A Comparative Study on the Reliability Efforts in Component-Based Software Systems[*]

Atef Mohamed and Mohammad Zulkernine

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6
{atef & mzulker}@cs.queensu.ca

Technical Report No. 2009-559

[*]This report should be cited as follows:

A. Mohamed and M. Zulkernine, *A Comparative Study on the Reliability Efforts in Component-Based Software Systems*, Technical Report No. 2009-559, School of Computing, Queen's University, Kingston, Ontario, Canada, June 2009.

Abstract

Component-based technologies are increasingly proving efficiency in current software applications. The enormous expansion of these applications has increased the demands for reliable component-based technologies. Depending on different viewpoints and assumptions, a component takes various definitions and forms (*e.g.*, class, function, module, agent, architectural component, or model-based component). As a result, numerous reliability works that involve varieties of the underlying strategies, objectives, and parameters are proposed for CBSS reliability. Surveying these reliability efforts is important for creating and selecting potential solutions that handle the reliability of CBSS applications. In this paper, we provide a taxonomy of the reliability efforts in component-based software based on the following reliability means: fault-tolerance, reliability assessment, and reliable design and operational activities. For each of these categories, we provide a comparative study of the proposed approaches, discuss the strengths and weaknesses, and show how do they integrate and contribute towards building dependable component frameworks. We compare and contrast the existing techniques considering their assumptions with respect to component definition and specification details.

Contents

List of Figures

List of Tables

1 Introduction

Component-based Software Systems (CBSSs) revolutionized the development of software technology using a divide and conquer paradigm to distribute software complexities among smaller subsystems or components. A CBSS is a set of interacting components. A component is a unit of composition and deployment that complies with a component model. Depending on different viewpoints and assumptions, a component takes various definitions and forms (*e.g.*, class [18], function [48], module [36], agent [60], architectural component [39], or model-based component [69]).

CBSS development is recognized as a standard paradigm for structuring large software systems. CBSS architectures depict various architectural styles (*e.g.*, distributed [60], mobile [44], client/server [28], peer-to-peer [53], and event-driven [16]). Component models and frameworks provide the required prerequisites that enable component integrations and interactions with the environments of CBSSs. Currently, several component models and frameworks are available in the industry. Some major examples are DCOM, COM+ [62], JavaBeans, Enterprise JavaBeans [70], and CORBA component model [79].

Component technology can increase the productivity of software systems by providing reusability, abstraction, composition, and fast development time [4, 15, 42]. The enormous expansion of component-based software applications has increased the demands for reliable component-based technologies. Software system reliability is the probability that a system performs its intended function for a specified period of time under a set of specified environmental conditions. The means to achieve software reliability are fault avoidance (or prevention), fault removal, fault/failure forecasting, and fault-tolerance. A number of approaches have been proposed for CBSSs to improve their reliabilities. Due to the diversity of CBSS architectures and component definitions, reliability efforts involve a variety of methodologies and viewpoints.

1.1 Motivations and challenges

Surveying reliability efforts is important for identifying and enhancing potential solutions for the reliability of CBSSs. In this survey, our research goals are to find appropriate answers for the following questions. What are the current research directions for CBSSs with respect to their reliability objectives? What are the component issues, assumptions, and parameters that are exploited in the current CBSS reliability techniques? How do the varieties of component definitions and specification details impact the current CBSS reliability research? How does the current CBSS reliability research integrate towards building dependable component frameworks?

The main challenges of this survey are as follows. No consensus has been reached about what a component is and how it is specified [51]. The relations between system architecture and component models are not precisely defined [15]. The definitions of several dependability attributes overlap with each other. Different viewpoints, assumptions, abstraction levels, and system environments complicate the problem of classifying reliability efforts in CBSSs.

1.2 Contributions

In this paper, we provide a taxonomy of the reliability efforts in component-based software based on the following reliability means: fault-tolerance, reliability assessment, and reliable design and operational activities. For each of these categories, we provide a comparative study of the underlying strategies, research objectives, and parameters. We show the strengths and weaknesses of these techniques and discuss how do they integrate and contribute towards building dependable component frameworks. We compare and contrast the existing techniques with respect to the major assumptions of component definition and specification details. The survey will identify a number of open issues that require more attention in the future research.

1.3 Paper organization

The structure of this paper is as follows. In Section 2, we briefly describe basic component terminologies and define software reliability. We describe the major reliability means in CBSSs and provide an overview of the proposed taxonomy. Sections 3, 4, and 5 provide detailed survey about fault tolerance, reliability assessment, and reliable design and operational activities of CBSSs, respectively. In each of these sections, we classify the corresponding work and provide the details about each class, and finally provide summary and related discussions. We present our conclusion and identify the limitations of the existing work and some open research issues in Section 6.

2 Classifying reliability efforts in CBSSs

Although the concept of componentization was introduced in the late 1960's, the practical application of this concept was adopted during the early 1990s. Since that time, a considerable amount of research on CBSS reliability has been proposed with different assumptions regarding component definition and specification details. Due to the diversity of these definitions and specification details, current reliability efforts adopt various strategies and exploit various parameters. In this section, we show our proposed taxonomy of the reliability efforts in CBSSs. Prior to describing the taxonomy, we briefly describe basic component terminology and define software reliability.

2.1 Component models and frameworks

A component is an independent unit within a system. Different from other software entities, the structure and behavior of a component are assumed to comply with a component model. A component is defined as a unit of composition and deployment with contractually specified interfaces, explicit context dependencies only, and no persistent state [15, 80]. It has one or more "provided" and/or "required" interfaces. An interface is defined as a mean by which a component connects to other components or the environment [71, 80]. In a CBSS, there is no limitation on the number of components, component sizes, or the number of interface connections of each component.

Component models and frameworks provide the required prerequisites that enable component integrations and interactions with the environments of CBSSs. A component model specifies the standards and conventions that components need to follow during component composition and interaction, and a component framework provides design time and run time infrastructure for components [7]. Currently, several component models and frameworks are available in the industry. Some major examples are DCOM, COM+ [62], JavaBeans, Enterprise JavaBeans (EJB)[70], and CORBA Component Model (CCM) [79]. In general, the specification of component technologies identifies a number of concepts such as architecture[1], configurations, connectors, bindings, properties, hierarchical models, style, and behavior.

[1]Architecture and system architecture are used interchangeably in this paper.

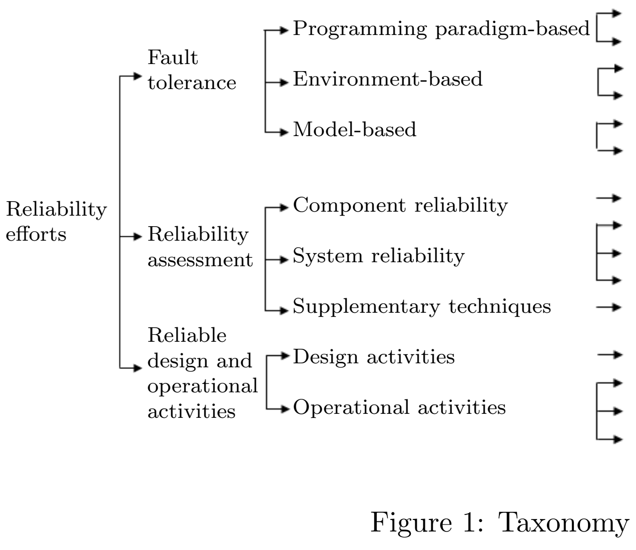
2.2 Software system reliability

Software system reliability is the probability that a system performs its intended function for a specified period of time under a set of specified environmental conditions [52, 74]. Faults, errors, and failures are the main impairments to software reliability. A fault is the hypothesized cause of an error. An error is a part of the system state which is liable to lead to subsequent failure. A failure occurs when the delivered service does not comply with the specification. Software failures are classified from the failure domain viewpoint into content, silent, early service delivery, performance, halt, and erratic failures.

Reliability and dependability are sometimes used interchangeably [14, 49], while most of the researchers consider dependability as the combination of reliability and security. However, reliability and security intersect in some of their sub-attributes (integrity and availability). Usually, design and operational faults are the main concerns in software reliability, while malicious faults are the main concerns in software security. In this survey, we focus on design and operational faults as they are the main threats to system reliability.

In general, the means to achieve software reliability are fault avoidance (or prevention), fault removal, fault/failure forecasting, and fault-tolerance. Fault avoidance or prevention techniques are employed in software development to reduce the number of faults introduced during construction. Fault removal techniques are employed during software verification and validation for detecting existing faults and eliminating them. Fault (or failure) forecasting estimates the presence of faults and their failure consequences during the design, validation, and operational phases. Fault-tolerance provides services complying with the specification in spite of design and operational faults by utilizing some kinds of redundancy. In our survey, we focus on the reliability means that are exploited mainly during CBSS design and operational phases. Particularly, our survey will involve fault-tolerance, failure forecasting, and fault avoidance efforts for structured design and software reusability. The survey will not cover fault removal and avoidance mechanisms based on formal methods and rigorous specification of system requirements.

2.3 Taxonomy of reliability efforts in CBSSs



Class/subclasses	Current work
Intra-component	[18, 36, 38, 39, 40]
Inter-component	[6, 27, 45, 53, 75]
Decentralized	[3, 44, 47, 55, 56, 60, 61]
Centralized	[20, 21]
Message-based	[16, 24, 30, 57, 66, 68, 81]
Replica-based	[12, 16, 26, 28, 58, 68, 69, 76]
Component reliability	[9, 19, 41]
State-based CBSS reliability	[10, 31, 32, 35]
Path-based CBSS reliability	[48, 84]
Propagation-based CBSS reliability	[13, 64, 73]
Supplementary techniques	[31, 35]
Composition	[2, 29, 50, 54, 63, 65, 78]
Configuration	[17]
Deployment	[5]
Upgrading	[11]

Figure 1: Taxonomy of reliability efforts in CBSSs

We classify the existing reliability efforts for CBSSs into fault-tolerance [18, 36, 38], reliability assessment [9, 19, 41], and reliable design and operational activities [2, 5, 11] (see Fig. 1). Fault-tolerance is achieved by tailoring the fundamental and traditional fault-tolerance approaches to Suit CBSSs, mainly by introducing software redundancy to system architecture levels. Reliability assessment involves failure forecasting techniques that evaluate system reliability quantitatively in terms of component reliabilities. Reliable design and operational activities are fault avoidance techniques that aim to achieve the reliability of a system during the design and operational activities of CBSSs.

In our classification, we compare and contrast the existing techniques with respect to the major assumptions of component definition and specification details. We identify three main types of components based on their specification details: programming paradigm-based, environment-based, and model-based components.

