A Layered Approach for Identifying Systematic Faults of Component-based Software Systems

Min Yang Jung
Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218, USA
myj@jhu.edu

ABSTRACT

Research on fault, fault tolerance, and fault-tolerant software systems has been done in a number of areas. Component-based software engineering, meanwhile, has emerged and been widely adopted in many application domains as software systems become more and more complex. However, conventional methods and techniques for fault tolerance are not directly applicable to the recent component-based software systems and faults have been defined within application-specific or domain-specific context. Furthermore, there has not been much attention to identify systematic faults, especially in the recent software systems. Thus, this paper presents a fault model that is useful for designing and analyzing component-based software systems. Using the *cisst* component model, the fault model is identified using a layered approach that decomposes a component-based software system into five different layers. With the fault model identified, an illustrative example is presented to show how this model is applied to a domain-specific application.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; H.1.1 [**Models and Principles**]: Systems and Information Theory

General Terms

Theory

Keywords

Component-based software engineering, systematic faults, layered fault model

1. INTRODUCTION

The complexity and scale of modern software systems have been significantly increased to meet both a large set of complicated functional requirements and nonfunctional requirements such as safety, maintainability, reusability, scalability, interoperability, and performance. In the software engineering domain, various software engineering principles and methodologies have been proposed to cope with these issues and component-based software engineering (CBSE) has emerged as an effective approach [40]. It has been widely adopted in various areas where computing systems play a key role.

Robotics is one representative domain where this trend appears. Typical robotic systems consist of hardware and software and these two inherently different parts should be integrated and coordinated together in order to perform given tasks safely and effectively. Furthermore, robots have been evolving to become more intelligent with much more capabilities as hardware and software technology advance and this requires recent robotic systems to be able to control and manage a variety of different types of sensors, actuators, and computational units. These introduce complexity issues to the modern robotic software systems.

In the robotics research community, there has been a recognition of this issue and the CBSE concept is being progressively adopted [3]. Many software packages and middlewares based on the concept of software reuse and the CBSE approach have been developed and released to facilitate robotics research [5, 34, 14, 28, 19], although the use of such software packages and middlewares is not yet a state-of-the-practice software development approach in the robotics field. During the last half century, robotics research has evolved from industrial robotics to service robotics such as medical robotics, rehabilitation robotics, and humanoid robotics to assist humans and get robots closer to human needs [13]. This requires robots to interact with humans and it necessarily leads to the issue of system safety. Of course, the system safety issue gets further deepened as the system complexity increases. Thus, the challenge is how to effectively cope with the system safety requirement.

Safety is a system problem [26, 30] and has been researched to prevent accidents in complex and safety-critical systems such as aircraft fly-by-wire controls, oil and chemical processing, hospital life-support systems, and other numerous commercial and industrial applications [10]. The concept of safety is one aspect of *dependability* which is an integrating concept that encompasses availability, reliability, safety, integrity, and maintainability [2]. Among many means that have been developed to attain these various attributes of dependability, *fault tolerance* is the most promising strategy [11]. Fault-tolerance and fault tolerant systems have been

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WCOP'11, June 20–24, 2011, Boulder, Colorado, USA.
Copyright 2011 ACM 978-1-4503-0726-0/11/06 ...$10.00.

studied for many years [35, 21, 24, 37] and several recent studies with the emergence of the CBSE approach focus on fault tolerance for component-based systems and architectures [22, 9, 31, 18, 6, 41].

In most studies, fault itself has been defined within the context of applications and not much attention has been paid to identify fault categories. Some previous studies summarized fault categories [27, 29] that are application-independent and thus can be used across application domains. However, they do not incorporate the concept of components and component-based software systems (CBSS). Thus, the goal of this paper is to propose a foundational fault model to design and analyze fault-tolerant CBSS by identifying application-independent but commonly found faults–i.e., *systematic faults*–of the systems. When this foundation is established, it can be further extended and applied to fault-related research for fault-tolerant CBSS. As a first step, this paper identifies typical types of faults in CBSS using a hierarchical and structured approach, i.e., the *layered* approach.

This paper is structured as follows: The following section describes the component model and framework used in this paper. In Section 3, the layers of the system are defined and the fault model is identified. Section 4 shows how the identified fault model can be applied to a domain-specific application.

2. BACKGROUND

The basic unit of CBSE is a *component*. One widely accepted definition, due to Szyperski[40], is: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

According to a recent survey [25], a software component model defines what components are (*Semantics*), how components are defined, constructed, and represented (*Syntax*), and how components are composed, assembled, or deployed (*Composition*). A wide variety of different component models have been proposed and developed both in the software engineering domain and the robotics research domain. In the software engineering domain, a partial list of component models includes JavaBeans [39], .NET [44], CCM [36], Koala [42], SOFA [33], Acme-like ADLs [8], UML 2.0 [7], PECOS [32], and Fractal [4]. In the robotics research community, the component model is defined by robot software packages or middlewares, such as Orca [28], OpenRTM [1], Orocos [5], ROS [34], and CLARAty [43]. We developed the *cisst* package [19] to support our research in medical robotics and computer-assisted surgery, and use this software framework to illustrate the concepts of this paper.

