CHAPTER 2
LITERATURE REVIEW

2.1 Introduction
Component-Based Software Engineering addresses the expectations and requirements of the customers and users as other branches of Software Engineering do. It follows all the development steps and phases as other development paradigms. The standard Software Engineering principles are applicable on the applications developed through Component-Based Software Engineering. Reusability presents the development team to concentrate on the quality aspects of the software [45]. Component-Based Software development emphasizes “development with reuse” as well as “development for reuse”. Development with reuse focuses on the identification, selection and composition of reusable components. The property of reusability is not applied only to develop the whole system but also to develop the individual components. The development for reuse is concerned with the development of such components that may be used and then reused in many applications, in similar and heterogeneous contexts.

Review work is broadly divided into four issues, namely, reusability of components, integration and integration complexities, testing and reliability of Component-Based Software. Critical review of the literature is done with a view to find the research gaps.

2.2 Reuse and Reusability Issues
Reusability is the focal-point of Component-Based Software Development. Software reusability is such an approach that defines the effective reuse of pre-designed and tested parts of experienced software in new applications. In CBSE, we integrate components of all classes according to the design architecture and applications requirements. There are some components for which code is not available called Black-Box components; some components may be available with their code and documentation, which are known as White-Box components. In literature, various researchers have classified software reusability quantification methods into different categories.

Prieto et al. [46] defined some attributes of a program and related metrics to compute reusability in their work. They proposed that reuse depends on size, program structure, documentation, programming language and reuse experience. Further, they used lines of code to count the size,
Cyclomatic complexity for structure, rating from 0 to 10 for documentation, inter-module language dependency to estimate difficulty of modification, and experience of using same module.

Caldiera and Basili [47] proposed one of the earliest methods to identify and qualify reusable components. They defined cost, usability, and quality as the three factors affecting the reusability. They characterized components reusability using four metrics: Volume like, operands and operators, using Halstead Software Science Indicators. Cyclomatic complexity using McCabe’s method to compute the component’s complexity, Regularity measures the component’s implementation economy, and Reuse frequency that is the indirect measure of the functional usefulness.

Jeffrey [48] identified two basic categories of reusability models: empirical and qualitative. Empirical models use experimental data to estimate complexity, size, reliability and similar issues, which can be used by automated tools to estimate the reusability. The qualitative models stay with pre-defined assumptions and guidelines to address issues like quality and certification.

Chen et al. [49] presented their findings based on a large number of reused components. They computed size, program volume, program level, difficulty to develop, and effort for all these reused components. Their conclusion is that, to increase the productivity we should decrease the values of these metrics.

Gregory [50] defines the theory of function, form and similarity to compute the software reusability. Function defines the actions of a component, form characterizes the attributes like structure and size and the similarity identifies the common properties of components. Author uses the well-defined metrics like McCabe’s complexity metric in his calculation.

Barnard [51] reused components developed in C++, Java and Eiffel to show his experimental findings. His work was based on estimations of component’s attributes like simplicity, genericity and understandability through available properties, methods and defined interfaces.

Lee and Chang [52] suggested metrics including the complexity and modularity of components to predict the reusability and maintainability of object-oriented applications. They defined complexity metrics as Internal-External Class Complexity, and modularity metrics as Class Cohesion-Coupling.
Cho et al. [53] proposed component’s reusability measures as the fraction of total interface methods and the number of interface methods in the component that have common functionality in their domain. Reusability is assumed higher as the value of ratio increases. They also defined Customizability metrics as the ratio between the customization methods and the count of methods present in the interface of that component.

Boxall and Arban [54] find in their study that the reuse level of a component is greatly affected by component’s understandability. They derived the value of understandability by using the attributes of component’s interfaces. In their work, Boxal and Arban taken interface size, counts of the argument, number of repetitions, scale of the repetitions and similar attributes to suggest some metrics for understandability.

Washizaki et al. [55] suggested a Reusability-Model for black-box components to reuse components efficiently. Authors identified some reusability affecting factors like functions offered, level of adaptation in new environments and varied requirements. They proposed metrics for better understandability of components.

Bhattacharya and Perry [56] focused on integration contexts of components rather than concentrating just on internal attributes of components. They proposed reusability estimations considering the integration architecture of the software. They have proposed Components characteristics and Architecture compliance metrics to measure the issues related to particular properties and integration issues of components.

Gui and Scott [57] suggested metrics to estimate the coupling properties among inter-components and cohesion properties of intra-components in their own terms. They used java components to propose their metrics.

Gui and Scott [58] further defined a metric-suite to compute the reusability of Java components and ranked them according to their reusability. These metrics are used to estimate the indirect coupling among components as well as to assess the degree of coupling.

Wijayasiriwardhane and Lai [59] suggested a size measurements technique named Component point to estimate the size of overall component-based systems. These Component points can be reused as a metric to analyse components for future use. They defined three classes of components according to their usability.
Danail et al. [61] categorized reusability metrics into two broad classes: the White-Box metrics and the Black-Box metrics. *White-box reusability metrics* are based on the logic and coding structure of the software. On the other hand, *Black-box reusability metrics* are based on interfaces and other attributes, as code is not available for black-box components.

Reusability measures defined in literature are analysed by considering development paradigms like: Convention Software and Programs, Objet-Oriented Software, and Component-Based Software and summarized in Table 2.1.

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Measures and Metrics Used</th>
<th>Key Findings</th>
<th>Factors affecting reusability</th>
<th>Author(s)/References</th>
</tr>
</thead>
</table>
| Conventional Software and Program | • Lines of code,  
• Cyclomatic complexity,  
• Rating from 0 to 10 for documentation,  
• Inter-module language dependency to estimate difficulty of modification, and  
• Experience of using same module. | • Defined some attributes of a program and related metrics to compute reusability in their work.  
• They proposed that reuse depends on size, program structure, documentation, programming language and reuse experience. | • Size,  
• Program structure,  
• Documentation,  
• Programming language, and  
• Reuse experience | Prieto-Diaz, Ruben and Peter Freeman [46] |
| Component-Based Software      | • Halstead Software Science Indicator to find volume,  
• *Cyclomatic complexity* using McCabe’s method. | • *Regularity* measures the component’s implementation economy,  
• *Reuse frequency* that is the indirect measure of the functional usefulness | • Cost,  
• Usability, and  
• Quality | G. Caldiera and V.R. Basili [47] |
| Component-Based Software      | • Automated tools to estimate the reusability,  
• Predefined assumptions | • Two categories of reusability:  
*Empirical models* use experimental data to estimate complexity, size, reliability and similar issues.  
*Qualitative models* stay with pre-defined assumptions and | • Cost, and  
• Reliability | Jeffrey S. Poulin [48] |
| Object-Oriented Software | • Used reused components,  
• Modularity  
• Complexity | • Presented their findings based on a large number of reused components.  
• Increase the productivity we should decrease the values of these metrics | • Size,  
• Volume,  
• Level,  
• Difficulty to develop, and  
• Effort | Chen, Deng-Jyi and P.J. Lee [49] |
|---|---|---|---|---|
| Component-Based Software | • Structure of components,  
• McCabe’s complexity | Defines the reusability on the basis of:  
• Function defines the actions of a component,  
• Form characterizes the attributes like structure and size and the  
• Similarity identifies the common properties of components. | • Functions of the components,  
• Size,  
• Complexity of components | W. Gregory Hislop [50] |
| Component-Based Software | • Reusable components,  
• Automated tools | Proposed the importance of reusability in his experimental findings.  
• His work was based on estimations of component’s attributes like simplicity, genericity and understandability through available. | • Available properties,  
• Methods and  
• Interfaces | J. Barnard [51] |
| Object-Oriented Software | • Internal-External Class Complexity, and  
• Modularity metrics as Class Cohesion and,  
• Class Coupling. | To estimate the reusability and maintainability of object-oriented software metrics considering the complexity and modularity of components. | • Complexity  
• Modularity | Y. Lee and K.H. Chang [52] |
| Component-Based Software | • Metrics for reusability,  
• Metrics for Customizability. | Proposed metrics for Reusability and Customizability:  
• The ratio of total number of interface methods to the number of interface methods in the component that have common  
• Total number of Interface methods,  
• Total number of common function Interface methods,  
• Total number of | E.S. Cho, M.S. Kim, and S.D. Kim [53] |
functionality in their domain.  
- Reusability is assumed higher as the value of ratio increases.  
- Ratio between the customization methods and total number of methods present in the interface of the component.