A programming paradigm-based component is the most common in the current component reliability research. The programming paradigm-based component is specified based on principal programming paradigms (*e.g.*, function [48], module [36], and class [18]). The interface connections between them are specified based on the interactions of these programming paradigm elements (*e.g.*, function call). Some programming paradigm-based techniques consider highly abstract components and specify the interface connections among them as abstract in/out ports for data exchange [6, 39, 53]. The specification of programming paradigm-based components and the interface connections among them do not follow the common component definition [15, 71, 80], where, in most cases, a programming paradigm-based component is not a unit of deployment.

An environment-based component is specified based on system paradigms (*e.g.*, distributed agent [21, 60] and mobile agent [61]). The interface connections between them are specified based on the interactions of system paradigm elements (*e.g.*, communication ports). In most of the work, environment-based components are considered as black boxes and are assumed to be free from design faults. These components are highly abstract, and they can embody components that comply with the common component definition [15, 71, 80].

A model-based component is specified based on the component model (*e.g.*, DCOM [62], EJB [70], and CCM [79]). The interface connections between components are specified according to the component model (*e.g.*, facet and receptacle). In most of the work, CCM is the underlying component model, and components are considered as black boxes that are free from design faults. A model-based component is considered as a unit of composition and deployment, and it complies with the common component definition [15, 71, 80].

2.3.1 Fault-tolerant CBSSs

Fault-tolerance is one of the major reliability means in CBSS. We classify fault-tolerance efforts for CBSSs according to the component type into three main classes: programming paradigm-based [6, 18, 36, 53], environment-based [21, 46, 47, 60, 61], and model-based [23, 25, 26, 69, 76]. In each of these classes, fault-tolerance techniques address specific types of faults, use specific fault-tolerance techniques, and achieve system reliability according to the assumptions and component specification details in each class. Fault-tolerance techniques for programming paradigm-based CBSSs address mainly design faults and use mainly design diversity and exception handling techniques to achieve system reliability. Both environment-based and model-based fault-tolerance techniques address only operational faults and use various approaches, while they mostly focus on checkpointing and replication to achieve system reliability.

The programming paradigm-based fault-tolerance class includes two subclasses: intra-component [18, 36, 39] and inter-component [6, 27, 45]. Intra-component techniques tend to provide component structures that help achieve system reliability, and inter-component techniques introduce component group and system level techniques to achieve system reliability.

The environment-based fault-tolerance class includes decentralized [3, 44, 47, 55, 56, 60, 61] and centralized [20, 21] techniques. While decentralized techniques are mainly concerned about the problem of where to store the control and log data, centralized techniques do not provide any efforts to overcome this problem and rely on their underlying platform to store these data. Both centralized and decentralized techniques include strategic solutions for adapting the replication strategy to obtain higher reliability, distinguish between component criticalities, and impose these criticalities into the fault-tolerance techniques.

The model-based fault-tolerance (precisely, CORBA-component model related) techniques are classified based on their main focus into two subclasses: message-based [24, 30, 57, 66, 81] and replica-based [26, 28, 58, 69, 76]. Message-based techniques provide the communication protocol infrastructure for reliable messaging while considering object replication and contribute to replica consistency. Replica-based techniques provide replica management, implement fault tolerance mechanisms, and support object group invocation. The details of each of the above techniques are discussed in Section 3.

2.3.2 Reliability assessment of CBSSs

Recent research attempts in the reliability assessment of CBSSs including failure propagation [13, 64, 73] and various supplementary [31, 35] techniques are not considered in the previous taxonomies [31, 32, 34, 35]. We consider these supplementary techniques in our taxonomy and classify current reliability assessment techniques into three main classes: component reliability model, system reliability model, and supplementary techniques. Obviously, component and system reliability models assess the reliabilities of individual components [9, 19, 41] and CBSSs [10, 13, 32, 48, 64, 73, 84], respectively. Supplementary techniques [1, 31, 33, 35, 37, 43, 77, 82, 83] provide support for the CBSS reliability assessment, while they may not directly provide a model for reliability assessment. We further classify system reliability models into three classes: state-based, path-based, and propagation-based. State-based models use the control graph to represent software system architecture and predict reliability analytically. Path-based models compute software reliability considering the possible execution paths of the program. Propagation-based models incorporate the dependency of component failures by considering the error propagation among system components. The details of reliability assessment efforts in CBSSs are discussed in Section 4.

2.3.3 Reliable design and operational activities of CBSSs

Reliable design and operational activities of CBSSs involve the work that addresses system reliability during software composition [2] as a software design activity and other operational activities (deployment [5], configuration [17], and upgrading [11]). Component composition provides a foundation for reasoning about emergent assembly-level behaviour as well as system behaviour and their relationships to the properties of the architectural elements. Configuration management is the task of tracking and controlling changes in the software. Component deployment is the activity of making a component available for use in a system. Component upgrading refers to the replacement of a component with a newer version of that same component. The details of the reliable design and operational activities of CBSSs are discussed in Section 5.

3 Fault-tolerant CBSSs

Despite rigorous software verification and validation efforts, design faults may exist and cause system failures. Fault-tolerance is one of the most common strategies for improving reliability of CBSSs. Fault-tolerance techniques enable a system to tolerate software faults remaining in the system after its development (design faults) or introduced to the system during the operational phase (operational faults). Fault-tolerance techniques achieve software system reliability through redundancy. A fault-tolerance technique may involve four activities to avoid failures at runtime: error detection, diagnosis, isolation or containment, and recovery [74]. The following section describes the fundamental fault-tolerance techniques, and the consecutive sections discuss the fault-tolerance classes of our taxonomy shown in Fig. 1.

3.1 Fundamental software fault-tolerance techniques

Design diversity [36], exception handling [53], checkpointing [47], and replication [20] are the main fundamental fault-tolerance techniques in CBSSs. In the following paragraphs, we briefly explain the major techniques used in CBSSs.

Software design diversity: Design diversity is the provision of identical services through separate designs and implementations. The basic design diversity-based fault-tolerance techniques include recovery blocks, consensus recovery block, n-version programming, and n-self-checking programming. In these techniques, decision algorithms are used to adjudicate the system result and determine the system success or failure state.

Exception handling: Exception handling is a fault-tolerance technique that handles the occurrence of abnormal or undesired execution conditions of CBSSs. It allows software developers to define exceptional conditions and additional code (exception handlers) that deal with those conditions. When an exception is raised, the exception handler interrupts the normal control flow of the computation and searches for an appropriate exceptional control flow from a set of predefined exception handlers.

Checkpointing: Checkpointing is the process of storing snapshots (recovery points) of a system's computational state. When a failure is detected, the checkpointed states are utilized for restoring the previous execution state of the system. Similar mechanisms can be used to establish these recovery points (*e.g.*, the audit trail and the recovery cache). A checkpoint saves a complete copy of the state when a recovery point is established. Recovery cache saves only the original state of the objects whose values have changed after the latest recovery point. Audit trail records all the changes made to the process state.

Replication: Replication uses redundant exact copies of software components. These exact copies are not able to tolerate software design faults. However, replication aims to protect the application against operational faults that may cause computer crash or communication failures. The main problem with replication is how to make the software copies consistent with respect to their behaviors and computation states. There are two essential replication styles: active and passive. In active replication, each replica processes all requests from other components and returns its response back to the requester. The requester takes the first response and ignores the other responses. In passive replication, only one primary replica processes the other component requests, and returns its response to the requester. The state of the primary replica is periodically synchronized directly to the other replicas (warm passive replication) or periodically checkpointed to a separate data store (cold passive replication).

3.2 Programming paradigm-based fault-tolerance

In programming paradigm-based fault-tolerance, a component is viewed as a unit of composition that interacts with other components through interface connections. The details about these interface connections consider only the principal programming paradigm and disregard the architectural style of the software system. Based on this view, fault-tolerance techniques for this class achieve reliability by re-fabricating existing fault-tolerance techniques to Suit the CBSS architectures. For example, design diversity techniques are used by applying diversity to the component level rather than the system level [36]. Exception handling techniques are used at the component level rather than the language construct level [40]. Architectural level fault-tolerance techniques aim to tolerate design faults associated with component and connector failures, architectural mismatches, and configuration faults [53]. Design diversity [6, 18, 36] and exception handling [27, 38, 39, 40, 53, 75] are the most widely used techniques due to their ability to tolerate design faults.

The programming paradigm-based fault-tolerance includes two subclasses: intra-component [18, 36, 38, 39, 40] and inter-component [6, 27, 45, 53, 75]. Intra-component techniques tend to provide component structures that help achieve system reliability, and inter-component techniques introduce component group and system level techniques to achieve system reliability. In the following subsections, we discuss the details of each subclass.

3.2.1 Intra-component fault-tolerance

Intra-component techniques provide component structures [18, 36, 39, 40] or wrappers [38] that help achieve system reliability. These fault-tolerant component structures are considered as new building blocks for CBSSs. Therefore, the proposed component structures inherit the basic characteristics of components, particularly, composition and independence. Executive component [18] is an example of these structures in which reusable object oriented components encapsulate some fundamental design diversity techniques. The basic n-version programming design diversity technique is used to structure the fault-tolerant component [36] for constructing multi-modular programming software systems. The idealized fault-tolerant architectural component [39, 40] is structured using $C2$ architectural style[2] to allow component exceptions propagation through the software system architecture. The idealialized fault-tolerant component is structured into two parts: a normal behavior part, that is, the main functional implementation, and an abnormal behavior part, that handles the exception conditions.

Component wrapper is another approach for applying intra-component fault-tolerance. A wrapper is a specialized component that is inserted between a component and its environment to deal with the ingoing and outgoing data errors of the wrapped component. Component wrappers are used with black box components (*e.g.*, Commercial Off The Shelf (COTS) and legacy components) to provide a protective layer in a specific operational context. By extending the idealized fault-tolerant architectural component [39] using component wrapper, Guerra *et al.* [38] present an idealized fault-tolerant COTS component.

[2] "$C2$ architectural style is a component and message-based style for constructing software systems based on flexible composition" [39].

	Intra-component approach	Inter-component approach	Object oriented programming	Modular programming	Design faults	Operational faults	Design diversity	Exception handling	Component handlers	Connector handlers	Component group handlers	Configuration handlers	Wrapper	Structured component
Duncan *et al.* [18]	✓		✓		✓		✓							✓
Grosspietsch *et al.* [36]	✓			✓	✓		✓							✓
Guerra *et al.* [39]	✓		✓		✓	✓		✓	✓	✓				✓
Guerra *et al.* [40]	✓		✓		✓	✓		✓	✓	✓				✓
Guerra *et al.* [38]	✓					✓		✓					✓	
Blackmon *et al.* [6]		✓			✓		✓					✓		
Filho *et al.* [27]		✓	✓			✓		✓				✓		✓
Issarny *et al.* [45]		✓	✓		✓	✓		✓	✓	✓		✓		✓
Lemos *et al.* [53]		✓	✓		✓			✓	✓		✓		✓	
Romanovsky *et al.* [75]		✓			✓			✓	✓		✓	✓	✓	

Table 1: Summary of programming paradigm-based fault-tolerance.