2.1 The *cisst* component model

The base component of the *cisst* component model is `mtsComponent` (Figure 1). It has an internal *component state* and *state table* [20], and consists of lists of *provided interfaces*, *required interfaces*, *input interfaces*, and *output interfaces*. The latter two interface types support the data flow for real-time vision processing and are not discussed further. Figure 2 shows the internal structure of *cisst* components. Each provided interface can have multiple *command objects* which encapsulate the available *services*, as well as *event generators* that broadcast events with or without payloads. Four strongly-typed command object classes are defined to handle

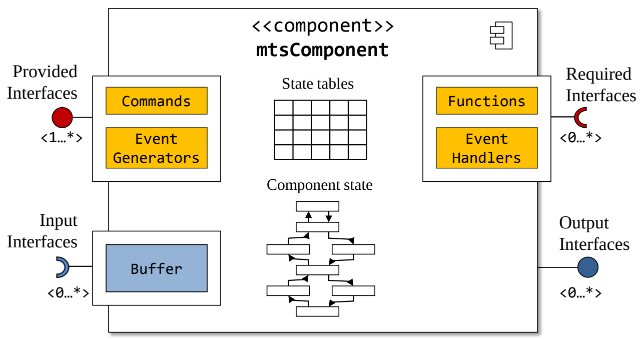


Figure 1: Overview of *cisst* base component, `mtsComponent`

commands with no parameters, one input parameter, one output parameter, or one of each. Each required interface has multiple *function objects* that are *bound* to command objects to use the services that the connected interface provides. It may also have *event handlers* to respond to events generated by the connected interface.

When two interfaces are *connected* to each other, all function

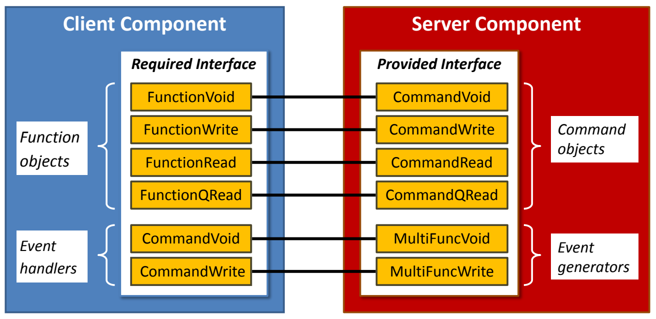


Figure 2: *cisst* Component Model: Structure and connection of client and server components

objects in the required interface are bound to the corresponding command objects in the provided interface, and event handlers in the required interface become *observers* of the events generated by the provided interface. Because the component with the required interface uses services provided by the component with the provided interface, we call the former the *client* component and the latter the *server* component.

2.2 The *cisst* component-based framework

The *cisst* component model is implemented by the *cisst* package [16]. The *cisst* package is a collection of component-based open-source C++ software libraries originally developed for computer-assisted intervention systems in a multi-threaded environment [19]. It includes an operating system abstraction library (*cisstOSAbstraction*) that supports Microsoft Windows, Linux, Mac OS X, real-time Linux variants (e.g., RTAI, Xenomai), and QNX. The *cisstMultiTask* library defines and implements the *cisst* component model and provides the component-based environment for *cisst*. Depending on the thread execution scheme, several different types of components are derived from this base class including `mtsTaskPeriodic`, `mtsTaskContinuous`, `mtsTask-FromCallback`, and `mtsTaskFromSignal`. All of these derived components contain a thread, whereas the base class does not (the client has to provide the thread). These components can run in a single process for optimal real-time performance or can be distributed across networks for larger scale or heterogeneous systems.

Multi-threaded architecture in a single process. In the *cisstMultiTask* framework, two different types of components/interfaces are defined: *system* components/interfaces and *user* components/interfaces. The system components include the *Local Component Manager (LCM)* and the *Global Component Manager (GCM)*. The LCM is unique within the process and all components in the same process are internally connected to it. This internal connection is automatically created and managed by the framework. The LCM controls and monitors the states of each component using this interface. Each LCM is connected to the GCM, which is globally unique to the whole system. The GCM manages all LCM(s) in a system and maintains system-wide information such as a list of processes, components, interfaces, and connections. In case of the multi-threaded architecture, the LCM and GCM exists in the same process. Figure 3 shows the multi-threaded architecture which has the best real-time performance.