| Component-Based Software | Customization methods,  
<table>
<thead>
<tr>
<th></th>
<th>Total number of common function customization methods.</th>
</tr>
</thead>
</table>
|                          | In their study they argued that the reuse level of a component is greatly affected by component’s understandability.  
|                          | Derived the value of understandability of a component by using the attributes of interfaces of that component. |
|                          | Understandability,  
|                          | Interface property |
|                          | M. Boxall, and S. Araban [54] |

| Component-Based Software | Proposes Component Reusability Model,  
|                          | Component’s Observability: Readability of component/Total Properties.  
|                          | Component’s Customizability: Writability of component/Total Properties. |
|                          | Functionality of the component,  
|                          | Adapting the changes in requirement,  
|                          | Porting to the new environment |
|                          | H. Washizaki, Y. Hirokazu and F. Yoshiaki [55] |

| Component-Based Software | Interaction contexts,  
|                          | Properties and attributes of components  
|                          | Architecture of CBS |
|                          | Focused on integration contexts of components rather than concentrating just on internal attributes of components.  
|                          | They proposed reusability estimations considering the integration architecture of the software.  
<p>|                          | They proposed Components characteristics and Architecture |
|                          | Integration architecture |
|                          | S. Bhattacharya and D.E. Perry [56] |</p>
<table>
<thead>
<tr>
<th>Compliance Metrics</th>
<th>Description</th>
<th>Complexity of Components</th>
</tr>
</thead>
</table>
| Object-Oriented Software | • Coupling and Cohesion metrics | Estimate the coupling properties among inter-components and cohesion properties of intra-components.
| Component-Based Software Engineering | • Coupling between objects, Response for class, Coupling Factors | • Ranked components according to their reusability. Defined metrics to estimate indirect coupling, degree of coupling, and functional complexity.
| Component-Based Software | • Interface complexity, Interaction complexity, Internal Logical files, External Interface files, Number of operations and interactions. | • Size measurement metric named Component point to estimate the size of the overall component-based system. Three classes of components: User components, Service components, and Domain components. Size, Weighting factor low, average and high Unadjusted component point.
| Component-Based Software | • Reusability metrics for available code, Interface properties | **White-box reusability metrics** are based on the internal logic and coding structure of the software. **Black-box reusability metrics** are based on interfaces and other attributes, as code is not available for black-box components.

Measures identified in the literature are defined for conventional software modules, functions or object-oriented classes. Researchers and practitioners paid relatively less focus on exploring the reusability of partially-adaptable and fully-adaptable components of Component-Based Software. In addition, metrics defined in literature are not applicable to all categories of components like
new, fully-qualified, partially-qualified and off-the-shelf components. We can not simply apply Black-Box methods or White-Box reusability metrics on the whole Component- Based Software, to assess the reusability of components in heterogeneous environment [60].

2.3 Interaction and Integration Complexity Issues

Component-Based Software applications are composed of independently deployable components. Assembling of these components has a common intension to contribute their functionalities to the system. Technically this assembling is referred to as integration of and interaction among components. We have sufficient number of measures and metrics to assess the complexity of stand alone programs as well as small-sized conventional software, suggested and practiced by numerous practitioners [42, 62, 63, 64, 65, 66, 67]. In literature, complexity of programs and software is treated as a “multidimensional construct” [62] [68]. Complexity depends upon various factors of a program [63].

Thomas J. McCabe [69] developed a method to assess the Cyclomatic complexity of a program. He used control-flow graph of code to compute the complexity. McCabe used graph theoretic notations to draw the control-flow graph where a graph denoted as ‘G’ having ‘n’ number of nodes, ‘e’ number of connecting edges and ‘p’ number of components. Cyclomatic complexity \( V(G) \) calculated as, 
\[
V(G) = e - n + 2p
\]
where 2 is the “result of adding an extra edge from the exit node to the entry node of each component module graph” [42]. In control-flow graph, a sequential block of code or a single statement is represented as a node, and control flows among these nodes are represented as edges. Cyclomatic complexity metric is easy to compute and maintenance, gives relative complexity of various designs.

Halstead's [64] identified a complete set of metrics to measure the complexity of a program considering various factors. These metrics include the program vocabulary, length, volume, potential volume, and program level. Halstead proposed methods to compute the total time and effort to develop the software. These metrics are based on the lines of codes of the program. He defined program vocabulary as the count of distinct operators and distinct operands used in the program. The count of total operators and operands used in a program is proposed as the Program length. The Program volume has been defined as the storage volume required representing the Program, and the representation of program in the shortest way without repeating operators and operands is known as potential volume. Halstead has also defined the relationship between these factors and metrics of programs.
Alan Albrecht [65] proposed Function-point analysis technique to measure the size of a system in terms of functionalities provided by the system. FPA categorizes all the functionalities provided by the software in five specific functional units: *External inputs* provided to the software, *External outputs* provided by the software, *External inquiries* of the system under consideration, *Internal logical files* presents data and content residing in the system, and *External interface files* are the data and contents residing with other systems and can be called to system under consideration. Three complexity weights High, Low and Medium are associated with these functional units using a set of pre-defined values. In function-point analysis, 14 complexity factors have been defined, which have a rating from 0 to 5. On the basis of these factors, Alan calculated the values of unadjusted function-point, complexity adjustment factors, and finally the value of function points [42].

Henry and Kafura [70] proposed a set of complexity computation method for software modules. Author’s suggested a “Software Structure Metrics Based on Information Flow that measures complexity as a function of fan-in and fan-out” [71]. Authors proposed the complexity as “the procedure length multiplied by the square of fan-in multiplied by fan-out.” This method is used to calculate the count of “local information flows” coming to (fan-in) and going from (fan-out) the module. Henry and Kafura defined a length of the module as the procedure length which calculated with the help of LOC or McCabe's complexity metric. This metric can be computed comparatively early stage of the development.