Wrappers are employed to protect the system against the erroneous behavior of the COTS components. They also protect COTS components against the erroneous requests from the rest of the system.

3.2.2 Inter-component fault-tolerance

Inter-component fault-tolerance techniques focus on fault-tolerant component integration rather than construction. Fault-tolerance support is embedded at the system level [6, 27, 45] or at a component group level [53, 75]. Exception handling at the system level provides fault-tolerance for system configuration components. In addition, it enables changing the system's running configuration according to the occurrence of configuration exceptions to prevent further failures. Exception handling at the component group level assumes that, if components collaborate to deliver a specified service, they should also collaborate to handle particular failure scenarios based on the specified service [45].

Blackmon *et al.* [6] apply component variants (design diversity) based on the different component criticalities. These criticalities are assessed through component interface complexity or granularity. In their multi-level exception handlers, Filho *et al.* [27] and Issarny *et al.* [45] attach exception handlers to the system configuration components and use the exception propagation to notify these configuration handlers. Exception flow in a software system architecture helps to enable changing the system's running configuration according to these unresolved exceptions to prevent further error exceptions. Other exception handlers are attached as wrappers for a group of components that collaborate in any specific operation [53] or atomic action [75].

3.2.3 Comparison of programming paradigm-based fault-tolerance techniques

Table 1 summarizes the existing programming paradigm-based fault-tolerance techniques and shows the subclasses (intra-component and inter-component), programming paradigms (object oriented and modular), fault types (design and operational), fault tolerance mechanisms (design diversity and exception handling), types of exception handlers (component, connector, component group, and configuration level), and other mechanisms (wrapper and structured component) of these techniques.

Most programming paradigm-based CBSSs consider component as object [18, 27, 39, 40, 45, 53], module [36], or highly abstract component [6, 38, 75]. Component design faults [6, 18, 27, 36, 39, 40, 45, 53, 75] are tolerated through design diversity [6, 18, 36] or through exception handlers [38, 39, 45, 43, 75] at the component level. Component level design diversity can help to tolerate design faults within components. Design diversity in traditional software systems is applied at the system level, while in CBSSs, it is applied at the component level. Applying design diversity at the component level allows earlier detection and recovery of component errors before they become system failures. However, design diversity is expensive and real diversity is hard to obtain.

In CBSSs, exception handling is applied at different architectural levels from component [39, 40] to system configuration levels [6, 27, 45, 75]. Traditional exception handling at the language construct level strengthens the ability of detecting and recovering execution errors inside component code, while exception handling at the component or at higher levels can handle these errors only after the component produces an output. Therefore, by the time an error is detected at the component level, handling that error may be too complicated and hard to achieve. While exception handling at the system architecture level may gain benefits of masking failures that are caused by interface design faults or operational faults, it may loose the advantage of early detecting errors based on component design faults. Therefore, architectural level exception handling is not a sufficient solution for achieving system reliability, and is considered as a complementary solution for exception handling at the language construct level (*i.e.*, inside component code).

Exception handling is also used at the connector level [39, 40, 45] to tolerate operational faults that may cause communication and message delivery errors. Exception handling is also applied at the component group level [53, 75] and at the system configuration level [6, 27, 45, 75]. This multi-level exception handlers aim to detect and handle system errors at different architectural levels. Exception handling at component group levels assumes collaborative handling of more complicated failure scenarios. This group level exception handling can be considered as an extra failure protection in case individual components are not able to mask their failures. Configuration level exception handling is considered as a solution that provides adaptive strategy to adjust system running configuration where all handlers fail to stop exception propagation.

Component wrappers [38, 75] are applied at the individual component level [38] and at the component group level [75] to provide extra protective layer for components in specified contexts. These wrappers can be an effective solution for COTS and legacy components where heterogeneity is assumed. However, wrappers may introduce additional complexity when they are applied simultaneously with component and connector level exception handling. Most of the intra-component techniques provide component structures [18, 27, 36, 38, 39, 45] that can be building blocks for fault-tolerant CBSSs. However, applying these component structures may require further tailoring to comply with any specific computational environment or architectural style.

3.3 Environment-based fault-tolerance

The environment-based fault-tolerance techniques consider components as black boxes and disregard their software design faults, while they focus mainly on the operational faults that may cause crash and communication failures. To avoid these failures, fault-tolerant multi-agent systems employ replication or exception handling strategies. In case of a host crash or communication loss, other replicas on different hosts may take control, or an exception can be raised to avoid a system failure. Fault-tolerance strategies are applied based on whether a system architecture is decentralized (*i.e.*, ad-hoc components) or centralized (*i.e.*, platform-based). We classify environment-based fault-tolerance techniques based on their underlying architectures into decentralized [3, 44, 47, 55, 56, 60, 61] and centralized [20, 21] techniques as described in the following subsections.

3.3.1 Decentralized fault-tolerance

In decentralized fault-tolerance, a system construction depends on decentralized architectures and uses ad-hoc communication schemes. Decentralized fault-tolerance aims to facilitate the fault-tolerance strategy in a decentralized manner. Marin *et al.* [60] introduce a framework called DARX, for building fault-tolerant decentralized multi-agent software applications using an adaptive replication strategy based on agent criticality. Almeida *et al.* [55, 56] extend DARX [60] by introducing an adaptive and predictive replication policy for critical agents based on their plan criticalities. Instead of using replication, Aliasov *et al.* [3, 44] use exception handling to support system error recovery. Meng *et al.* [61] provide pragmatic framework for agent execution to support system fault-tolerance by deploying independent checkpointing with passive replication strategy. Khokhar *et al.* [47] use a concept of antecedence graphs for checkpointing the agent communication messages.

3.3.2 Centralized fault-tolerance

Centralized techniques rely on the underlying platform to store the replication data based on different criticality measures for system components. Faci *et al.* [21] extend DARX [60] for building a fault-tolerant platform (called DimaX) for centralized multi-agent architectures using adaptive replication strategy based on agent role criticality. Gatti *et al.* [20] extend DimaX [21] by improving the capability of critically calculation using an enhanced XMLaw scheme. The XMLaw scheme implements a mechanism for agent interaction called law enforcement as an object oriented framework.

3.3.3 Comparison of environment-based fault-tolerance techniques

In distributed multi-agent systems [3, 20, 21, 44, 47, 55, 56, 60, 61], host crash and communication failures are major threats to system reliability. Mobile-agent systems [44, 61] are also prone to these failures during their migration process. To mask these failures, fault-tolerant multi-agent systems employ replication [20, 21, 47, 55, 56, 60, 61] or exception handling [3, 44] techniques. Table 2 summarizes the existing environment-based fault-tolerance techniques and shows the subclasses (decentralized and centralized), system paradigms (distributed agents and mobile agents), fault tolerance mechanisms (replication and exception handling), storage location of the replicated data (component, middleware, and platform), and other parameters (agent criticality and supportive activities) of these techniques.

Both replication and exception handling techniques are able to tolerate operational faults. However, replication strategy has a number of advantages. Replica distribution among different hosts avoids a host from being a single point of failure. It also helps the load balancing among the available system resources. Replication allows minimizing the resource utilization of multi-agent systems using passive replication. Minimizing the resource utilization includes decreasing the workload and consequently, it helps decrease crash failures. However, by disregarding active replication, a system may be prone to more performance failures. Furthermore, by using a replication strategy, a fault-tolerant multi-agent

	Decentralized techniques	Centralized techniques	Distributed agents	Mobile agents	Replication	Exception handling	Component datastore	Middleware datastore	Platform datastore	Agent criticality	Supportive activities
Marin *et al.* [60]	✓		✓		✓			✓		✓	
Almeida *et al.* [55]	✓		✓		✓			✓		✓	
Almeida *et al.* [56]	✓		✓		✓			✓		✓	
Batista *et al.* [3]	✓		✓			✓					
Iliasov *et al.* [44]	✓		✓	✓		✓					
Meng *et al.* [61]	✓		✓	✓	✓		✓				✓
Khokhar *et al.* [47]	✓		✓		✓		✓				✓
Faci *et al.* [21]		✓	✓		✓				✓	✓	
Gatti *et al.* [20]		✓	✓		✓				✓	✓	

Table 2: Summary of environment-based fault-tolerance.

system disregards content failures. Exception handling techniques provide architectural traps [3, 44] for agent failures in general. Therefore, these techniques can also tolerate some content failures based on the operational faults of agent communications.

Storing replication data is one of the main concerns of a replication strategy. Most of the decentralized techniques store component information at other system components [47, 61] or at middleware components [3, 44], while other decentralized techniques adopt an exception handling strategy at the component level to avoid the decentralization problem. In centralized techniques replication data are stored at the underlying platform [20, 21]. Consequently, centralized techniques do not require additional effort to create replicated data stores, while they only focus on managing these data efficiently.

Most of the environment-based fault-tolerance techniques choose the replication style and management based on component criticality measures [20, 21, 55, 56, 60]. Criticality measures are computed based on component role [20, 60] or component plan [55, 56]. Other techniques conduct supportive or supplementary activities [47, 61] for the replication strategy. Examples of these activities are execution management [61] and checkpointing methodology [47].

With respect to software failure, environment-based fault-tolerance techniques can mask only silent failures, while they cannot mask content failures or performance failures. Moreover, by considering agents as black boxes, software failures caused by design faults of these agents are not considered in the existing fault-tolerance techniques.

3.4 Model-based fault-tolerance

The model-based fault-tolerance represents fault-tolerance techniques that include component models and frameworks. The model-based component is specified based on component model (*e.g.*, DCOM [62], EJB [70], and CCM [79]). The interface connections between them are specified in details according to the component model (*e.g.*, CCM facet and receptacle ports). In most of the work, CCM is the underlying component model and components are considered black boxes that are free from design faults. These components comply with the widely accepted component definition [15, 71, 80].

In general, model-based component specification identifies a number of concepts such as architecture, configurations, connectors, bindings, properties, hierarchical models, style, and behavior. Component models are also supported by frameworks that provide design time and run time infrastructure for CBSSs [16, 69]. In addition to these concepts, CCM provides transparent distribution[3] of the application objects [72]. The CORBA component model works above the Object Request Broker (ORB)[4] and thus allows building distributed and heterogeneous software applications. CCM provides language and operating system independent features, while other component models are usually language (EJB) or platform (DCOM) dependent [28]. Because of these advantages, almost all existing reliability efforts for component models are tailored for CCM.

Natarajan *et al.* [69] and Felber *et al.* [25] provide taxonomies on CORBA fault-tolerance techniques by classifying them based on their relative location to the ORB into integration [57, 81], interception [16, 68], and service [16, 22, 30, 58, 66, 69] approaches. An integration approach incorporates replication into the core ORB. The interception approach uses interception objects underneath the ORB, *i.e.*, between the ORB and the operating system. The service approach provides fault-tolerance support primarily through collections of objects above the ORB, *i.e.*, between the ORB and the CORBA application. However, their taxonomy classifies fault-tolerance techniques based on the CORBA anatomy and mainly explains the exploited parts of CORBA architecture and the impacts on CORBA interoperability[5], transparency[6], ORB compliance, among other infrastructure related factors. The taxonomy does not differentiate among fault-tolerance techniques from reliability viewpoint and does not discuss their sufficiency for reliable CORBA applications.