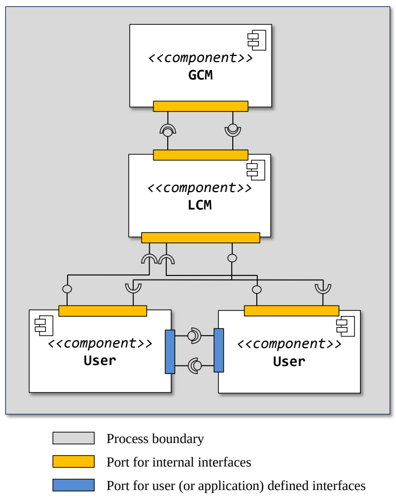


Figure 3: Multi-threaded architecture

Multi-process architecture for distributed components. Recently, the multi-*threaded* architecture above has been extended to support multi-*process* and multi-*host* scenarios by the introduction of a network layer [17]. This extension uses the Proxy Pattern [38] to seamlessly support the component-based data exchange mechanism between two remote components across the network. With the extension, users can use two different configurations: (1) the *standalone configuration* that only supports multi-threaded systems and (2) the *networked configuration* that supports not only multi-threaded systems but also multi-process (and multi-host) systems. Furthermore, depending on the characteristics or requirements of components, both configurations can be deployed together to take advantage of both the optimal performance of multi-threaded components in the same process and data exchange between different processes across networks. This provides system designers with more freedom and flexibility in deploying components when designing or building a system. The conversion from the standalone configuration to the networked configuration requires minimal user code-level changes and can be done seamlessly by the LCM [17].

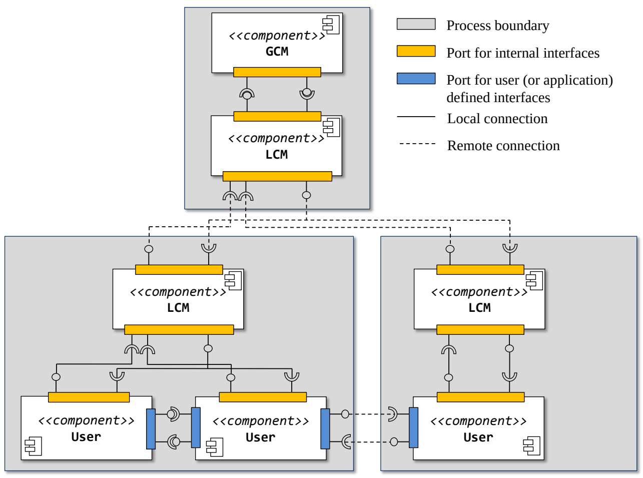


Figure 4: Multi-process (distributed) architecture

3. APPROACH

This section presents the layered approach proposed to identify systematic faults of CBSS. A *fault* is defined as the cause of an *error* that is a part of the system state which is liable to lead to the delivery of a *service* (or system behavior) not complying with the service *specification* [23]. The layered approach focuses on *systematic* faults caused by *structural* units of CBSS, which are domain or application independent and are closely tied to the underlying framework or middleware that the system is based on. This type of fault may also be related to the non-functional aspects of a system such as performance, availability, or safety. The approach considers the *cisst* component model (Section 2.1) with the multi-process architecture for distributed components (Figure 4). The *cisst* framework is then decomposed into five different layers (the *System*, *Process*, *Component*, *Interface*, and *Execution* layer) and the systematic faults are identified at each layer. Faults of each layer are not separated from the others but are linked together in a hierarchical manner such that faults of each layer include all faults from the lower layers. Although the identified fault model considers the multi-process architecture, it should be noted that the fault model can also be applied to the multi-threaded architecture as well because *cisst* supports both architectures seamlessly with the same programming model.