Kenneth Morris [72] proposed some object-oriented metrics to assess complexity and productivity metrics. Author’s identified some complexity factors like Maintainability, Reusability, Extensibility, Testability, Comprehensibility, Reliability and Authorability, that they called “productivity impact variables”. Morris proposed a complete set of nine eligible metrics for Methods, Class, Inheritance, Coupling and Cohesion.

Boehm [66] developed the ‘object-point’ metric through level of complexity of the amount of screenshots, reports and components. The level of complexities is categorized as simple, medium or difficult.

Chidamber and Kemerer's [73] proposed a metric suite for object-oriented software called as CK Metrics-suite. This metric suite is one of the most detailed and popular research works for object-oriented applications. Authors defined metric suite for complexity, coupling cohesion, depth of inheritance, and response set. These metric set are used to assess the complexity of an individual
class as well as the complexity of the entire software system. In their metrics, Chidamber and Kemerer used Cyclomatic method for the complexity computation of individual classes.

Abreu and Rogerio Carapuca [74] [75] [76] proposed a metric set named ‘Metrics for Object-Oriented Design’. In this metric suite, two fundamental properties of object-oriented programming are used, attributes and methods. Authors proposed metrics for the basic structural system of object-oriented idea as encapsulation, inheritance, polymorphism, and message passing. This suit consists metrics for methods and attributes as assessment method for encapsulation.

Cho et al. [53] developed some measure for the quality and complexity of components for CBSE. They used mathematical equations and expressions in their metrics. In their work, authors identified three categories of complexity, quality of component, customizability and reusability. They used size, costs, efforts, and reuse level as the complexity factors.

Washizaki et al. [55] proposed metrics through a hierarchical model consisting of three layers, quality, criteria and metrics. They identified factors like understandability, adaptability and portability that affect the complexity of the Component-Based Software.

Narasimhan et al. [77] suggested couple of metrics to assess the complexity of Component-Based Software. The packing density metric maps the count of integrated components, and the interaction density metric is used to analyse the interactions among components. They identified some constituents of the component in their work; these constituents include line of code, operations, classes, and modules. Authors also suggested a set of criticality criteria for component integration and interaction.

Vitharana et al. [78] developed a method for fabrication of components. Authors suggested some managerial factors like cost-efficiency; assembling easiness, customization, reusability, and maintainability. These are used to estimate technical metrics as coupling-cohesion, count, volume and complexity of components. They developed ‘Business Strategy-based Component Design’ model.

Kung-Kiu Lau and Zheng Wang [79] argued that reusability is not only the purpose of components integration but it is also a systematic process to construct the software system. To fulfil the basic objectives of CBSE, Kung and Zheng analyse an idealized component life cycle and suggested that
components should be composed according to the life cycle. Further they pointed out that the language of composition should have proper and compatible syntax and semantics.

Rashmi Jain et al. [80] assesses the association and mappings of cause-and-effect among the requirements of the system, structural design of the system and the complexity of the procedure of the systems integration. They argued the requirement of fast integration of components so that the complexity impact of integration on architectural design of components can be controlled. Authors identified 5 major factors to analyse the integration complexity of software system. Further these factors are divided into 18 sub-factors including commonality in hardware and software subsystems, percentage of familiar technology, physical modularity, level of reliability, interface openness, orthogonality, testability and so on.

Trevor Parsons et al. [81] proposed some specific dynamic methods for attaining and utilising interactions among the components in component-based development. They also proposed component-level interactions that achieve and record communications between components at runtime and at design time. For their work, authors used Java components.

Lalit and Rajinder [82] proposed a set of integration and interaction complexity metrics to analyse the complexity of Component-Based Software. They argue that complexity of interaction have two implicit features, first within the component, and second interaction from the other components. Their complexity metrics include percentage of component interactions, interaction percentage metrics for component integration, actual interactions, and total interactions performed, complete interactions in a Component-Based Software.

Sharma and Kushwaha [84] presented an integrated method to assess the development and testing efforts through analysing the “improved requirement based complexity (IRBC)” in the context of Component-Based Software.

Some complexity assessment techniques for CBSE are [66, 85, 86] on the basis of complexity properties including communication among components, pairing, structure, and interface. The interaction and integration complexity measures available in the literature are explored considering the development paradigms like: Convention Software and Programs, Objet-Oriented Software, and Component-Based Software and summarized in Table 2.2.
<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Measures and Metrics Used</th>
<th>Key Findings</th>
<th>Factors affecting Interaction and Integration Complexity</th>
<th>Author(s)/ References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional Software and Programs</td>
<td>• Line of Code, • Interaction among Statements, • Nodes and Interactions</td>
<td>• Author used control flow graph of a program to compute the Cyclomatic complexity.</td>
<td>• Conditional Statements, • Loop Statements, • Switch cases</td>
<td>Thomas J. McCabe [69]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• McCabe used graph theoretic notations to draw the control flow graph where a graph G with n vertices, e edges and p connected components.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conventional Software and Programs</td>
<td>• Line of Code, • Count of operators, • Count of Dissimilar operands, • Total count of Dissimilar operators, • Total count of Dissimilar operands.</td>
<td>Proposed a complete set of metrics to measure the complexity of a program considering various factors, like Program vocabulary, Program length, Program volume, Potential volume, and others.</td>
<td>• Program Vocabulary, • Program Length, • Program Volume, • Effort, • Time</td>
<td>M. H. Halstead [64]</td>
</tr>
<tr>
<td>Modular Programming</td>
<td>• External inputs • External outputs • External Enquiries • Internal logical files • External Interface files</td>
<td>Proposed Function-point analysis technique to measure the size of a system in terms of functionalities provided by the system.</td>
<td>• 5 functional units, • 14 Complexity factors, • Complexity adjustment factors, • Degree of influence.</td>
<td>Alan Albrecht and J. E. Gaffney [65]</td>
</tr>
<tr>
<td>Modular Programming</td>
<td>• Fan-in information, • Fan-out information, • Complexity of the module • Line of Code • McCabe Cyclomatic Complexity</td>
<td>• Author’s suggested a “Software Structure Metrics Based on Information Flow that measures complexity as a function of fan-in and fan-out”.</td>
<td>• Number of calls to the module, • Number of calls from the module, • Length of the module</td>
<td>S. Henry and D. Kafura [70]</td>
</tr>
<tr>
<td>Object-Oriented Software</td>
<td>• Methods, • Inheritance, • Coupling, • Cohesion,</td>
<td>Proposed some object-oriented metrics to assess complexity and productivity metrics,</td>
<td>• Maintainability, • Reusability, • Extensibility,</td>
<td>Kenneth Morris [72]</td>
</tr>
</tbody>
</table>
| Object-Oriented Software | Lines of code to count the size, Number of Screenshots, Number of reports | Authors suggested the object-point metric that is computed using counts of the number of screenshots, reports and components based on their complexity levels. Complexity levels are classified as simple, medium or difficult. | Line of Code, Complexity Levels | B. Boehm [66]  