We classify model-based techniques based on their strategic architectural element into two subclasses: message-based [24, 30, 57, 66, 81] and replica-based [26, 28, 58, 69, 76]. Message-based techniques provide the communication protocol infrastructure for reliable messaging while considering object replication. Message-based techniques support fault-tolerance by achieving replica consistency. Replica-based techniques provide replica management and implement fault tolerance mechanisms. Replica-based techniques support fault-tolerance by achieving object group invocation. Some work extends the two subclasses to provide integrated fault-tolerance solutions for model-based CBSSs [16, 68]. In the following sections, we explain the details of these subclasses.

3.4.1 Message-based fault-tolerance

The underlying strategy for message-based techniques is based on a group communication mechanism that assumes that replicas are distributed among the system architecture and extends the object communication mechanism provided by the ORB to handle groups of replicas. These techniques are also concerned about state replication and aim to perform the replication in a reliable way. The main goals of these techniques are to achieve replica consistency and allow group invocation through message multicasting. Maestro [81] is an integration approach that supports replica consistency by providing object group communication tools and client/object interoperability tools while assuring reliable and safe message

[3]Transparent distribution implies that application services can invoke server objects without additional efforts regarding the actual locations of the server objects [68].

[4]Object Request Broker (ORB) provides communication infrastructure for distributed and heterogeneous CORBA applications.

[5]Interoperability of CORBA ensures that the system can be used by the clients and servers that are running on the ORBs from different vendors [23].

[6]In CCM, transparency dictates the hiding of the use of replication from the application programmer by viewing a group of server replicas as a single object.

delivery through a layered implementation of state machine replication. Electra [57] is an integration approach that consists of a modified ORB. The ORB maintains replica consistency and invocation using a reliable group communication mechanism by converting the application's/ORB's messages into multicast messages. Object Group Service (OGS) [24] is a group communication framework that provides support for replica consistency and group invocation by enabling the view of a group of CORBA objects as a single entity despite concurrent invocations or replica failures. Newtop [66] is an object group management framework. However, it differs from OGS [24] in that it allows objects to belong to multiple object groups. FTS [30] provides a lightweight CORBA service for fault-tolerance by combining interception and service approaches using an extended object adapter implemented on top of standard Portable Object Adapter (POA). Aquarius ([12]) provides a data-centric approach for object replication and invocation using a shared server object that can be manipulated by clients while hiding object replicas at independent servers.

3.4.2 Replica-based fault-tolerance

Replica-based techniques focus on the replica management duties (creation and destruction), fault management (error detection and recovery mechanisms), and property management (replica distribution and system configuration) to achieve reliable group invocation. Proteus [76] provides fault-tolerance in AQuA [16] by dynamically managing the replicated distributed objects and configuring the system in response to faults and changes in desired dependability levels. Fraga *et al.* [28] present an adaptive fault-tolerant component model (Afault-tolerant CCM) based on CORBA by introducing a set of components that monitors the failure occurrences in the application. The Distributed Object-Oriented Reliable Service (DOORS) [69] is a part of fault-tolerant CORBA standard that exploits CORBA service approach to provide replication management, failure detection and recovery as service objects above the ORB. The Interoperable Replication Logic (IRL) [58] exploits the service approach by providing a set of protocols, mechanisms, and services that offer interoperable ORB compliant architecture and allow a CORBA system to handle object replication. Lightweight fault-tolerance [26] aims to avoid unnecessary complexities of the standard fault-tolerant CORBA for applications that do not require strong data consistency.

Some work [16, 68] integrates message-based and replica-based fault tolerance techniques. Eternal, [68] is a part of fault-tolerant CORBA standard that implements a number of techniques as ORB portable interceptors to provide replication management and strong replica consistency without requiring modifications to the application or to the ORB. AQuA [16] integrates the dependability management of Proteus [76] with Maestro [81] as the underlying group communication system that provides reliable multicast, total ordering, and virtual synchrony.

3.4.3 Comparison of model-based fault-tolerance techniques

The model-based fault tolerance is mainly concerned about two objectives: achieving replica consistency [30, 58] and allowing group invocation [26, 28, 69, 76] or both [12, 16, 24, 57, 66, 68, 81]. The majority of message-based [16, 24, 30, 57, 66, 68, 81] techniques aim to achieve replica consistency, while the majority of replica-based [12, 16, 26, 28, 58, 68, 69, 76] techniques aim to allow group invocation. Table 3 summarizes the existing model-based fault-tolerance techniques and shows the subclasses (message-based and replica-based), previous taxonomies [25, 69] (integration, interception, and Service), work objectives (consistency and invocation), CCM reliability means (reliable messaging, state replication, replica management, replica distribution, error detection, and error recovery), and other features (adaptive, light

	Message-based	Replica-based	Integration aproach	Interception aproach	Service aproach	Consistency	Invocation	Reliable messaging	State replication	Replica management	Replica distribution	Error detection	Error recovery	Adaptive	Light weight	Content failure
Vaysburd *et al.* [81] (Maestro)	✓		✓			✓	✓	✓	✓							
Maffeis *et al.* [57] (Electra)	✓		✓			✓	✓	✓	✓							
Felber *et al.* [24] (OGS)	✓				✓	✓	✓	✓	✓			✓	✓			
Morgan *et al.* [66] (Newtop)	✓				✓	✓	✓			✓		✓				
Friedman *et al.* [30] (FTS)	✓				✓	✓		✓		✓		✓	✓		✓	
Chockler *et al.* [12] (Aquerius)		✓			✓	✓	✓		✓	✓	✓					
Sabnis *et al.* [76] (Proteus)		✓			✓		✓			✓	✓	✓	✓	✓		✓
Fraga *et al.* [28]		✓			✓		✓					✓		✓		
Natarajan *et al.* [69] (Doors)		✓			✓		✓			✓		✓	✓	✓		
Marchetti *et al.* [58] (IRL)		✓			✓	✓				✓	✓	✓	✓			
Felber *et al.* [26]		✓			✓		✓			✓			✓		✓	
Narasimhamn *et al.* [68] (Eternal)	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
Ren *et al.* [16] (Aqua)	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓

Table 3: Summary of model-based fault-tolerance

weight, and consideration of content failures) of these techniques.

The means to achieve a fault-tolerant CORBA application are reliable messaging, state replication, replication management, replica distribution, error detection and notification, and error recovery. In general, the first two means are achieved by the message-based techniques, while the last four are addressed by the replica-based techniques. Message-based techniques provide an important infrastructure for reliable messaging and contribute to replica consistency. Replica-based techniques implement the fault-tolerance technique and support group invocation. Both message-based and replica-based techniques are required for a complete fault-tolerance solution [16, 68].

Reliable messaging [16, 24, 30, 57, 68, 81] includes checking for duplicate messages, dropped messages, and total message ordering for system objects. State replication [12, 16, 24, 57, 68, 81] involves multicasting object messages into the group of replicas corresponding to such object. Replication management [12, 16, 26, 30, 58, 66, 68, 69, 76] is concerned mainly with creating and deleting replicas, issuing the Interoperable Object References (IORs), controlling the replication style and other settings. Replica distribution [12, 16, 58, 68, 76] is concerned about the location of object replicas. Error detection [16, 24, 28, 30, 58, 66, 68, 69, 76] is mainly achieved by using heartbeat messages or time out mechanisms. Error notification is usually considered as part of the error detection mechanism that provides information to the error recovery [16, 24, 26, 30, 58, 68, 69, 76] and replica management to respond for the detected error. Depending on the replication style (active or passive), error recovery either selects another replica to provide a response or it performs voting among the active replicas to select the correct response.

Some model-based techniques incorporate adaptive fault tolerance strategies [16, 28, 68, 69, 76], in which they dynamically manage replicated objects and configure the system in response to faults and changes for desired dependability levels. Management and configuration choices include the number and type of faults an application intends to tolerate (crash, value, and time), the performance expected, the styles of replication, the types of voting, the degrees of replication to use, and the location of the replicas, among other factors. Lightweight fault-tolerance [26, 30] in CORBA avoids unnecessary complexities of the standard fault-tolerant CORBA for applications that do not require strong data consistency.

The model-based fault-tolerance techniques focus mainly on masking crash and communication loss failures. Some model-based fault-tolerance techniques tolerate content failures [16, 68, 76] by using a voting algorithm to adjudicate the result among a number of replica outputs (messages). However, voting among these replicas cannot mask content failures that are based on component design faults. By disregarding content failures based on design faults, fault-tolerant CORBA assumes that the system components are bugs free.

3.5 Comparison of fault-tolerance classes of CBSSs

Table 4 summarizes the existing fault-tolerance classes of CBSSs and shows the fault tolerance mechanisms (design diversity, exception handling, replication, and checkpointing), fault types (design and operational faults), failure types (silent, performance, and content), major software architectural styles (distributed, client/server, event driven, and peer-to-peer), and the other existing work of these classes.

Different classes of fault-tolerance techniques for CBSSs adopt strategies and goals according to component viewpoints and specification details. Design diversity [6, 18, 36] and exception handling [38, 39, 45, 43, 75] are used in the programming paradigm-based techniques to tolerate design faults, while few works in this class address operational faults using component wrapper approaches [38, 75]. The environment-based and model-based techniques depend on checkpointing and replication [20, 21, 47, 55, 56, 60, 61] as main strategies for their sufficiency for tolerating operational faults based on the assumption that the components are design fault free. The reason for addressing mainly design faults at the programming paradigm-based techniques is the unavailability of the knowledge about operational environment. The reason for addressing only operational faults at the environment-based and model-based techniques is the consideration of components as fault free.

By tolerating component design faults, programming paradigm-based techniques are able to tolerate all types of component failures. The environment-based and model-based techniques tolerate operational faults by masking communication failures such as message corruption and link disconnection, and crash failures such as computer and process crash. However, errors that may cause the aforementioned failures are detected when a component stops responding and appears silent (silent failures). Therefore, environment-based and model-based techniques can tolerate only silent failures that are caused by operational faults, while they tolerate neither content failures nor performance failures.

Few environment-based techniques address content failures by using exception handling [3, 44]. However, by using the component wrapper approach, they only focus on component interfaces, *i.e.*, they only tolerate content failures that are caused by operational faults or interface design faults. In other words, these techniques also disregard content failures that are caused by component design faults.

In model-based techniques, inconsistent replicas may produce different message contents in response to a service request. Content failures are also addressed in these techniques by using voting algorithms among the inconsistent groups of replicas [16, 68, 76]. However, the causes of these content failures are the operational faults that lead to the replica group inconsistency. Therefore, these techniques also disregard content failures based on design faults.