3.1 System Layer

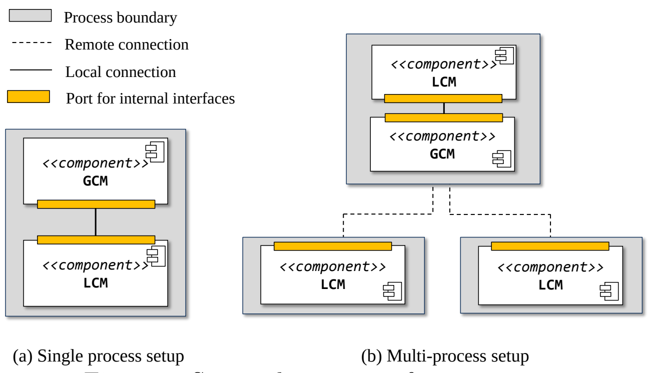


Figure 5: System layer view of *cisst* system

The system layer is the top level view of the system. Two elements that exist at this layer are a *process* and a *connection* between processes. The connection represents an *internal* connection used for system management purposes. The *cisst* framework uses two different types of internal connections: (1) generic connections for component management, which are defined by the interface layer (Section 3.4), and (2) middleware-specific connections for system boot-strapping. As in Figure 5, the GCM is unique to the whole system and each process has a single LCM. Every LCM connects to the GCM to join the system that the GCM manages. This layer aims to make sure the system operates properly by guaranteeing that all processes are *functionally* alive and the GCM maintains *valid* connections with all processes in the system. If a process hangs due to deadlock or unintended infinite loop, or it crashes or terminates abnormally, the process does not function properly (***process faults***). When multiple processes run across networks, all processes should join the system by establishing a connection to the GCM. Each process remains in the system unless a process terminates or the connection becomes invalidated due to network issues such as disconnection, exceptions, or heart-beat failures (***connection faults, network faults***). In the *cisst* framework, the network abstraction layer that has middleware-independent design [17] is defined and the current implementation uses ICE [15].

3.2 Process Layer

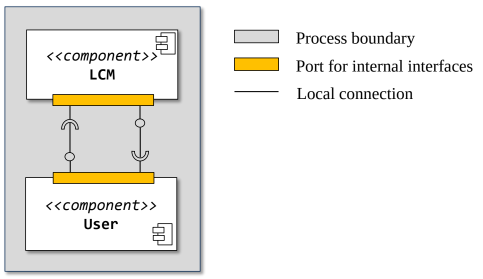
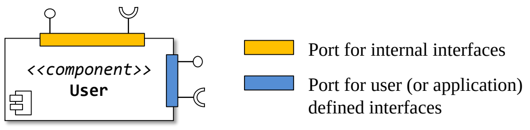


Figure 6: Process layer view of *cisst* system

The process layer is defined within a process boundary and is comprised of *components* in the same process as in Figure 6. The primary concern of this layer is that all components in the same process operate properly. The *proper* operation of a component means that a component remains valid (i.e., in the expected states) throughout the life cycle of the software system. Otherwise, ***component faults*** occur. As an example, a component fails to operate properly if it does not start correctly due to improper initialization or state change from normal to one of the unexpected states at run-time (e.g., from `started` to `paused`). To systematically deal with this type of fault, a component model is expected to define component states so that the system can monitor each component's state to determine if it is running properly. In the *cisst* framework, eight different states are defined and the state information about all components in the entire system is available to all components through system-wide services.

3.3 Component Layer

The component layer focuses on the *context integrity* of components. The definition of context integrity varies depending on component models because each component



<<component>>
User
Port for internal interfaces
Port for user (or application) defined interfaces

Figure 7: Component layer view of *cisst* system

model has its own design and intended use or domain. However, the context integrity in general can be considered in two respects: the *structural* (or *functional*) and *non-structural* (or *non-functional*) aspect. The structural aspect looks at basic elements that build a component (e.g., interface) or functional properties of a base component, whereas the non-structural aspect focuses on qualitative properties which play a key role in terms of stability, availability, and performance.

In case of *cisst*, thread execution requirement violation can occur when a component of type `mtsTaskPeriodic` does not run at the specified period (***structural faults***) or when a component of type `mtsTaskFromSignal` does not run when a signal is raised. In contrast, ***non-structural faults*** cannot be defined within the framework because those faults need to be defined within application-specific or domain-specific context.

3.4 Interface Layer

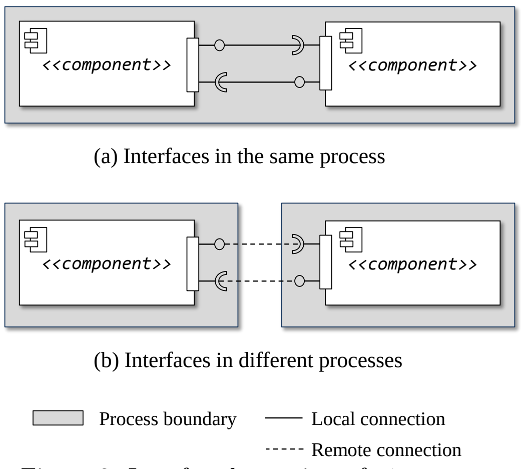


Figure 8: Interface layer view of *cisst* system

The interface layer is where *connections* between components are defined. In CBSS, this layer contains core information about the entire system because the connection topology determines how components are deployed, composed, and assembled to build a system (the *"composition"* of components [25]). It also determines information flows between interfaces and forms the basis of other mechanisms such as dynamic component composition and fault-tolerance via reconfiguration. Connections at this layer include both internal and user (application) connections, whereas the connection at the system layer has only internal connections.

Generally, potential faults resulting from the connection can be considered at two phases, the *on-connect* and *post-connect* phases. When establishing a connection between two interfaces, the contractual specification [40] of the two interfaces should match. Otherwise, a connection fails to establish (***on-connect faults***). This issue becomes more complicated if a system is required to support dynamic composition, component reconfiguration, or backward compatibility as components evolve because the interface contract and compatibility management becomes non-trivial in those cases. After a connection is established, it can be disconnected due to network issues as discussed in Section 3.1 (***post-connect faults***).