Object-Oriented Software | Cyclomatic method, Class complexity, Methods, Object-oriented properties | Proposed one of the most detailed and popular research works in the context of object-oriented software, including Weighted Method per Class, Depth of Inheritance Tree, Number Of Children, Lack of Cohesion in Methods | Complexity, Coupling, Cohesion, Inheritance, Number of children, and Response set | S. Chidamber and C. Kemerer [73]  

Object-Oriented Software | Method Hiding Factor, Attribute Hiding Factor, Method Inheritance Factor, Attribute Inheritance Factor for Inheritance, Polymorphism factors, Coupling factors | Authors identified two fundamental properties of object-oriented programming are used, attributes and methods. The Method Hiding Factor and (MHF) and Attribute Hiding Factor (AHF) are proposed together as measure of encapsulation. | Encapsulation, Inheritance, Polymorphism, and Message passing | Fernando Brito and Rogerio Carpuca [74]  

Component-Based Software | Levels of complexity, Quality of components, Customizability. | Proposed metrics to measure the quality and complexity of components. They used mathematical equations and expressions in their metrics. | Size, Costs, Efforts, and Reuse level | E.S. Cho, M.S. Kim, and S.D. Kim [53] |
<table>
<thead>
<tr>
<th>Component-Based Software</th>
<th>• Indicates these values as high or low, • Establishes a relationship among these proposed metrics.</th>
<th>• Proposed metrics through a hierarchical model consisting of three layers, quality, criteria and metrics.</th>
<th>Understandability, Adaptability, and Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>H. Washizaki, Y. Hirokazu and F. Yoshiaki [55]</td>
</tr>
<tr>
<td>Component-Based Software</td>
<td>• Line of code, Operations, • Classes, and • Modules, • Number of components</td>
<td>• Suggested two sets of metrics to assess the complexity of Component-Based Software. • They derived two complexity metric suites Component Packing Density metrics and Component Interaction Density.</td>
<td>• Risk associated with components, • Constituents, • Interactions among components, Narasimhan et. al. [77]</td>
</tr>
<tr>
<td>Component-Based Software</td>
<td>• Coupling, • Cohesion, • Number of Components, • Component Size, • Complexity.</td>
<td>• Proposed a methodology for fabrication of components.</td>
<td>Syntax and Semantics Padmal Vitharana, Hemant Jain, and Fatemeh “Mariam” Zahedi [78]</td>
</tr>
<tr>
<td>Component-Based Software</td>
<td>• Prioritization of Requirements, • Functional Modularity, • Feasibility, • Interface, • Testability</td>
<td>• Assesses the association and mappings of cause-and-effect among the requirements of the system, architecture of the system and the complexity of the procedure of the systems integration. • Identified 5 major factors to analyse the integration complexity of software system. • Further these factors are divided into 18 sub-factors</td>
<td>• Commonality in hardware and software subsytems, • Percentage of familiar technology, • Physical modularity, • Level of reliability, • Interface openness, • Orthogonality, testability Rashmi Jain, Anithashree Chandrasekaran, George Elias, and Robert Cloutier [80]</td>
</tr>
<tr>
<td>Component-Based Software</td>
<td>• Static Interaction complexity, • Dynamic Interaction</td>
<td>• Proposed some specific dynamic methods for attaining and utilising</td>
<td>Call traces, Call graphs, Runtime paths Trevor Parsons, Adrian</td>
</tr>
</tbody>
</table>
complexity, interactions among the components in component-based development. • They also proposed component-level interactions that achieve and record communications between components at runtime and at design time.

Calling context trees Mos, Mircea Trofin, Thomas Gschwind, and John Murphy [81]

Component-Based Software

• Interface,
• Implementation,
• Deployment,
• Incoming and
• Outgoing interactions

• Proposed a set of integration and interaction complexity metrics to analyse the complexity of Component-Based Software, including Percentage of component Interactions, Interaction percentage metrics for component integration, Actual interactions.

• Maintainability,
• Reusability, and
• Reliability Latika Kharb, Rajender Singh [82]

Methods and metrics proposed so far in the literature are defined on the basis of interactions among instructions, operations, procedures, and functions of individual and standalone programs and codes. These metrics are appropriate for small-sized codes. Some measures are also defined for object-oriented software, but for CBSE applications these methods are not inadequate. In the CBSE, components have connections and communications with each other to exchange services and functionalities. Interaction edges are used to denote the connections among components. So there is an edge for each requesting communication and similarly an edge for each responding communication. But practitioners and researchers have not included both edges in their complexity computations. They have used single edge theory in their graph representations and in all their assessments, which is not true for CBSE [38] [83].

2.4 Testing Issues

Testing is one of the critical and crucial phases of the overall development of software. Testing is the fundamental activity to verify the correctness, precision and compatibility of the software at the individual level as well as the system level. Practitioners identified that the improper testing results
to untrustworthy and undependable products [87], [88]. In today’s software development environment, testing commences just after the finalization of systems requirements.

In the CBSE, testing starts from the component level and move forwards to the integrated CBS system level [42]. Testing is commonly used to verify and validate software [89], [90], [91]. Verifying components in CBSE constitutes the collection of procedures to certify the functionalities of components at individual level. Validating components include the group of procedures to assure the integrity of integrated components according to the architectural design and fulfilling the needs of the customer. In the literature testing techniques has been divided into two classes: Black-Box and White-Box.

2.4.1 Black-Box Testing

Black-Box testing methods focus on the behaviour of the software. In this technique the functional behaviour of the software is assessed through inputs provided and outputs observed. Black-box testing treats the internal logic of code as a black-box and the testing observations are captured using inputs and outputs only.

Following are the some Black-Box testing strategies proposed in the literature [92, 93, 94, 95, 96, 97, 98] and classified as:

i. Boundary-Value Analysis (BVA)

ii. Equivalence-Class Partitioning (ECP)

iii. Decision Table-Based Testing (DTB)

iv. Cause-Effect Graphing (CEG)

i. Boundary-Value Analysis (BVA)

In the Boundary-Value Analysis (BVA) testing, assumption is that rather than within the limits of the conditions, a large number of errors occur at the boundaries. In the BVA emphasis is on verifying that the software behaves normally (accurately) at the extreme boundaries [92], [93]. Boundary value analysis method is used to draw test cases focusing on the input values at the “edges” of the logical conditions [94]. This method usually tests the centre values, boundary values, as well as the values just below the boundary and just above the boundary.
ii. Equivalence-Class Partitioning (ECP)

The Equivalence-Class Partitioning (ECP) is a black box testing strategy based on the assumption that if we divide the input values into classes or partitions on the basis of their validity or invalidity then the impact of each value under the particular class will be equivalent. That is the validity of every data member of the legitimate class is equal or similar; and every data member of an illegitimate class is equally invalid. In ECP the input domain is partitioned and tested for validity. Usually ECP is applied to the input classes, but in sometimes it is applied to the output classes also [95], [96].

iii. Decision Table-Based Testing (DTB)

The Decision Table-Based Testing (DTB) testing method is used to test the permutations of input conditions rather than testing single input values. Decision-table based testing was proposed to overcome the limitations of the BVA and the ECP. The limitation is that the BVA and ECP are applicable only when we have to test single values of input and output domain. The DTB testing method combines two or more than two input logical conditions and assesses and examines these complex conditions. There are four quadrants in Decision table, Condition stub: combine the input conditions, Action stub: specify the output conditions, Condition Entries: input the entries of condition entries and Action Entries: input the action entries, shown in Table 2.3.