Among the CBSS architectural styles, distributed architectures [28, 30, 57, 58, 66, 68, 69, 76, 81] are the most commonly used for the multi-agent and mobile agent system paradigms of the environment-based techniques, and for the CCM of the model-based techniques [12, 16, 24, 26]. Client/server [28]

	Design diversity	Exception handling	Replication	Checkpointing	Design faults	Operational faults	Silent failures	Performance failures	Content failures	Distributed	Client/Server	Event driven	peer-to-peer	Existing work
Programming paradigm-based														
Intra-component techniques	✓	✓			✓	✓	✓	✓	✓					[18, 36, 38, 39, 40]
Inter-component techniques		✓			✓		✓	✓	✓				✓	[6, 27, 45, 53, 75]
Environment-based														
Decentralized techniques		✓	✓	✓		✓	✓		✓	✓				[3, 44, 47, 55, 56, 60, 61]
Centralized techniques			✓	✓		✓	✓			✓				[20, 21]
Model-based														
Message-based techniques			✓	✓		✓	✓			✓	✓	✓		[16, 24, 30, 57, 66, 68, 81]
Replica-based techniques			✓	✓		✓	✓		✓	✓	✓	✓		[12, 16, 26, 28, 58, 68, 69, 76]

Table 4: Summary of fault-tolerance in CBSSs

and event-driven [16] architectures are considered as native environments for CCM, and very few works address the peer-to-peer [53] architectures in the environment-based techniques class.

4 CBSS reliability assessment

Reliability assessment involves failure forecasting techniques that evaluate system reliability quantitatively in terms of component reliabilities. These techniques can be used at the operational phase and at the early design stages of software systems. Applying these techniques in the early design stages supports the appropriate design decisions that affect the artifacts of the later development stages. Reliability assessment based on system architectures has a number of advantages over the prevalent software reliability assessment which is black box-based, *i.e.*, a software system is treated as a whole, and only its interactions with outside world are considered [31]. Some example advantages [32] are the ability to compare alternative architectures with respect to their reliabilities, the ability to relate system reliability to its structure and individual component reliabilities, and the ability to assess the reliability of operational systems to identify components which provide maximum potential for reliability improvement.

Few attempts to classify CBSS reliability models have been proposed [31, 32, 34, 35] and broadly accepted. However, it is argued that, "without exception, all the existing models assume that failure processes associated across different components are mutually independent" [34]. Recently, some work considers the dependence of component failures [13, 64, 73] and some others provide supplementary techniques to indirectly support CBSSs reliability models. In addition to the existing reliability assessment techniques, we also consider these recent research attempts in our taxonomy.

We classify reliability assessment techniques into three main classes: component reliability model, system reliability model, and supplementary techniques. Obviously, component and system reliability models assess the reliabilities of individual components [9, 19, 41] and CBSSs [10, 13, 32, 48, 64, 73, 84], respectively. Supplementary techniques [1, 31, 33, 35, 37, 43, 77, 82, 83] provide support for CBSS reliability assessment, while they may not directly provide a model for reliability assessment. We further classify system reliability models into three classes: state-based, path-based, and propagation-based.

State-based models use control graphs to represent software system architecture and predict reliability analytically. Path-based models compute software reliability considering the possible execution paths of the program. Propagation-based models incorporate the dependency of component failures by considering the error propagation among system components. In the following subsections, we provide the details about each class of CBSS reliability assessment.

4.1 Component reliability models

Component reliability models exploit design and/or operational information of an individual component to evaluate its reliability. If a component is developed as part of a system, then both design and operational information are available and can be used for evaluating the component reliability [19]. However, with respect to COTS components, design information may not be available for the user (or customer) and operational information of the integrated system and the deployment environment at the user site is not available for the component vendor. In this case, the component can be evaluated as a black box at the user site [41] or without the operational profile at the vendor site [9, 41]. Everett [19] estimates individual component reliability by using an Extended Execution Time (EET) model that walks through a stepwise procedure for performing the reliability analysis. Cheung *et al.* [9] provides a stepwise framework for reliability prediction of components by identifying important parameters of the reliability modeling process and studying their effects on component reliability. Hamlet *et al.* [41] provide individual component reliability prediction by introducing a mapping from operational profiles to reliability parameters and relating operational profiles to component functional sub-domains.

4.2 System reliability models

System reliability models aim to aggregate component failure behaviors to reason about the whole system failure behavior. These models exploit information about software system architecture and individual components. The information includes component failure behavior, control flow, transition probabilities between components, component usage stress, and component failure dependencies. Based on the model view of software system architecture and architectural dynamics, system reliability models are classified into three types: state-based, path-based, and propagation-based.

4.2.1 State-based models

State-based models for CBSS reliability represent a system architecture using control flow graph and assume that the transfer of control (*transition*) follows the Markov property[7]. One of the main assumptions of these models is that the summation of transition probabilities from a component to all other component equals to 1. The architecture of a software system has been modeled with a Discrete Time Markov Chain (DTMC) or a Continuous Time Markov Chain (CTMC) [31]. Cheung [10] provide a DTMC model for multi-modular software reliability based on the reliability of the components and the probabilistic distribution of the utilization of the components. Gokhale and Lyu [32] develop simulation procedures to assess the impact of individual components on the reliability of the whole system in the presence of fault detection and repair strategies that may be employed during testing.

[7]Markov property assumes that, given the present control state, future transition states are independent of the past transition states.

4.2.2 Path-based models

Path-based models compute system reliability by considering the possible execution paths. An execution path is a sequence of components executed consecutively based on a test input. Krishnamurthy and Mathur [48] provide a path-based method for estimating software system reliability using reliabilities of its components and the sequences of components executed during system run. Yacoub, Cukic, and Ammar [84] introduce a Scenario-Based Reliability Analysis (SBRA) for CBSS architectures based on execution scenarios, *i.e.*, sets of component interactions triggered by specific input stimulus.

4.2.3 Propagation-based models

Propagation-based models aim to incorporate component failure dependency in the reliability assessment of CBSSs. In propagation-based models, a component failure may impact other components and does not always lead to a system failure. Cortellessa and Grassi [13] analyze the reliability of a CBSS based on error propagation probability by exploiting information about the component reliabilities and the transition probability among the error propagation paths of each component. Mohamed and Zulkernine [64] evaluate system reliability in terms of failure propagation between components by introducing the concept of Architectural Service Routes (ASRs) and incorporating architectural attributes based on these ASRs into the reliability analysis of CBSSs. Popic *et al.* [73] evaluate software system reliability by viewing interface connectors between components as a set of connector elements and deriving the error propagation probabilities between components as the probabilities of any changes among connector elements.

4.3 Supplementary work for reliability assessment

The majority of the recent work for CBSS reliability assessment provides supplementary support for reliability assessment, while they do not propose new reliability models. Some of the goals of these supplementary techniques include error propagation analysis based on fault injection [43, 82, 83] or using stochastic approaches [1]. The goals also include importing other quality attribute models to assess the reliability and vice versa [37, 77]. Some work provides survey [34], framework unification [31], and empirical case study [32, 35] on existing reliability models.

4.4 Comparison of reliability assessment techniques

Table 5 summarizes the existing reliability assessment techniques and shows the main classes (component reliability assessment, state-based, path-based, propagation-based, and supplementary techniques), software development phases (design, testing, and operation), failure behaviours (component failure probability, constant failure rate, and reliability growth model), major parameters (flow control and component usage stress), and other features (component failure dependency and architectural loops) of these techniques. Based on the model parameters, a component or a system reliability model is used either during the early design [9, 41, 64, 73], testing [19, 31, 32, 48], or operational phases [13, 31, 35, 41, 48, 84].

Component failure behavior is used to incorporate the impact of individual components on the whole system reliability. This failure behaviour can be specified by using failure probability [10, 13, 31, 48, 46, 84], constant failure rate [31], or reliability growth model [19, 31, 32]. A component failure probability is

	Component reliability assessment	State-based approach	Path-based approach	Propagation-based approach	Supplementary approach	Design phase	Testing phase	Operational phase	Component failure probability	Constant failure rate	Reliability growth model	Flow control	Component usage stress	Component failure dependency	Architectural loops
Everett [19]	✓						✓				✓		✓		
Leslie Cheung *et al.* [9]	✓					✓									
Hamlet *et al.* [41]	✓					✓		✓							
Cheung [10]		✓							✓			✓	✓		
Gokhale and Lyu [32]		✓					✓				✓		✓		
Krishnamurthy and Mathur [48]			✓				✓	✓	✓			✓			✓
Yacoub, Cukic, and Ammar [84]			✓					✓	✓			✓			
Cortellessa and Grassi [13]				✓				✓	✓			✓		✓	
Mohamed and Zulkernine [64]				✓		✓			✓					✓	
Popic *et al.* [73]				✓		✓							✓	✓	
Gokhale and Trivedi [31]					✓		✓	✓	✓	✓	✓	✓			
Goseva *et al.* [35]					✓			✓							

Table 5: Summary of reliability assessment for CBSSs.

the probability of its failing during a single execution. The use of component failure probability in system reliability is based on the assumption that the knowledge about component reliabilities is available. However, most of system reliability models ignore how to determine these component reliabilities. Constant failure rate is the frequency of component failures. Reliability growth model (or failure intensity) is the change in the frequency of component failures over time. Reliability growth models allow assessing system reliability during the testing phase.

Besides component failure behavior, control flow [10, 13, 31, 48, 84] and usage stress [10, 19, 32, 73] are major parameters for most of the reliability models, since they incorporate the dynamic behavior of component interactions in system reliability. Flow control is expressed by transition graphs and transition probabilities among components. Transition probabilities are considered based on sequential processing, where the control leaves a component to only one other component. Usage stress is the average amount of execution time a component takes per visit multiplied by the number of visits in a single system run. Most state-based and path-based techniques consider flow control and usage stress in their computation, while most of the propagation-based techniques disregard these parameters.

With respect to component failure dependency, both state-based and path-based reliability models assume that components fail independently, while propagation-based models [10, 64, 73] incorporate component failure dependency in the reliability measure through the error propagation analysis. By disregarding component failure dependency, state-based and path-based techniques assume that a component failure does not affect any other component and it directly causes a system failure.

Most of the existing system reliability models consider only simple architectures with no loops[8].

[8]Architectural loops exist when a component is dependent on itself or mutually dependent on another component, directly, or through other components.

Among the works listed in Table 5, only one model [48] addresses this problem. However, the model considers only component self dependency and disregards architectural loops based on a number of components.

Most reliability assessment techniques consider programming paradigm-based components. A small number of works consider model-based components in the reliability assessment techniques. For example, Cecchet *et al.* [8] evaluate the performance and scalability of EJB applications. Marsden *et al.* [59] evaluate CORBA system dependability by using a fault injection mechanism. Yi-Min *et al.* [85] discuss the reliability and availability issues in DCOM. However, these techniques involve different model parameters and quality attributes and consequently, they are different from the majority of the techniques surveyed in this category.