3.5 Execution Layer

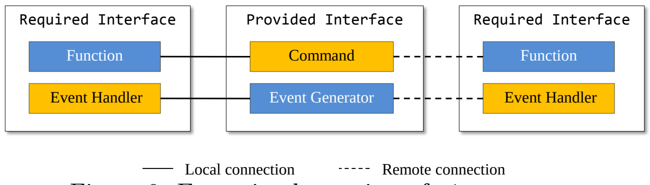


Figure 9: Execution layer view of *cisst* system

The execution layer is highly dependent on the design or implementation of middlewares or packages used. However, systematic faults in general can be identified based on the component model and related framework that implements the model. According to the *cisst* component model, services that a provided interface provides are encapsulated as *command objects* and used by bound *function objects* of the connected required interfaces. Likewise, a provided interface defines events as *event generators* which are broadcasted to the connected required interfaces and are handled by *event handlers (observers)*. As core structural elements of interfaces, these four elements constitute the command layer as shown in Figure 9 and provide the fundamental data exchange mechanisms for the system.

The key issues of this layer are *execution efficiency* and *payload validity*. The execution efficiency represents how efficiently commands can be executed and is determined by design requirements (e.g., payload size and frequency of the command execution) and implementation efficiency. If payloads with large data such as multimedia (e.g., image, audio, or video) are used between components at high frequency such as a few hundreds or a few kilo Hz, this issue should be considered carefully in the design phase of the *execution* or *interface* layer because the execution efficiency may be a bottleneck of the performance of the entire system (***execution inefficiency***). This is not exactly a fault but could be understood as an extended concept of systematic faults considering that a fault is defined as *a discrepancy in the software which impairs its ability to function as intended* [27]. For example, it is not unusual for recent robot application systems to use a real-time stereo high-definition (HD) vision system with 30 frames-per-second (fps). This becomes more challenging in systems with the multi-process architecture because it introduces network-related issues that may potentially lead to ***network faults***. The payload validity is also important for correct execution of commands. If a numerical-type payload is out of range or an encoded (or compressed) multimedia payload is corrupted, they directly lead to systematic faults of this layer (***invalid payload faults***).

3.6 Summary

Table 1 summarizes the systematic fault models identified in the previous sections.

4. ILLUSTRATIVE EXAMPLE

In this section, a robot control system with vision processing based on the *cisst* framework is presented as an illustrative example. This real-world scenario demonstrates how the fault model in Table 1 can be applied to CBSS to analyze and identify potential faults at the design phase in a structured and systematic way.

Table 1: Model of systematic faults of component-based software systems

Layer	Elements	Faults
System	Process, Connection (internal)	Process faults Connection faults Network faults Faults from *Process* layer
Process	Component	Component faults Faults from *Component* layer
Component	Context integrity	Structural faults Non-structural faults Faults from *Interface* layer
Interface	Connection	On-connect faults Post-connect faults Faults from *Execution* layer
Execution	Command object, Function object, Event generator, Event handler	Execution inefficiency Network faults Invalid payload faults

Figure 10 shows an architecture of a typical robot control application. In general, the computer system contains five primary elements: the *application, sensor, effector, operator*, and *computer* [10]. This concept also applies to the presented system (each element is identified in the figure) although the system extends the concept of *computer* to the distributed CBSS. The robot on the left has real-time video capture functionality and interacts with the component-based control software system in the middle. The control system consists of two processes: `Control` and `Visualization`. In the `Control` process, the `Joint Encoder` component reads the current state (e.g., position, velocity, or acceleration) from the robot and the `Controller` component uses this information for the execution of robot control algorithms. Output of the control algorithm is then passed to the `Actuator` component which actually moves the robot by controlling motors. The control system may have an additional component, the `Console` component, for development purposes such as monitoring, debugging, or event injections. The robot's real-time video data is read by the `Video Reader` component that makes the video data available to the rest of the system. The `Visualization` process has two components: `Visualizer` and `GUI`. The `Visualizer` component connects to the `Video Reader` in the `Control` process to apply filters to video stream data. This may requires the multi-process (distributed) architecture if high-performance computing is necessary. The `Visualizer` component can have a connection with the `Controller` component if it uses outputs from the `Controller` for visualization (e.g., showing the next goal position). The `GUI` component reads video data from the `Visualizer` component and visually displays them through the graphical user interface.

To show how Table 1 is "*translated*" into the domain-specific context to identify potential fault scenarios, the layered approach is applied to the robot control system above as follows. For simplicity, it is assumed that (1) no fault exists between the robot and the `Joint Encoder`, `Video Reader`, and `Actuator` components, and (2) the internal structural elements of the *cisst* framework such as LCM, GCM, internal interfaces and connections are also ideal, i.e., fault-free.