<table>
<thead>
<tr>
<th>The four quadrants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Stub</td>
</tr>
<tr>
<td>Action Stub</td>
</tr>
</tbody>
</table>

In the DTB method, the Condition stub and Condition entries correspond to an input condition or set of input conditions. The Action stub and Action entries correspond to output or set of outputs [97], [98].

iv. Cause-Effect Graphing

The Cause effect-graphing method is also used to combine the input conditions so that more elaborative assessment can be performed. Here in this technique, inputs are recognized as causes and outputs are recognized as effects. We draw a Boolean graph known as cause-effect graph by joining these causes (inputs) and effects (effects). It is a simple graphing technique where nodes represent input conditions and edges represent linking between nodes.
2.4.2 White-Box Testing

White-Box techniques are used to address the testing requirements of the structural design and internal code of the software. White-box testing methods ensure that all execution paths and independent logics of the program or module have been tested adequately. This testing is also responsible for testing the logical decisions given in the code must be tested on their true and false sides, all looping and branching codes at their extremes as well as within the boundaries must be checked at least once. To analyse the structure of the software, we have Basis Path testing technique, which is used in White-Box technique.

Thomas J. McCabe [69] proposed a method to count the number test cases of a program. He proposed Cyclomatic complexity based on the structural design of the code. He defined a control-flow graph having ‘n’ nodes, ‘e’ edges and p connected components. Cyclomatic complexity is calculated as $V(G) = e - n + 2p$, here 2 is the “result of adding an extra edge from the exit node to the entry node of each component module graph” [41].

Henderson-Sellers [99] proposed another formula to compute the Cyclomatic complexity. This proposed formula was an amendment to the McCabe’s formula for complexity [41]. Henderson-Sellers’ defined, “$V(G) = e - n + p + 1$”. This altered formula argued that to make components strongly linked, an edge is added, and is denoted as constant 1 to the multi-component flow graph [8]. Author used the concept of modularity.

A. Orso et al. [100] suggested a technique of using component’s metadata in the testing of components. This metadata consists of data and control dependencies, source code, complexity metrics, security attributes, information retrieval mechanisms and execution procedures. Authors suggested using this metadata in regression testing of components and their interfaces. They also proposed a Specification-based Regression Test Selection technique for CBS.

Michael R. Lyu et al. [101] proposed that the system’s effectiveness testing can be increased by regulating the factors of the components. These factors that eligible for optimization are cost, reliability, effort, and similar attributes of the system components. Authors considered single as well as multiple application systems for “software component testing resource allocation”. They used “reliability-growth curves” to model the association between the failure rates and “cost to decrease this rate”. Authors used interaction among components and and failure rates of components in their methodology.
Bixin Li et al. [102] developed a matrix-based method to compute the dependences in Component-Based Software. Authors identify some categories of dependencies available in Component-Based Software and they also defined ‘component-dependence graph’ and the ‘dependence matrix’ to record the dependencies. Taken dependence-graph and dependence-matrix as the basis authors proposed a mathematical foundation for assessing these dependencies in Component-Based Software.

Jehad Al Dallal and Paul Sorenson [103] introduced a method called “all-paths-state” to produce state-based test cases through testing at class level. They identified Framework Interface Classes to cover the maximum number of specifications. Authors suggested that this technique will help to generate reusable test cases effectively. They also proposed that this framework design is equivalently helpful for reusable modules as well as newly developed modules.

Gill and Tomar [104] presented their work focusing on testing requirements and documentation process of test cases of Component-Based Software. Authors suggested that the access to the source code, functions provided by the component, compatibility with other components, components middleware, interactions made by various reusable components, specifications of the components, as the testing requirements of Component-Based Software. Authors also discussed the boundaries of the testing of Component-Based Software.

Tamal Sen and Rajib Mall [105] proposed a testing technique based on state and dependence of components. This technique is called as regression test selection technique targeting the regression test suite. They argued that this method will reduce the number of test cases in regression testing. Authors’ assumption is that the link between the state model in design phase and the executable code is very strong, and while maintaining the design state model code is altered without affecting the state of the component.

Gordon Fraser and Andrea Arcuri [106] proposed an approach of covering all coverage goals of testing at a time rather than on coverage goal at a time. Authors proposed that covering all coverage goals at a time leads to a better and effective result in terms of minimizing test suites. For their work, authors have used object-oriented software.

Stephen H. Edwards [107] described test case methods for black-box and white-box components in CBSE. This technique lied on the concept of generating flow-graph from the specifications of
components, and then applying conventional graph coverage methods. Authors focused on test case design using pre-condition, post-condition, and fault detection ratios.

Hervé Chang et al. [108] proposed a technique to built and install reusable integration connectors effectively. They define these connectors as the solution of problems arises during the integration of off-the-shelf components. Authors defined that these problems in the form of exceptions. These connectors are usually installed on the basis of integration information available with the components. They suggested the technique so that these connectors can heal exceptions automatically. They provide the complete structure, exception behaviour and the integration behaviour of these healing connectors.

Testing and test case generation techniques available in the literature are reviewed considering the nature of testing pattern. We have identified two categories of testing patterns: White-Box testing and Black-Box testing. Summary of testing issues of CBSE are shown in Table 2.4.

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Measures and Metrics Used</th>
<th>Key Findings</th>
<th>Testing Factors</th>
<th>Author(s)/ References</th>
</tr>
</thead>
<tbody>
<tr>
<td>White Box Testing</td>
<td>• Statements of the Code</td>
<td>• McCabe used graph theoretic notations to draw the control flow graph and proposed a formula for Cyclomatic complexity. • This formula gives the count of test cases. • It also computes the number of independent logics in the program code. • Here vertices denote the instruction of the code and edges present flow of controls among vertices.</td>
<td>• Conditional Statements, • Loop Statements, • Switch cases</td>
<td>Thomas J. McCabe [69]</td>
</tr>
<tr>
<td></td>
<td>• Interaction among Statements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Control flow graph</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Conditional Statements, • Loop Statements, • Switch cases</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Multi-component flow graph</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Source code</td>
<td>• Proposed techniques for regression testing of components using metadata</td>
<td>• Metadata including data and</td>
<td>A. Orso, M. J. Harrold, D.</td>
</tr>
<tr>
<td></td>
<td>• Interaction among</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4 Summary of Testing Issues of Component-Based Software
<table>
<thead>
<tr>
<th>Testing Method</th>
<th>Components/Specifications</th>
<th>Methodology/Techniques</th>
<th>Dependencies/Attributes</th>
<th>References</th>
</tr>
</thead>
</table>
| **Black-Box Testing**  | - Components specification  
- Components specification  
- Also suggested regression test method using software specification. | - Proposed a methodology for to optimize the testing schedules, subject to reliability constraints.  
- To generate optimization opportunities techniques in testing phase.  
- Proposed that the effectiveness of the system testing can be increased through regulating the factors of the components. | - Cost,  
- Reliability,  
- Failure rate,  
- Testing schedule | Michael R. Lyu, Sampath Ranganajan, and Aad P. A. van Moorsel [101] |
| **White-Box Testing**  | - Inter-component Interaction,  
- Single application,  
- Multi application systems | - Proposed a matrix method to assess the dependences in Component-Based Software named component-matrix graph, and dependence matrix.  
- They identified and defined categories of dependencies including: State, Cause and effect, Input/Output, Context, and Interface-Event Dependence. | - Data,  
- Control,  
- Time,  
- Context,  
- State | Bixin Li, Ying Zhou, Yanc heng Wang and Junhui Mo [102] |
| **Black-Box Testing**  | - All-transition coverage  
- Pair coverage  
- Full predicate coverage  
- Round-trip path coverage | - Proposed a testing method all path-state to reduce the number of test cases.  
- Authors used black-box and white-box methods of testing. | Reusable and newly developed classes, | Jehad Al Dallal, and Paul Sorenson [103] |
| **White-Box Testing**  | - Source code,  
- Interactions,  
- Specification of components | - Focused on two aspects of testing Component-Based Software: testing requirements and documentation process of test cases.  
- Suggested that source code, functions, compatibility, middleware, interactions, specifications as the testing guidelines and testing process step wise and phase wise. | Guidelines and testing process step wise and phase wise. | Nasib S. Gill and Pradeep Tomar [104] |
In Component-Based Development, software systems are partitioned into a number of small and manageable components. Due to this division the architectural design as well as the behavioural