5 Reliable design and operational activities of CBSSs

Reliable design and operational activities focus on the techniques that improve the reliability of the system during the design and operational phases of CBSSs. We focus on composition [2] as a design activity, and deployment [5], configuration [17], and upgrading [11] as operational activities. Component composition provides a foundation for reasoning about emergent assembly-level behaviour as well as system behaviour and their relationships to the properties of the architectural elements. Configuration management is the task of tracking and controlling changes in software components. Component deployment is the activity of making a component available for use in a system. Component upgrading refers to the replacement of a component with a newer version of that same component.

Many research works address the component activities and their issues in general while disregarding the system quality and in particular system reliability. Only a few works consider the reliability of the system during these activities and most of those works consider composition activity. We classify the work in this category based on the CBSS activities into reliable design [2, 29, 50, 54, 65, 78] and operation [5, 11, 17]. The details of these classes are explained in the following subsections.

5.1 Reliability in the design activities of CBSSs

A number of composition approaches consider system reliability as a main design goal. Consequently, they provide reasoning about assembly-level and system reliabilities and their relationships to the properties of the architectural elements. The main differences that distinguish these approaches from fault-tolerant CBSS are the underlying strategies and objectives. Unlike fault-tolerance techniques, reliability in the composition approaches disregard the details of fault-tolerance, while they aim to achieve reliability by enhancing system design [2, 78], component placement [54], component ranking [29], and level of decomposition [50, 65].

Arora and Kulkarni [2] present a component-based method for designing fault-tolerant systems that are able to tolerate multiple fault-classes independently by employing two types of components, namely detectors and correctors. Freixas and Pons [29] investigate the problem of finding the component(s) that have the biggest effect on system reliability (referred as the most important components) by introducing a number of ranking relations between components. Laprie *et al.* [50] refer to the significance of the level of decomposition with respect to the reliabilities of the software systems when applying design diversity. They propose a trade-off between smaller components that allow a better mastering of the decision algorithms and larger components that aid the degree of diversity. Lipton and Gokhale [54] present an optimization approach that produces a desirable deployment configuration of the application components given the application architecture, component reliabilities, and interface reliabilities. Mohamed and Zulkernine [63] provide an approach for trading off among the components' abilities of masking different failures with respect to component criticalities and optimal software fault-tolerance configurations. Mohamed and Zulkernine [65] also study the level of decomposition of CBSSs with respect to its impact on system reliability based on various component failure criticalities. Sotirovski [78] present some architectural properties that they consider as important parameters for achieving high reliability.

5.2 Reliability in the operational activities of CBSSs

In CBSS, each component is a unit of deployment that has its own requirements from the system resources. The allocation of these resources to different components may be a complex task especially in the CBSS applications where the resources are distributed and heterogeneous. Due to the resource allocation and the integration issues, the deployment of components in a system may compromise its reliability. Therefore, reliability must be ensured by guaranteeing the appropriate functionalities and the correctness of the system after the deployment activity of CBSSs. Belguidoum and Dagnat [5] provide a generic model and architecture for a dependable deployment.

Software upgrading introduces new versions of system components. Although, these new versions may involve better functionalities, they are not guaranteed to be more reliable. With regular maintenance activities of installing fixes and repairs, new versions of components may introduce new errors to the system. Reliability of the upgrading activities assures the system functionality according to the specification by avoiding the new errors that may be introduced by the new versions. Cook and Dage [11] provide a framework called HERCULES, for upgrading system components. Instead of removing the old version of a component, the framework keeps it running along with the new component.

CBSSs are usually modified or reconfigured during operational phase to control software functionalities or behaviours. This reconfiguration is often conducted while the system is not in use. The availability of the software system is a critical reliability sub-attribute for some mission-critical systems, especially, when these reconfigurations are dynamic and frequent. David *et al.* [17] present an end-to-end solution to define and execute reliable dynamic reconfigurations of component-based systems while guaranteeing their continuity of service.

5.3 Comparison of reliable design and operational activities of CBSSs

Table 6 summarizes the existing reliable design and operational activities and shows the main classes or activities (composition, configuration, deployment, and upgrading) and the major research goals (component placement, component ranking, component criticality, level of decomposition, multi-failure types, and post-activity reliability assurance) of these techniques. Most of the works focus on increasing system reliability by enhancing the composition among system components [2, 29, 50, 54, 63, 65, 78], while few works aim to incorporate reliability guarantees in the CBSS operation activities [5, 11, 17]. The main objectives of the composition approaches involve determining appropriate component placement [54] among distributed computers, identifying a component's rank [29, 78] among system components with respect to component importance, identifying and incorporating component criticality in the architectural design decisions [29, 63, 65], selecting the level of decomposition [50, 65] which involves system granularity and component sizes, enabling the system to tolerate multiple types of failures [2, 63], and finally ensuring system reliability during and after the operational activity [5, 11, 17].

	Composition	Configuration	Deployment	Upgrading	Component placement	Component ranking	Component criticality	Level of decomposition	Multi-failure types	Post-activity reliability assurance
Arora and Kulkarni [2]	✓								✓	
Freixas and Pons [29]	✓					✓	✓			
Laprie *et al.* [50]	✓							✓		
Lipton and Gokhale [54]	✓				✓					
Mohamed and Zulkernine [63]	✓						✓		✓	
Mohamed and Zulkernine [65]	✓						✓	✓		
Sotirovski [78]	✓					✓				
Belguidoum and Dagnat [5]			✓							✓
Cook and Dage [11]				✓						✓
David *et al.* [17]		✓								✓

Table 6: Summary of reliable design and operational activities.

6 Conclusions and open issues

CBSS development is recognized as a useful paradigm for structuring large software systems. Component technology can increase the productivity of software systems by providing reusability, abstraction, composition, and fast development time. Ensuring reliability in CBSSs is important for their effective applications in large scale and safety critical systems. In this paper, we survey the reliability efforts of CBSSs. Our survey will help the appropriate selection of CBSS reliability techniques from the existing ones, and it will serve as a stage for proposing novel or improved techniques. In this section, we summarize our survey and discuss a number of open issues.

6.1 Conclusions

In this paper, we classify reliability efforts in CBSSs based on the reliability means (fault-tolerance, reliability assessment, and reliable design and operational activities). We compare and contrast the existing techniques according to various component definitions and specification details.

In our taxonomy, fault-tolerance techniques for CBSSs are classified into programming-paradigm-based, environment-based, and model-based techniques. The programming paradigm-based fault tolerance includes inter-component and intra-component techniques. The environment-based fault-tolerance is either decentralized or centralized. The model-based fault-tolerance includes message-based and replica-based techniques. CBSS reliability assessment techniques are classified into component reliability models, system reliability models, and other supplementary techniques. We further classify system reliability models into state-based, path-based, and propagation-based models. Reliable design and operational activities are classified into reliable design activities and reliable operational activities.

In programming paradigm-based fault-tolerance, design diversity and exception handling are used to tolerate component and interface design faults. Intra-component techniques provide component structures or wrappers that help achieve system reliability. Inter-component techniques provide fault-tolerant component integration at the architectural level. Exception handling is applied at various levels of system architecture: component, connector, component group, and system configuration. Most of the intra-component techniques provide component structures that can be used as building blocks for fault-tolerant CBSSs.

The environment-based fault-tolerance approaches employ checkpointing and replication strategies to tolerate operational faults by assuming that components are design fault free. Decentralized fault-tolerance techniques facilitate the fault-tolerance strategy in a decentralized manner. Centralized techniques rely on their underlying platforms to store the replication data based on different criticality measures for system components. Most of the environment-based fault-tolerance techniques choose the replication style and management based on component criticality measures. The environment-based fault-tolerance techniques can mask only silent failures due to host crash and communication errors, while they cannot mask content failures or performance failures.

In model-based fault-tolerance, message-based techniques support fault-tolerance by achieving replica consistency and group invocation. Replica-based techniques support fault-tolerance using failure detection and error recovery mechanisms. The means to achieve model-based fault-tolerance are reliable messaging, state replication, replication management, replica distribution, error detection and notification, and error recovery. The model-based fault-tolerance techniques focus mainly on masking crash and communication loss failures.

In CBSS reliability assessment, component and system reliability models assess the reliabilities of individual components and CBSSs, respectively. Supplementary techniques provide support for CBSS reliability assessment, while they may not directly provide a reliability model. State-based models use control graph to represent software system architecture and predict reliability analytically. Path-based models compute software reliability considering the possible execution paths of a program. Propagation-based models incorporate the dependency of component failures by considering the error propagation among system components.

CBSS reliability techniques can be used during the early design, testing, and operational phases. Component failure behavior, flow control, and usage stress are the major parameters for most of these techniques. Unlike propagation-based models, state-based and path-based reliability models assume that components fail independently. Most of the propagation-based models disregard transition probabilities and usage stress in their computation and focus only on the corruptions (content failures) among data flows through component interfaces. Most of the existing reliability models cannot assess complicated architectures with loops and do not consider concurrent processing in their transition probabilities.

Most of the reliable activity efforts focus on increasing system reliability by enhancing the composition among system components. Few reliable activity works aim to incorporate reliability guarantees in CBSS operational activities. The main objectives of the composition approaches involve determining appropriate component placement among distributed computers, identifying component ranking among system components with respect to component importance, identifying and incorporating component criticality in the architectural design decisions, selecting the level of decomposition which involves system granularity and component sizes, enabling the system to tolerate multiple types of failures, and ensuring system reliability during and after the component activity.

6.2 Open issues

Like other researchers in this area, we also observe the lack of reliability efforts in CBSSs. In the following paragraphs, we discuss some open issues of CBSS reliability research.

Integration of CBSS fault-tolerance techniques: Each CBSS fault-tolerance class can handle certain types of faults or failures. Design diversity and exception handling are employed in the programming paradigm-based techniques for their abilities to tolerate design faults since they can be tailored to mask all types of failures. The environment-based and model-based techniques employ replication strategies for their abilities to tolerate operational faults by assuming that the components are design fault free. Fault-tolerance techniques in each class are not considered as complete solutions for tolerating all failures based on both design and operational faults. Therefore, these techniques can be integrated into more comprehensive fault-tolerance solutions that are able to address most reliability impairments. The other approach is to propose customized fault-tolerance solutions that are able to mask only the critical failures of our choice.