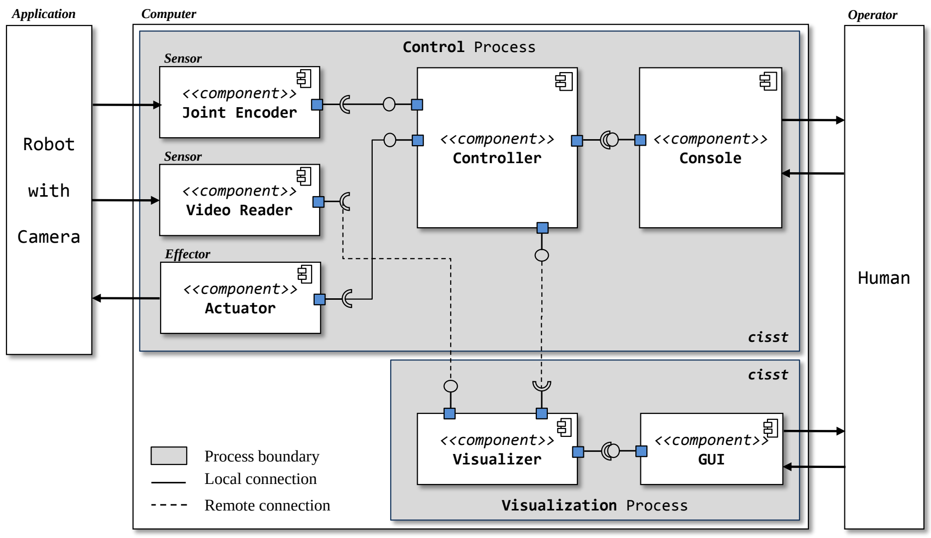


Figure 10: Overview of robot control system with vision processing feature (internal structural elements of the *cisst* framework such as the LCM, GCM, internal interfaces, and internal connections are not shown)

The *system* layer defines *process* and *connection* as its elements. In the presented system, there are two processes (`Control` and `Visualization`) and two hidden connections (one between the GCM process and the `Control` process and the other one between the GCM process and the `Visualization` process). If the `Control` process crashes or hangs due to an unknown error, it does not operate properly (**process fault**) and the process leaves the system. Then, the `Visualization` process detects the system-level change as a form of network disconnection or exception (**network fault**) and all connections with the `Control` process become invalidated (**connection fault**).

The *process* layer focuses on all *components* within the same process. The `Control` process consists of the `Joint Encoder, Video Reader, Actuator, Controller`, and `Console` component and the `Visualization` process has the `Visualizer` and `GUI` components. The most dramatic case would be the case where the `Joint Encoder` stops at run-time while the robot is moving (**component fault**). Considering that the output of the `Joint Encoder` is the most key data to the whole system, this fault obviously results in system-wide critical issues and renders all the subsequent processing pipeline invalidated. If the `Actuator` or `Controller` component fails at start up or the `Visualizer` pauses at run-time when the `Controller` uses the output of the `Visualizer` component (e.g. visual servoing), the system would also experience system-wide faults.

In the *component* layer, the *context integrity* of all components in the system is required. For demonstration purposes, the `Video Reader` and `Controller` components are selected with the following assumptions: (1) For the `Controller` component, it would be necessary that the control loop should run at a frequency of 1 kHz without overrun. Also, (2) the `Video Reader` component should be able to support real-time video processing with HD quality at 30 fps and should provide a captured video stream to the `Visualizer` component with lossless compression ratio 2:1. In this case, if the period becomes irregular or shows significant jitter, that would be an example of a **structural fault**. The `Video Reader` causes **non-structural faults** if the `Video Reader` does not meet these requirements when it runs at a lower frame rate (e.g., due to processing delay), it has compression ratio less than 2:1, or when data loss happens during the video compression process.

The *interface* layer focuses on all connections between components in the system. In the given system, there are four local connections and two remote connections. If one of the connections fails at start up due to the mismatched interface specification or a new connection fails at run-time due to the interface incompatability issue, this causes the **on-connect faults**. For a remote connection between two remote interfaces, faults from the network abstraction layer lead to faults from this layer (**post-connect fault**).

Among a large set of elements of the *execution* layer, the command object of the `Visualizer` component that retrieves and processes video input from the `Video Reader` would be a good example of **execution inefficiency**. If the algorithm of that command object is poorly designed or its implementation requires a lot of computation time, the command object would be a bottleneck of the performance of the entire system. Any use of remote command objects introduces potential network-related issues (**network faults**). The function object of the `Controller` component can be an example of an **invalid payload fault**. If the `Controller` component fails for some reason and requests the `Actuator` component to move the robot with invalid payload (e.g., joint position or velocity values beyond the limit), faults can happen by the `Actuator` component (in reality, however, this kind of error is usually gracefully handled by lower level drivers of the robot).