| White-Box Testing | Dependency graph, Source Code, Component state, Reverse engineering | Authors proposed a regression selection technique to minimize the count of test cases in Component-Based Software. They also proposed an algorithm to automatically generate the code from the state model. | Method parameters, Operations, Conditions (like looping and branching) | Tamal Sen and Rajib Mall [105] |
| Black-Box Testing | EVOSUITE tool, Line of code, Number of test cases | Proposed all coverage testing technique to cover goals of testing at a time rather than on coverage goal at a time. Claimed that this approach will minimize the number of tests. They used Primitive, Constructor, Field, Method and assignment statement in their study. | Object-oriented classes, Member variables, Numeric, Boolean, String, and Enumeration variables Test suites | Gordon Fraser and Andrea Arcuri [105] |
| White-Box Testing | Pre-condition, Post-condition, Fault injection, Mutants, Test cases | Proposed specification-based test case generation technique. Used flow graphs defined with the help of specifications. | Components specification, Enumerating paths | Stephen H. Edwards [107] |
| White Box Testing | Integration among Components, UML Modeling, Reusability | Proposed the concept of healing connectors from the developer’s point of view. Authors defined that these problems in the form of exceptions. These connectors are usually installed on the basis of integration information available with the components. | Proved through case studies. | Herve Chang and Leonardo Mariana, Mauro Pezze. [108] |
attributes of the software changes. To adapt changes, various categories of components are engineered and used in varied frameworks. Afterward, those currently engineered components and the existing components are assembled according to a pre-planned architecture to contribute their functionalities. None of the techniques defined in literature are applicable in the scenarios where we have two or more than two components. To calculate, the number of test cases in white box testing we have some techniques like Cyclomatic complexity (though it is used in the context of individual components only). In the context of black-box testing, we have no such technique that can calculate the number of test cases for Component-Based Software [38] [60].

2.5 Reliability Issues
Reliability is one of the key issues of a system’s quality. Reliability defines not only the correctness but also the precision attribute of software. In literature there are a number of quality reliability models and reliability assessment methodologies are coined to predict the reliability of CBSE applications. To develop quality software, we have to integrate various independent components according to the architectural design of the software. To measure the reliability of such software and in a heterogeneous environment, one cannot apply the traditional reliability metrics only. Different researchers have categorized software reliability in diverse categories. Eusgeld [109] defined software reliability as:

a. Black box reliability,

b. Metric based reliability, and

c. Architecture based reliability or white box reliability.

a. **Black-box reliability** [110, 111, 112] emphasizes on the reliability assessment through considering the failure observations of the software over a period of time during testing. These reliability estimation models treat the system as a whole rather than treating its internal structure and intra-component interactions.

b. **Software metric based reliability** [113] estimates the reliability of the software by analysing the software’s static properties like lines of code, asymptotic complexity of the software, experience of developer, process of Software Engineering methods and techniques used for testing.

c. **Architecture-based reliability** [114] focuses on the estimation of the reliability by considering the internal structure of the software. During reliability prediction and assessment, architecture
represents the way of integration and interaction among components. While calculating the reliability of the software, components reliabilities are also considered.

Goseva et al. [115] categorized all the architecture-based reliability estimation methods into three broad categories:

a. State-based estimation methods,

b. Path-based models, and

c. Additive methods.

a. *State-based estimation methods* [116, 117, 118] use the notion of control graph to show the software architecture as well as the interaction among components. In state based models, the basic assumption is that components fail independently. In this technique component’s states follow the Markov behaviour i.e. the current state and the past states of a component are independent to each other.

b. *Path-based models* [114, 115, 119] estimate the reliability by considering the possible execution paths of the program. From starting component to ending component, every possible path is extracted. On the other hand, additive models neither consider the Markov behaviour nor the architecture of the software.

c. *Additive methods* [120, 121] use a ‘Non-Homogeneous Poisson Process’ (NHPP) to model the reliabilities of the components. In these models, the software failure is represented in terms of NHPP through the count of number of component failures and the intensity functions of failures for individual component.

In the literature, there are number of reliability models and reliability assessment methodologies have been coined to assess the reliability of Component-Based Software.

Littlewood’s architecture-based model [122] is one of the initial models developed for reliability estimation of intra-modules and inter-modules. In this architectural model, author proposed that components as well interfaces can fail with constant failure rates. In this model, continuous time Markov chain is used to estimate the software reliability.
Cheung’s reliability model [123] is a state-based model, uses control-flow graph to represent the components architecture. In this work, transition probability matrix is drawn to show the control transfer among components.

Kubat’s model [117] used execution times of components. In this estimation method discrete time Markov chain is considered to represent transition between the components. Probabilities of execution of components are taken into account during reliability estimation of CBSE applications.

Gokhale [118] used random production of faults in components to estimate the failure of components at arrival. In this technique, reliability is estimated through simulation by counting the number of failures of components. Here the basic assumption is the availability of failure rates and repair rates of components.

Sanyal [124] proposed reliability estimation technique based on component dependency graph and fault propagation delay. Author used reverse engineering technique along with the source code, assuming that the logic and code of the component is accessible. This approach may not be helpful for black-box components or the components whose code is not accessible.

Krishnamurthy and Mathur [114] have given a path-based reliability model which was based on test data, test cases and execution paths of the applications architecture. Their assumption was based on the theory that components fail independently and the interfaces are error free. This method computes the path reliabilities by executing the sequence of components one after one and then averaging all to assess the reliability of the software.

Everett’s reliability estimation model [120] assesses components reliability using extended execution growth model. In this growth, model the duration of processing of each component is tracked to estimate the reliability of CBSE applications. This method considers the parameters inferred from component’s attributes and from the knowledge of test cases as well as operational usage. Thus technique considers the execution-time of individual components. Here the total count of failures as well as failure intensities is added as the function of reliability of the software.