Component design faults in fault-tolerant CBSSs: The environment-based fault-tolerance techniques focus mainly on masking crash and communication loss failures. Few environment-based techniques address content failures by using exception handling. However, the techniques only tolerate content failures that are caused by operational faults or interface design faults. In other words, these techniques also disregard content failures based on component design faults. The model-based techniques mask communication failures such as message corruption and link disconnection, and halt failures such as computer and process crash. By considering inconsistent replicas as object variants, content failures are sometimes addressed and masked by using voting algorithm to adjudicate the results among a number of replica outputs (messages). However, the causes of these content failures are also the operational faults that lead to replica group inconsistency. Therefore, these techniques also disregard content failures based on component design faults. In both environment-based and model-based techniques, errors are detected using watchdog, heartbeat messages, or timer mechanisms by detecting when a component appears silent and stops responding. Therefore, environment-based and model-based techniques mainly focus on tolerating operational faults and interface design faults by masking only silent failures and disregarding component design faults.

Component failure dependency in system reliability models: State-based and path-based reliability models assume that components fail independently. By disregarding component failure dependency, state-based and path-based techniques assume a component failure does not affect any other component and it causes directly a system failure. However, in practice, a component failure may affect other components, while it may or may not cause a system failure. Therefore, component failure dependency needs to be considered in system reliability model.

Failure types in propagation-based system reliability models: Most of the propagation-based techniques for CBSS reliability assessment focus only on the corruptions (content failures) among data flows through component interfaces. Therefore, evaluating system architectures based on the existing propagation-based techniques disregard the impacts of other types of failures (*e.g.*, performance and silent failures) on system reliability. For example, if data transferred between two components are correct, while they are late in delivery, existing techniques may not consider this case as a failure.

Architectural loops in system reliability models: One limitation of the existing reliability assessment techniques is the inability to assess complicated architectures with loops. With the current software complexity, most software system architectures includes complicated architectural loops. Therefore, system reliability models need to incorporate architectural loops in the computation of system reliability.

Concurrency in system reliability models: With respect to system control flow and component usage stress, the existing system reliability models either do not incorporate them or only consider them based on sequential processing, where control transfers from a component to only one other component. To apply these techniques in concurrent and distributed systems, component failure behaviors and operational profiles should consider concurrent processing with respect to the transition probabilities of system control flow and component usage stress measures.

Lack of reliability assessment work based on component models: Most of the reliability assessment techniques consider programming paradigm-based components. A small number of works consider model-based components in their reliability assessment techniques. However, the lack of reliability assessment research that involve component models and architectural styles obstructs the effective application of these component models in mission critical and large scale software systems.

Level of decomposition: Software designers need to decide about the level of decomposition which involves component sizes and the number of components. In fault-tolerant CBSSs, decomposition level affects the architectural level fault-tolerance mechanisms and their parameters (*e.g.*, the size of variants and decision points of design diversity techniques and component level exception handling). However, the basis to choose the decomposition level of a CBSS has not been addressed adequately in the existing research. Decomposition level is important due to its major impact on reliability. By disregarding this impact, current software fault-tolerance techniques are prone to reliability decrease due to the inappropriate selection of the level of decomposition.

Nondeterminism in fault-tolerant CORBA applications: Only a few fault-tolerant CORBA techniques address the case of inconsistent replicas by using voting algorithm to adjudicate the correct message response of this group of replicas [16]. Consequently, the majority of fault-tolerant CORBA techniques are limited to deterministic application behaviors[9]. Nondeterminism can arise from sources such as multithreading, shared I/O, and error exceptions. Moreover, CORBA implementations from vendors are themselves multi-threaded and thus, nondeterministic [67]. Therefore, the majority of fault-tolerant CORBA techniques need more support for non-determinism, in particular, the techniques need either to achieve strong consistency with the multi-threaded implementations or to adopt solutions (*e.g.*, voting) that handle the cases of inconsistent replicas.

[9]Determinism suggests that, if two replicas of an object start from the same initial state and receive the same set of input invocations, they will reach the same final state and output the same responses.

References

[1] W. Abdelmoez, D.M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H.H. Ammar, B. Yu, and A. Mili, “Error propagation in software architectures,” Proc. of the 10-th IEEE International Symposium on Software Metrics (METRICS’04). Sep 2004, Morgantown, WV, pp. 384-393.

[2] A. Arora and S.S. Kulkarni, “Component based design of multitolerant systems,” IEEE Transactions Software Engineering, Jan 1998, vol. 24 pp. 63-78.

[3] A. Iliasov, A. Romanovsky, B. Arief, L. Laibinis, and E. Troubitsyna, “On Rigorous Design and Implementation of Fault Tolerant Ambient Systems,” IEEE 10-th Int’l Symp. on Object and Component-Oriented Real-Time Distributed Computing, 2007. (ISORC’07), 2007, pp. 141-145.

[4] M. Behnam, I. Shin, T. Notle, and M. Nolin, “An Overrun Method to Support Composition of Semi-independent Real-Time Components,” IEEE Proc. of the Int’l Conference on Computer Software and Applications (COMPSAC08), Vasteras, Aug 2008, pp. 1347-1352.

[5] M. Belguidoum and F. Dagnat, “Dependability in Software Component Deployment,” Proc. of the 2-nd Int’l Conference on Dependability of Computer Systems (DepCoS-RELCOMEX’07). Jun 2007, Szklarska, pp. 223-230.

[6] C.L. Blackmon and M.-L. Yin, “A design tool for large scale fault-tolerant software systems,” Reliability and Maintainability (RAMS), Annual Symp., Jan 2004, Page(s): 256-260.

[7] H.P. Breivol.d and M. Larsson, “Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles,” Proc. of the 33-rd EUROMICRO Conference on Software Engineering and Advanced Applications., Lubeck, Aug 2007, pp. 13-20.

[8] E. Cecchet, J. Marguerite, and W. Zwaenepoel, “Performance and scalability of EJB applications,” Proc. of the 17-th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2002, pp. 246-261.

[9] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, “Early Prediction of Software Component Reliability,” Proc. of the 30-th International conference on Software engineering (ICSE’08), China, 2008, pp. 111-120.

[10] R.C. Cheung, “A user-oriented software reliability model,” IEEE Trans. Software Engineering, vol. SE-6, no. 2, pp. 118125, March 1980.

[11] J.E. Cook and J.A. Dage, “Highly Reliable Upgrading of Components,” Proc. of the Int’l Conference on Software Engineering, 1999, Los Angeles, CA, USA, pp. 203-212.

[12] G. Chockler, D. Malkhi, B. Merimovich, and D. Rabinowitz, “Aquarius: A Data-Centric approach to CORBA Fault-Tolerance,” Proc. of The workshop on Reliable and Secure Middleware, in the 2003 Int’l Conference on Distributed Objects and Applications, Sicily, Italy (November 2003)

[13] V. Cortellessa and V. Grassi, “A Modeling Approach to Analyze the Impact of Error Propagation on Reliability of Component-Based Systems,” Proc. of the 10-th International Symposium on Component Based Software Engineering (CBSE’07), pp: 140-156, LNCS4608, Nov 2007, doi:10.1007/978-3-540-73551-9 10.

[14] D. Cotroneo a, N. Mazzocca b, L. Romano a, and S. Russo a, “Building a Dependable System from a Legacy Application with CORBA,” EUROMICRO Journal of Systems Architecture, Sep 2002, vol. 48 pp. 81-98.

[15] I. Crnkovic and M. Larsson, “Building reliable component-based software systems,” Artech House, 2002

[16] M. Cukier, J. Ren, C. Sabnis, W.H. Sanders, D.E. Bakken, M.E. Berman, D.A. Karr, and R. Schantz, “AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects,” Proc. IEEE 17-th Symp. Reliable Distributed Systems, pp. 245-253, Oct. 1998.

[17] P.C. David, M. Lger, H. Grall, T. Ledoux, and T. Coupaye, “A Multi-stage Approach for Reliable Dynamic Reconfigurations of Component-Based Systems,” *Proc. of the International Federation for Information Processing (IFIP’08)*, 2008, LNCS 5053, pp: 106111.

[18] R.V. Duncan and L. L. Pullum, “Object-oriented executives and components for fault tolerance,” IEEE Proc. of Aerospace Conference, 2001, vol. 6, pp. 2849-2855.

[19] W.W. Everett, “Software component reliability analysis,” Proc. of the 1999 IEEE Symposium on Application - Specific Systems and Software Engineering and Technology. 1999, p. 204.

[20] M.A. de C. Gatti, C.J.P. de Lucena, and J-P. Briot, "On Fault Tolerance in Law-Governed Multi-agent Systems," Springer Berlin/Heidelberg, 2007, LNCS 4408 pp. 1-20.

[21] N. Faci, Z. Guessoum, and O. Marin, "DimaX: A Fault Tolerant Multi-Agent Platform," Proc. of the 2006 Int'l workshop on Software engineering for large-scale multi-agent systems, 2006, pp. 13-20.

[22] J.C. Fabre and T. Perennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach," IEEE Trans. Computers, vol. 47, no. 1, pp. 78-95, Jan. 1998.

[23] M. Fahad, A. Nadeem, and M.R. Lyu, "A Survey of Fault Tolerant CORBA Systems," Springer Berlin/Heidelberg, 2007, LNCS 4803 pp. 505-521.

[24] P. Felber, "The CORBA Object Group Service: A Service Approach to Object Groups in CORBA," PhD thesis, Swiss Federal Inst. of Technology, Lausanne (1998).

[25] P. Felber and P. Narasimhan, "Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems," IEEE Transactions on Computers, 2004, pp. 497-511.

[26] P. Felber, "Lightweight Fault Tolerance in CORBA," Proc. Int'l Symp. Distributed Objects and Applications (DOA '01), Sep 2001, pp. 239-247.

[27] F.C. Filho, P.A. de C. Guerra, and C.M.F. Rubira, "An Architectural-Level Exception-Handling System for Component-Based Applications," Springer Berlin/ Heidelberg, 2003, LNCS 2847, pp. 321-340.

[28] J. Fraga, F. Siqueira, and F. Favarim, "An Adaptive Fault-Tolerant Component Model," Proc. of the 9-th IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03), 2003, pp. 179-186.

[29] J. Freixas and M. Pons, "Identifying Optimal Components in a Reliability System," IEEE Transactions on Reliability, Mar 2008, vol. 57, pp. 163-170.

[30] R. Friedman and E. Hadad, "FTS: A High Performance CORBA Fault Tolerance Service," Proc. of IEEE Workshop Object-Oriented Realtime Dependable Systems, Jan. 2002.

[31] S.S. Gokhale and K.S. Trivedi, , "Analytical Models for Architecture-Based Software Reliability Prediction: A Unification Framework," IEEE Transactions on Reliability, vol. 55, Dec 2006, pp. 578-590.

[32] S.S. Gokhale and M.R. Lyu, "A Simulation Approach to Structure-Based Software Reliability Analysis," IEEE Transactions on Software Engineering. 2005, vol. 31, pp. 643-656.

[33] S.S. Gokhale, M.R. Lyu, and K.S. Trivedi, "Reliability Simulation of Component-Based Software Systems," Proc. of the 9-th IEEE International Symposium on Software Reliability Engineering (ISSRE98), Paderborn, Germany, 1998, pp. 192-201.