5. DISCUSSION

The goal of this study is to identify systematic faults that are potentially but typically found in CBSS. The benefits of the identified fault model can be categorized in three respects. At the initial system design phase, the fault model provides system designers with a tool or guideline to systematically analyze the system with requirements and to forecast potential systematic faults and performance issues in advance (*design phase benefit*). Also, if a component-based framework adopts the fault model and deals with systematic faults at the framework level, it would increase productivity by reducing the implementation overhead for fault and error handling at the user or application layer (*implementation phase benefit*). Furthermore, the fault model can be extended to develop methods for fault tolerant or safety critical systems, such as automatic fault analysis, automatic generation of test cases, or system safety evaluation (*analysis phase benefit*).

Although the proposed approach uses the *cisst* component model and its underlying framework, the identified fault model may be applied to other component models and frameworks flexibly because of its fundamental property of application or domain-independence and its basis on analysis of general properties of components and CBSS.

6. RELATED WORKS

For safety-critical and fault-tolerant systems, faults and fault-related research have been studied with relevant areas such as system safety and reliability engineering.

[35] uses recovery blocks, conversations, and fault-tolerant interfaces to review and present the methods for structuring complex computing systems. But it focuses on fault tolerance, rather than identifying faults of systems. Using preventive and detective tools and techniques, [27] presents methods to reduce computer software errors. It presents symptomatic fault categories with frequencies such as logic, data handling, interface, data input/output, computational, data base, and data definition. This categorization is conceptually similar to the fault model that this paper presents but it is not directly applicable to recent CBSS. A previous survey [29] of software faults presents a number of different sets of fault categories, which may complement the fault model identified in this paper. However, many different sets of categories have a flat structure and are not systematically organized. Thus, it is not suitable to modern CBSS.

With the emergence of the concept of components or CBSE, fault-tolerant CBSS have been studied [45, 9, 6, 41]. [45] tries to define the structural elements of components and CBSS using a component interaction graph. It presents an approach that has a conceptual similarity with the layered approach in this paper but it only defines partial aspects of components and CBSS. [9] present an approach for fault tolerant CBSS by structuring unreliable components into a fault tolerant architecture. In the proposed fault model, this structuring can be an example of how fault tolerance can be implemented for the *component* layer. [6] also presents fault-tolerant CBSS that complements fault tolerance with fault removal techniques. In [41], an automated middleware specialization approach is proposed for failure handling and recovery.

In the medical robotics domain, Fei [12] presents a medical robot system with a systematic method to analyze, control, and evaluate the system. Although the system does not adopt the component concept, its layered approach for software design has close similarities with the approach proposed in this paper.

7. CONCLUSION

In this paper, the layered approach has been proposed to identify the systematic fault model of CBSS. It provides system designers with a systematic and structured tool to design and analyze CBSS and can bring benefits at the design, implementation, and analysis phase. The future work has three directions. The first one is to further develop and complement the proposed fault model by applying it to other component models or component-based frameworks. It will improve the generality and flexibility of the proposed fault model. The second direction is to implement the complete fault model using the *cisst* framework and evaluate the validity or effectiveness of the model experimentally. The third is to extend the fault model so that it provides a useful basis for fault-tolerant or safety-critical systems research. Although the proposed model needs more review and improvement for its completeness, it could be useful for both the software engineering and robotics domains.

8. REFERENCES

[1] N. Ando, T. Suehiro, and T. Kotoku. A Software Platform for Component Based RT-System Development: OpenRTM-Aist. In *Simulation, Modeling, and Programming for Autonomous Robots*, volume 5325 of *LNCS*, pages 87–98. Springer Berlin / Heidelberg, 2008.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1:11–33, January 2004.

[3] D. Brugali and P. Scandurra. Component-based robotic engineering (Part I). *IEEE Robotics and Automation Magazine*, pages 84–96, Dec 2009.

[4] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proc. of the 7th Intl. Workshop on Component-Oriented Programming (WCOP)*, 2002.

[5] H. Bruyninckx, P. Soetens, and B. Koninckx. The real-time motion control core of the Orocos project. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, volume 2, pages 2766–2771, Sep 2003.

[6] A. Bucchiarone, H. Muccini, and P. Pelliccione. Architecting fault-tolerant component-based systems: from requirements to testing. *Electronic Notes in Theoretical Computer Science*, 168:77–90, 2007.

[7] J. Cheesman and J. Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley, 2000.

[8] P. Clements. A survey of architecture description languages. In *Software Specification and Design, 1996., Proc. of the 8th Intl. Workshop on*, pages 16–25, Mar 1996.

[9] P. A. de C. Guerra, C. M. F. Rubira, R. de Lemos, C. Guerra, C. Mary, and F. Rubira. An idealized fault-tolerant architectural component. In *Proc. of the 24th Intl. Conf. on Software Engineering - Workshop on Architecting Dependable Systems*, May 2002.