H. Singh et al. Reliability prediction [125] model uses unified modeling language to analyse the reliability of Component-Based Software. They used some scenarios and case studies to estimate the reliability in initial phases of construction of the software. They proposed their work with the integration of UML models, and failure rates of components. Authors proposed an algorithm to
predict the reliability of CBSE applications considering the number of components and number of scenarios and their probability of execution.

Cortellessa et al. [126] suggested a reliability assessment technique for Component-Based Software. Their goal is to collect component’s failure rates and connector failure rates. Authors assumption is that the every component’s individual reliability is available. Authors used UML diagrams, deployment diagrams, and sequence diagrams to denote the communications between components. This technique also assumes that components fail without affecting each other.

Reussner et al. [127] proposed a technique to assess the reliability by analysing the usage profiles and the environments component reliability. They proposed that the component’s reliability can be calculated as a factor of the usage profile including the reliabilities of external services. This technique is applicable to black-box components. This method of reliability prediction is applied to open and distributed systems. Components of this method follow the Markov chain property.

Sherif-Yacoub et al. [128] proposed a ‘scenario-based’ reliability estimation approach to for the reliability of CBS. Authors used predefined execution scenarios by defining component dependency graph. By using dependency graphs they proposed some algorithms to compute the reliability of the CBSE applications and further extended the algorithm for distributed components and hierarchy of subsystems for reliability estimations.

G.N. Rodrigues et al. [129] given a reliability estimation method considering the structure of the component and the scenarios present in component-based application. This method is not for components reliability, but it assumes that the component’s reliability is given. In this method component interactions are taken in general. In this technique, components follow Markov property. Authors have assumed that components failures are independent.

Swapna Gokhale and Kishor Trivedi [130] analysed and proposed the reliability prediction Component-Based Software considering its design structure, and the failure behaviour of the components. Authors proposed a unifying framework for the prediction of reliability of state-based models as well as for architecture-based software. This proposed work combines the software’s architecture with the components failure behaviour and the interfaces into a composite method which are analysed to predict the reliability of the application.
Vibhu Saujanya Sharma and Kishor S. Trivedi [131] developed an ‘architecture-based unified hierarchical model’ to assess and predict the performance of the software, reliability of the application, security and cache behaviour of the components. Authors used ‘discrete time Markov chains (DTMCs)’ to model software and defined equations for assessing the overall behaviour of the software application on its architectural design and the behavioural properties of the standalone components. In their hierarchical model they covered ‘reliability’, ‘performance’, ‘security’ and ‘cache-miss’ behaviour predictions.

Fan Zhang et al. [132] suggested an approach based on a component-dependence graph. Basically this technique is sub-domain technique based on architectural-path reliability model, and suggested algorithms to estimate the reliability of the CBSE applications. Authors assumed that the software operational profile is known. They assumed that control flows from one component to the other component. This method is capable of categorizing software applications reliability on change in the operational profile.

Hsu et al. [133] developed an adaptive method using path-based prediction for complex CBS systems. They used three techniques for the estimation of path reliability that are, sequence, branch and design of loops. The path reliability is utilized to estimate the reliability of the whole application. This method also suggests the effect of failure for every component in the overall reliability of the application.

Marko Palviainen et al. [134] proposed an assessment method for the prediction of reliability of components, and the CBSE applications. This work defines a method for reliability prediction of under development components, initially during the development phases of software. This method also suggests the effects on reliability through selection of right components. Authors combine the heuristic, model-driven reliability prediction and techniques at the level of component and at the level of system to explore the development of reliable CBS applications.

Lance Fiondella et al. [135] suggested a reliability assessment technique using ‘correlated failures’. They defined an estimation of components reliability considering reliabilities of individual components, correlation of failures, and the architecture of the software. This proposed method follows Multivariate Bernoulli distribution to compute the overall Component-Based Software reliability.

The reliability methods available in the literature are explored by considering the path-based paradigm of estimation and summarized in the Table 2.5.
<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Measures and Metrics Used</th>
<th>Key Findings</th>
<th>Reliability Estimation Factors</th>
<th>Author(s)/ References</th>
</tr>
</thead>
</table>
| Path-based          | • Dependence graph  
   • Components                                                      | • Proposed Program Dependency Graph and Fault Propagation Analysis for analytical reliability estimation of component based-applications.  
   • The approach is code-based (Reverse-engineering) where dependency graphs are generated from source code.  
   • This method is not applicable for off-the-shelf components. | • Source code,  
   • Interactions                                                      | Sanyal et al. [124]                               |
| Path-based          | • Test cases,  
   • Components,  
   • Intra-component dependencies                                      | • Proposed method estimates the path reliability based on the sequence of components executed for each test run and then averaging them over all test runs to obtain an estimate of the system reliability.  
   • This method is based on test data, test cases and execution paths of the applications architecture.  
   • Their assumption was based on the theory that components fail independently and the interfaces are error free. | • Component s,  
   • Interfaces,  
   • Execution paths,  
   • Component s reliability,  
   • Execution sequence of components                                    | S. Krishnamurt hy and A. P. Mathur [114]            |
| Scenario-based(Path-based) | • UML Diagrams,  
   • Use Case diagrams,  
   • Sequence diagrams,  
   • Independence failure,  
   • Regularity failure,  
   • Number of components.                                                   | • Proposed a reliability estimation technique using unified modeling language.  
   • Authors used pr-defined scenarios and case studies to predict the reliability in early phases of development.  
   • Proposed an algorithm considering the number of components and number of scenarios and their probability of execution. | • Reliabilities of components ,  
   • Operational failures,  
   • Probability of execution of use case,  
   • Number of sequence diagrams                                           | H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj [125] |
<p>| Path-based          | • Use case diagrams,                                             | • Propose an estimation method for reliability of Component-                | • User profiles,                           | V. Cortellessa,                                 |</p>
<table>
<thead>
<tr>
<th>Method</th>
<th>Diagrams/Models Used</th>
<th>Assumptions</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence-based</td>
<td>• Sequence diagrams, • Deployment diagrams, • Calculating formulas</td>
<td>Based Software aiming components and connector failure rates.</td>
<td>H. Singh and B. Cukic [126]</td>
</tr>
<tr>
<td>Deployment</td>
<td></td>
<td>• This method assumes that individual component’s reliability is known.</td>
<td></td>
</tr>
<tr>
<td>diagrams</td>
<td></td>
<td>• For component modeling UML diagrams are used.</td>
<td></td>
</tr>
<tr>
<td>Calculating</td>
<td></td>
<td>• This method estimates that various components fail independently.</td>
<td></td>
</tr>
<tr>
<td>formulas</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path-based</td>
<td>• Control Instructions, • Kens, • Markov property, • Probability of call of components, • Service reliability, • Overall reliability.</td>
<td>• This method is used to estimate the reliability of Component-Based Software by composition of usage profiles and the reliability of environment components.</td>
<td>R. H. Reussner, H. W. Schmidt, and I. H. Poernomo [127]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Method is applicable to black-box components also whose code is not available.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Components fail independently.</td>
<td></td>
</tr>
<tr>
<td>Scenario-based</td>
<td>• Component dependency graph, • Interaction among components, • Transition probability, • Transition reliability.</td>
<td>Proposed a scenario-based reliability estimation method for Component-Based Software.</td>
<td>Sherif Yacoub, Bojan Cukic, and Hany H. Ammar [128]</td>
</tr>
<tr>
<td>(Path-based)</td>
<td></td>
<td>• Assuming that profiles of execution scenarios are available.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Authors proposed a stack-based algorithm to assess the reliability.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• This methodology can be applied in the early phases of the software development since execution scenarios are designed in the design phase.</td>
<td></td>
</tr>
<tr>
<td>Path-based</td>
<td>• Execution Scenarios, • Multiple scenarios, • Markov Property, • Scenario specification, • Interaction among</td>
<td>Given a reliability estimation method considering the structure of the component and the scenarios present in component-based application.</td>
<td>G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel [129]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• This method is not for components reliability, but it assumes that the component’s reliability is given.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Interactions among</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operational profiles, • Scenarios, • Component s reliability</td>
<td></td>
</tr>
</tbody>
</table>
components, components are taken in general.
• This method follows Markov property.
• Authors have assumed that components failures are independent.