[34] K. Goseva-Popstojanova and K.S. Trivedi, "Architecture based approach to reliability assessment of software systems," Int'l Journal on Performance Evaluation. 2001, vol. 45, pp. 179-204.

[35] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, "Large Empirical Case Study of Architecture-based Software Reliability," Proc. of 16th IEEE International Symposium on Software Reliability Engineering (IS-SRE'05), Nov 2005, Morgantown, WV, pp. 10 pp.

[36] K.E. Groosspietsch, "Optimizing the reliability of component-based n-version approaches," Proc. of the Int'l Parallel and Distributed Processing Symp. (IPDPS'02), Florida, Apr 2002, pp. 138-145.

[37] L. Grunske, "Early quality prediction of component-based systems A generic framework," The Journal of Systems and Software, 2007, vol. 80, pp. 678-686.

[38] P.A. de C. Guerra, C. Rubira, A. Romanovsky, and R. de Lemos, "Integrating COTS Software Components into Dependable Software Architectures," Proc. of the 6-th IEEE Int'l Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'03). Hokaido, Japan. May 2003.

[39] P.A. de C. Guerra, C. Rubira, and R. de Lemos, "An Idealized Fault-Tolerant Architectural Component," Proc. of the ICSE 2002 Workshop on Architecting Dependable Systems (WADS'02), May 2002.

[40] P.A. de C. Guerra, C. Rubira, and R. de Lemos, "A Fault-Tolerant Software Architecture for Component-Based Systems," Springer Berlin/Heidelberg, 2003, LNCS 2677, pp. 129-149.

[41] D. Hamlet, D. Mason, and D. Woitm, "Theory of software reliability based on components," Proc. of the 23-rd International Conference on Software Engineering (ICSE'01). May 2001, pp. 361-370.

[42] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren, "SaveCCM: a Component Model for Safety-Critical Real-Time Systems," Proc. of 30-th Euromicro Conference, Special Session Component Models for Dependable Systems, September 2004, pp. 627-635.

[43] M. Hiller, A. Jhumka, and N. Suri, “An Approach for Analysing the Propagation of Data Errors in Software,” IEEE Proc. of the International Conference on Dependable Systems and Networks (DSN’01), Goteborg, Sweden, Jul 2001, pp. 161-170.

[44] A.Iliasov and A.Romanovsky, “CAMA: Structured Coordination Space and Exception Propagation,” Springer Berlin/Heidelberg, 2006, LNCS 4119 pp. 181-199.

[45] V. Issarny and J-P. Banatre, “Architecture-based exception handling,” Proc. of the 34-th Annual Hawaii Int’l Conference on System Sciences (HICSS’34). 2001.

[46] Z.A. Khan, S. Shahid, H.F. Ahmad, A. Ali, and H. Suguri, “Decentralized architecture for fault tolerant multi agent system,” Proc. of Autonomous Decentralized Systems (ISADS 2005), 2005, pp. 167-174.

[47] M.M. Khokhar, A. Nadeem, and O.M. Paracha, “An Antecedence Graph Approach for Fault Tolerance in a Multi-Agent System,” Proc. of the 7-th IEEE Int’l Conference on Mobile Data Management (MDM’06), 2006, pp. 137-141.

[48] S. Krishnamurt and A.P. Mathur, “On the estimation of reliability of a software system using reliabilities of its components,” Proc. of the 8-th International Symposium On Software Reliability Engineering (ISSRE’97). Nov 1997, pp. 146-155.

[49] A. Lanoix, D. Hatebur, M. Heisel, and J. Souquieres, “Enhancing Dependability of Component-Based Systems,” Springer-Verlag Berlin Heidelberg. 2007, LNCS 4498, pp. 4154.

[50] J.C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, “Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures,” IEEE Computer, vol. 26, no. 7, pp. 39-51, Jul 1990, doi:10.1109/2.56851.

[51] M. Larsson, “Predicting Quality Attributes in Component-based Software Systems,” Mlardalen University Press, Arkitektkopia, Vsters, Sweden, 2004, ISBN: 91-88834-33-6.

[52] N.G. Leveson, “Software safety: why, what, and how,” ACM Computing Surveys (CSUR) archive, Jun 1986, vol. 18, pp. 125-163.

[53] R. de Lemos, P.A. de Guerra, and C.M.F. Rubira, “A fault-tolerant architectural approach for dependable systems,” IEEE Sotware. Mar 2006, vol. 23, pp. 80-87.

[54] M.W. Lipton and S.S. Gokhale, “Heuristic Component Placement for Maximizing Software Reliability,” Springer Series in Reliability Engineering, Recent Advances in Reliability and Quality in Design., 2008, pp. 309-330.

[55] A. de Luna Almeida, S. Aknine, J-P. Briot, and J. Malenfant, “Plan-based replication for fault-tolerant multi-agent systems,” Proc. of the 20-th Int’l Parallel and Distributed Processing Symp.. (IPDPS‘06), 2006, pp. 7 pages.

[56] A. de Luna Almeida, S. Aknine, J-P. Briot, and J. Malenfant, “A Predictive Method for Providing Fault Tolerance in Multi-agent Systems,” Proc. of Int’l Conference on Intelligent Agent Technology, (IAT ’06), 2006, pp. 226-232.

[57] S. Maffeis, “Run-Time Support for Object-Oriented Distributed Programming,” PhD thesis, Univ. of Zurich, Feb. 1995.

[58] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni, “An Interoperable Replication Logic for CORBA Systems,” Proc. 2-nd Int’l Symp. Distributed Objects and Applications (DOA ’00), pp. 7-16, Feb. 2000.

[59] E. Marsden, J.C.Fabre, and J. Arlat, “Dependability of CORBA systems-service characterization by fault injection,” Proc. of 21-st IEEE Symp. on Reliable Distributed Systems, 2002, pp. 276-285.

[60] O. Marin, M. Bertier, and P. Sens, “DARX-a framework for the fault-tolerant support of agent software,” Proc. of the 14-th IEEE Int’l Symp. on Software Reliability Engineering (ISSRE’03), Havre, France, Nov 2003, pp. 406-416.

[61] M.M. X. Meng and H. Zhang, “An efficient fault-tolerant scheme for mobile agent execution,” Proc. of 1st Int’l Symp. on Systems and Control in Aerospace and Astronautics. (ISSCAA‘06), 2006, pp. 1306-1310.

[62] Microsoft, “The Component Object Model Specication,” Microsoft Standards, Report v0.99, Redmond, WA: Microsoft, 1996.

[63] A. Mohamed and M. Zulkernine, “Improving Reliability and Safety by Trading Off Software Failure Criticalities,” Proc. of the 10-th IEEE Int’l Symp. on High Assurance System Engineering (HASE’07). Nov 2007, Dallas, Texas, pp. 267-274.

[64] A. Mohamed and M. Zulkernine, "On Failure Propagation in Component-Based Software Systems," Proc. of the 8-th IEEE Int'l Conference on Quality Software (QSIC'08), IEEE CS Press, Oxford, UK, August 2008, pp.402-411.

[65] A. Mohamed and M. Zulkernine, "At What Level of Granularity Should We be Componentizing for Software Reliability?," Proc. of the 11-th IEEE Int'l Symp. on High Assurance System Engineering (HASE'08). Dec 2008, Najing, China, pp.273-282.

[66] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little, "Design and Implementation of a CORBA Fault-Tolerant Object Group Service," Proc. of the 2-nd IFIP WG 6.1 Int'l Working Conf. Distributed Applications and Interoperable Systems, June 1999.

[67] P. Narasimhan, "Fault-Tolerant CORBA: From Specification to Reality," IEEE Transactions on Computers, 2007, pp. 110-102.

[68] P. Narasimhan, "Transparent Fault Tolerance for CORBA," PhD thesis, Dept. of Electrical and Computer Eng., Univ. of California, Santa Barbara, Dec. 1999.

[69] B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt, "Doors: Towards High-Performance Fault Tolerant CORBA," Proc. 2nd Int'l Symp. Distributed Objects and Applications (DOA '00), Feb. 2000, pp. 39-48.

[70] Object Management Group, "The Common Object Request Broker: Architecture and Specification," OMG Standards Collection, Report v2.4, 2000.

[71] Object Management Group, "OMG Unified Modeling Language (OMG UML)," Superstructure, Version 2.1.2, OMG Available Specification without Change Bars, formal/2007-02-05, Nov 2007.

[72] Object Management Group, "CORBA Component Model Specification," OMG, Version 4.0 formal/06-04-01, Apr 2001.

[73] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic, "Error propagation in the reliability analysis of component-based systems," Proc. of 16-th IEEE International Symposium on Software Reliability Engineering (ISSRE'05). Nov 2005, Morgantown, WV, (10)pages.

[74] L.L. Pullum, "Fault tolerance techniques and implementation," Artech House, 2001, ISBN 1-58053-470-8.

[75] A. Romanovsky, "Exception Handling in Component-Based System Development," Proc. of the 25-th Int'l Computer Software and Application Conference (COMPSAC 2001), Chicago, IL, October, 2001. pp. 580-586.

[76] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W.H. Sanders, D.E. Bakken and D.A. Karr, "Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA," Proc. of Dependable Computing for Critical Applications, 1999, Page(s): 149-168.

[77] V.S. Sharma a and K.S. Trivedi, "Quantifying software performance, reliability and security: An architecture-based approach," The Journal of Systems and Software, 2007, vol. 80, pp. 493-509.

[78] D. Sotirovski, "Towards fault-tolerant software architectures," Proc. of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, Netherlands, 2001, pp. 7-13.

[79] Sun Microsystems, "Enterprise JavaBeans Specication," Sun Microsystems, Inc., Report v2.0, California, Oct 2000.

[80] C. Szyperski, "Component software: beyond object-oriented programming," Addison-Wesley, 1998, ISBN 0-201-17888-5.

[81] A. Vaysburd and K. Birman, "The Maestro Approach to Building Reliable Interoperable Distributed Applications with Multiple Execution Styles," Theory and Practice of Object Systems, vol. 4, no. 2, pp. 73-80, 1998.

[82] J. Voas, "Error propagation analysis for COTS system," Journal of Computing and Control Engineering, Dec 1997, vol. 8, pp. 269-272.

[83] J. Voas, G. McGraw, A. Ghosh, and K. Miller, "Glueing Together Software Components: How Good Is Your Glue?," Proc. of the Pacific Northwest Software Quality Conference, oct 1996, pp. 338-349.

[84] S.M. Yacoub, B. Cukic, and H.H. Ammar, "Scenario-Based Reliability Analysis of Component-Based Software," Proc. of the 10-th International Symposium on Software Reliability Engineering (ISSRE'99). 1999, pp. 22-31.

[85] W. Yi-Min, O.P. Damani, and L. Woei-Jyh, "Reliability and availability issues in distributed component object model (DCOM)," Fourth International Workshop on Community Networking Proceedings, 1997, pp. 59-63.