[10] W. Dunn. Designing safety-critical computer systems. *IEEE Computer*, 36(11):40 – 46, Nov 2003.

[11] M. C. Elder. *Fault tolerance in critical information systems*. PhD thesis, University of Virginia, 2001.

[12] B. Fei, W. S. Ng, S. Chauhan, and C. K. Kwoh. The safety issues of medical robotics. *Reliability Engineering & System Safety*, 73(2):183–192, 2001.

[13] E. Garcia, M. Jimenez, P. De Santos, and M. Armada. The evolution of robotics research. *IEEE Robotics and Automation Magazine*, 14(1):90–103, Mar 2007.

[14] B. P. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *Proc. Intl. Conf. on Advanced Robotics (ICAR)*, pages 317–323, 2003.

[15] M. Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, Jan-Feb 2004.

[16] Johns Hopkins University. The *cisst* libraries. `http://www.cisst.org/cisst`.

[17] M. Y. Jung, A. Deguet, and P. Kazanzides. A component-based architecture for flexible integration of robotic systems. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2010.

[18] B. Kaiser, P. Liggesmeyer, and O. Mäckel. A new component concept for fault trees. In *Proc. of the 8th Australian workshop on Safety critical systems and software - Volume 33*, SCS '03, pages 37–46. Australian Computer Society, Inc., 2003.

[19] A. Kapoor, A. Deguet, and P. Kazanzides. Software components and frameworks for medical robot control. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 3813–3818, May 2006.

[20] P. Kazanzides, A. Deguet, and A. Kapoor. An architecture for safe and efficient multi-threaded robot software. In *IEEE Conf. on Technologies for Practical Robot Applications (TePRA)*, pages 89–93, Nov 2008.

[21] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 9:25–40, 1989.

[22] S. Kulkarni. *Component Based Design of Fault-Tolerance*. PhD thesis, Ohio State University, 1999.

[23] J.-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proc. of the 25th Intl. Symp. on Fault-Tolerant Computing*, page 2, Jun 1995.

[24] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, Jul 1990.

[25] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. Software Eng.*, 33:709–724, 2007.

[26] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[27] M. Lipow. Prediction of software failures. *Journal of Systems and Software*, 1:71–75, 1979-1980.

[28] A. Makarenko, A. Brooks, and T. Kaupp. Orca: Components for robotics. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), Workshop on Robotic Standardization*, Dec 2006.

[29] B. Marick. A survey of software fault surveys. Technical Report UIU CD CS-R-90-1651, University of Illinois at Urbana-Champaign, Dec 1990.

[30] J. A. McDermid. Computing tomorrow. chapter Engineering safety-critical systems, pages 217–245. Cambridge University Press, New York, NY, USA, 1996.

[31] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Eternal–component-based framework for transparent fault-tolerant corba. *Software: Practice and Experience*, 32(8):771–788, 2002.

[32] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P. O. Müller, C. Zeidler, T. Genssler, and R. v. d. Born. A component model for field devices. In *Proc. of the IFIP/ACM Working Conf. on Component Deployment*, pages 200–209, London, UK, 2002. Springer-Verlag.

[33] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: architecture for component trading and dynamic updating. In *Proc. of the 4th Intl. Conf. on Configurable Distributed Systems*, pages 43–51, May 1998.

[34] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[35] B. Randell. System structure for software fault tolerance. In *Proc. of the Intl. Conf. on Reliable Software*, pages 437–449, New York, NY, USA, 1975. ACM.

[36] D. C. Schmidt and S. Vinoski. Object interconnections: The corba component model: Part 1, evolving towards component middleware. *C/C++ Users Journal*, Feb. 2004.

[37] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, Dec 1990.

[38] M. Shapiro. Structure and encapsulation in distributed systems : the proxy principle. In *Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 198–204, 1986.

[39] Sun Microsystems. *JavaBeans Specification*, 2007. `http://www.oracle.com/technetwork/java/docs-135218.html`.

[40] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[41] S. Tambe, A. Dabholkar, and A. Gokhale. Fault-tolerance for component-based systems - an automated middleware specialization approach. In *Proc. of the IEEE Intl. Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC, pages 47–54, 2009.

[42] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, Mar 2000.

[43] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. In *IEEE Aerospace Conf.*, volume 1, pages 121–132, Mar 2001.

[44] A. Wigley, S. Wheelwright, R. Burbidge, R. MacLeod, and M. Sutton. *Microsoft .NET Compact Framework (Core Reference)*. Microsoft Press, 2003.

[45] Y. Wu, D. Pan, and M.-H. Chen. Techniques for testing component-based software. In *Proc. of the 7th IEEE Intl. Conf. on Engineering of Complex Computer Systems*, pages 222–232, 2001.