<table>
<thead>
<tr>
<th>Path-based</th>
<th>Proposed a method of reliability prediction named “Composite Method” that combines the architecture of the application with the failure behaviour of the components and the interfaces into a composite model which can then be analysed to predict the reliability of the application.</th>
<th>Probability of transition from one component to other, Reliability of components, Expected number of visits, Expected time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Proposed an architecture-based unified hierarchical model to assess and predict the performance of the software, reliability of the application, security and cache behaviour of the components. Proposed a method of security prediction and cache-miss analysis by taking software architecture into consideration. This model is fairly accurate as it takes into account the second-order architectural effects also.</td>
<td>Variance of the number of visits to each component, Component, Reliability information for each component, Vulnerability index of each component,</td>
</tr>
<tr>
<td></td>
<td>Proposed a sub-domain based reliability estimation technique. This approach uses dependency among components to analyse the</td>
<td>Reliability of components, Weight factor</td>
</tr>
<tr>
<td></td>
<td>needed to predict the reliability of the component.</td>
<td></td>
</tr>
</tbody>
</table>

Swapna S. Gokhale, and Kishor S. Trivedi [130]

Vibhu Saujanya Sharma, Kishor S. Trivedi [131]

Fan Zhang, Xingshe Zhou, Junwen Chen, Yunwei
<table>
<thead>
<tr>
<th>Component Type</th>
<th>Components</th>
<th>Description</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transition probability, Transition reliability.</td>
<td>components. • Here operational profiles are assumed to be known. • Reliability is defined as the product of transition probability from one component to other, reliability vector on each sub domain of component, and the weight vector for each sub domain of component achieved during the transition.</td>
<td>during transition</td>
<td>Dong [132]</td>
</tr>
<tr>
<td>Path-based</td>
<td>Modular graph, Transition probability, Failure rate of node</td>
<td>An adaptive framework of incorporating path testing into modular software reliability estimation is proposed. • Authors used three techniques for the estimation of path reliability that are, sequence, and branch and loop structures. • Then this path reliability is utilized to estimate the reliability of the whole application. • This method also suggests the effect of failure of each component on the overall reliability of the application.</td>
<td>Chao-Jung Hsu and Chin-Yu Huang [133]</td>
</tr>
<tr>
<td>Path-based</td>
<td>Component integration, Component level reliability, System level reliability, UML diagrams</td>
<td>Proposed reliability prediction and estimation technique that helps in finding the appropriate software components and integrating them in a feasible way. • Estimating the component properties, • predicting the reliability of components, • Measuring the provided reliability values of the components, • Making decisions regarding which components satisfy the required reliability characteristics and are to be selected for integration.</td>
<td>Marko Palviainen, Antti Evesti, Eila Ovaska [134]</td>
</tr>
</tbody>
</table>
We cannot simply apply black box reliability or white box reliability metrics on the whole Component-Based Software which consists of different types of components. In addition, components share interactions and integrations among them. Hence while assessing the reliability of CBS; we have to consider the integration, interaction complexities and reusability issues of components into account.

2.6 Research Gaps Identified
There are various issues identified during the literature review, and some of them are considered as research gaps. These gaps are defined as- Lack of:

- **Measures and metrics to asses the reusability at individual component level and at system level in Component- Based Software Engineering:** Measures identified in the literature are defined for conventional software modules, functions or object-oriented classes. Researchers and practitioners paid relatively less focus on exploring the reusability of new, fully-qualified, partially-qualified and off-the-shelf components. In addition, single category of reusability metric (Black-box or White-Box) is not suitable for the whole CBSE application, since these complex applications contain both Black-Box and White-Box components.

- **Selection methods of components other than the Requirements Criteria:** If we have a number of components available for alike intentions and we have to select single out of these available components, there should be some selection and verification criteria to select that component in addition to the .

- **Interaction and Integration measures and metrics to capture the complexities for components in Component-Based Software:** When components interact, they have different configurations, diverse complexity levels, coupling issues and incompatible interface problems. These complexities have a great impact on the quality of a system based on components. It is not only sufficient but necessary to have apposite and suitable metrics to quantify the complexities produced by these interactions and integrations. Most of the interaction, integration and complexity quantification methods available in literature are program oriented i.e., these methods are suitable for small scale software. But for large, complex software as component-based systems these metrics are inefficient.

- **Testing and Test Case Generation Techniques for Component-Based Software:** For conventional software we have a collection of testing techniques and methods. There are
Black-Box, White-Box and other techniques through which we can test our software. But for CBSE we don’t have well specified, clear testing techniques and test case generation techniques.

- **Reliability prediction and assessment metrics for Component-Based Software:** A major category of reliability estimation methods defined in literature are either hardware oriented or conventional software/program oriented. They have not considered the basic feature of CBSE, that is, reusability in their reliability metrics. In addition, components share interactions and integrations among them. Hence while assessing the reliability of CBS; we have to consider the integration, interaction complexities and reusability issues of components into account.

- **Measures to identify the level and degree of Componentization:** There are no established measures and metrics to find the optimized minimum cost level of componentization. Trade-off between the number of components and the integration cost is not defined yet.

- **Cost estimation techniques to predict the overall cost of CBS products:** Practitioners have paid comparatively less attention to explore the hidden attributes of Component-Based Software, like overall cost estimation techniques can be developed in this domain.

- **Certification Procedures and Methods to certify the validity and usability of components:** There are no standard procedures to certify the validity of components in Component-Based Software, especially for third party components. We have no defined method to verify components usability other than the customer’s requirement specification.

- **Risk assessment techniques:** There are no standard techniques for risk analysis in Component-Based Software. Risks associated with the components use, as well as, to estimate the risk factors of overall Component-Based Software have a great scope.

In this thesis, some of these identified gaps are defined as the statement of the problem and mentioned in section 1.4. Some investigations are carried out to develop and suggest appropriate solutions for these unexplored problems.

### 2.7 Summary

In this chapter a detailed review of the available literature is given on the basis of thorough and critical review. Some major research gaps are identified in the above defined fundamental areas of Component-Based Software.