. 4, pp. 386–398, 2005.

[32] B. Zhou, J. Bao, J. Li, Y. Lu, T. Liu, and Q. Zhang, "A novel knowledge graph-based optimization approach for resource allocation in discrete manufacturing workshops," *Robotics and Computer-Integrated Manufacturing*, vol. 71, p. 102160, 2021.

[33] N. Ramzy, H. Ehm, S. Durst, K. Wibmer, and W. Bick, "Knowgraph-tt: Knowledge-graph-based transit time matching in semiconductor supply chains," *INFOCOMMUNICATIONS JOURNAL*, vol. 14, no. 1, pp. 51–58, 2022.

[34] J. Deng, C. Chen, X. Huang, W. Chen, and L. Cheng, "Research on the construction of event logic knowledge graph of supply chain management," *Advanced Engineering Informatics*, vol. 56, p. 101921, 2023.

[35] F. B. e Abreu and W. Melo, "Evaluating the impact of object-oriented design on software quality," in *Proceedings of the 3rd international software metrics symposium*. IEEE, 1996, pp. 90–99.

[36] L. Reynoso, M. Genero, and M. Piattini, "Measuring ocl expressions: a "tracing"-based approach," *Proceedings of QAOOSE*, vol. 2003, 2003.

[37] J. Cabot and E. Teniente, "A metric for measuring the complexity of ocl expressions," in *Model Size Metrics Workshop co-located with MODELS*, vol. 6. Citeseer, 2006, p. 10.

[38] K. Magel, R. M. Kluczny, W. A. Harrison, and A. R. Dekock, "Applying software complexity metrics to program maintenance," 1982.

[39] W. W. Eric, "Identify fault-prone software modules in telecommunications systems," in *Motorola 2006 System, Software and Simulation Symposium, Chicago*, 2006.

[40] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[41] B. Jagdish and D. Carl, "Automated metrics and object oriented development," *Dr. Dobb's Journal December*, 1997.

[42] C. C. A. Erika and K. Ochimizu, "Quality prediction model for object oriented software using uml metrics," *IEICE Technical Report; IEICE Tech. Rep.*, vol. 107, no. 505, pp. 49–54, 2008.

[43] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE transactions on Software Engineering*, no. 5, pp. 510–518, 1981.

[44] T. Zimmermann, N. Nagappan, L. Williams, K. Herzig, and R. Premraj, "An empirical study of the factors relating field failures and dependencies," in *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, 2010, pp. 71–80.

[45] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[46] U. R. T. Force, "Omg unified modeling language specification, v. 1.3. document ad," 99-06, Tech. Rep., 1999.

[47] H. Gomaa, "Designing concurrent, distributed, and real-time applications with uml," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 1059–1060.

Dr.D.Jeya Mala is currently working as ASSOCIATE PROFESSOR SENIOR in the School of Computer Science and Engineering (SCOPE), Vellore Institute of Technology, Chennai, Tamil Nadu, India. She has received a design patent from IP, Government of India and has published more than 60 research articles. She is a member of several professional societies such as IEEE, ACM, Indian Science Congress Association, Computer Society of India, i-Soft, an invited member of Machine Intelligence Research Labs etc. She is listed in the Who's Who list of SEBASE repository of University College of London, UK for her research work. Her research interests include Artificial Intelligence, Software Engineering, Software Test Optimization, Machine Learning, Cyber Security and Block Chain.

Mr.A.Pradeep Reynold has completed his Masters in Technology. He has published more than 10 articles in reputed international journals, conferences and hook chapters. He has received several laurels and awards from various national level bodies. He is a life member of Quality Council of India (QCI, Government of India). His research interests include: Real time complex systems analysis, Safety assessment, Environmental Sustainability, Software Engineering and Security.