State-based Safety of Component-based Medical and Surgical Robot Systems

by

Min Yang Jung

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

May, 2015

Abstract

Safety has not received sufficient attention in the medical robotics community despite a consensus of its paramount importance and the pioneering work in the early 90s. Partly because of its emergent and non-functional characteristics, it is challenging to capture and represent the design of safety features in a consistent, structured manner. In addition, significant engineering efforts are required in practice when designing and developing medical robot systems with safety. Still, academic researchers in medical robotics have to deal with safety to perform clinical studies.

This dissertation presents the concept, model and architecture to reformulate safety as a visible, reusable, and verifiable property, rather than an embedded, hard-to-reuse, and hard-to-test property that is tightly coupled with the system. The concept enables reuse and structured understanding of the design of safety features, and the model allows the system designers to explicitly define and capture the run-time status of component-based systems with support for error propagation. The architecture leverages the benefits of the concept and the model by decomposing safety features into reusable mechanisms and configurable specifications. We show the concept and feasibility of the proposed methods by building an open source framework that aims to facilitate research and development of safety systems of medical robots. Using the cisst component-based framework, we empirically evaluate the proposed methods by applying the developed framework to two research systems – one based on a commercial robot system for orthopedic surgery and another robot soon to be clinically applied for manipulation of flexible endoscopes.

Advisor:

Peter Kazanzides PhD, Research Professor
Department of Computer Science, The Johns Hopkins University

Readers:

Russell H. Taylor PhD, Professor
Department of Computer Science, The Johns Hopkins University

Scott Smith PhD, Professor
Department of Computer Science, The Johns Hopkins University

Acknowledgments

For the past years in graduate school, I have learned and changed a lot in many ways, both professionaly and personally. A number of individuals have directly or indirectly helped me throughout this long journey and deserve my sincere gratitude. Without their help, it would have been not possible for me to come this far, indeed.

First and foremost, I would like to thank my advisor, Dr. Peter Kazanzides, for his support and guidance throughout my PhD life. His door was always wide open and the work contained in this dissertation would not have been possible without his help, insight, encouragement, and generosity. Professionally, his enthusiasm and perfectionism have profoundly affected the way I conduct my scientific research and engineering work in general. Also, it was an amazing and rare experience, as a graduate student, to be able to work on a commercial-level system while directly learning from someone who had developed the pioneering commercial products in industry for more than a decade. Now I understand how multi-threads were achieved under DOS and thus appreciate how easily we can use multi-threads in the modern operating systems. Also, I thank him for understanding my personal situation where I had to become a Southwest A-List customer and, more recently, a weekly driver on I-95.

I would also like to thank Dr. Russell Taylor. Through him and his classes, I was introduced to the fields of medical robotics and computer-integrated surgery. His deep and broad expertise in various areas of medical robotics, as well as his endless passion for research, have very much motivated me over the years. I am incredibly grateful to Dr. Taylor for his insight and comments on my work on safety.

I am also very grateful to Dr. Scott Smith for being on my thesis committee and for taking the time to read my dissertation. His comments and advice from the early days of this work have been incredibly helpful. It was fortunate for me to have comments from a different perspective with deep expertise in the area of programming languages and systems.

My research on safety as well as a large portion of this dissertation were made possible with generous support from THINK Surgical (formerly, Curexo Technology Corporation). I would also like to acknowledge NIH R01 EB007969 for its support in the interim.

Through the long journey towards my PhD, my friends and colleagues made the past years at LCSR/CISST-ERC a very enjoyable, memorable, and pleasant experience with their open, warm mind and friendship. Thus, I would like to thank: Tian Xia, Marcin Balicki, Xingchi He, Babak Matinfar, Paul Thienphrapa, Simon Leonard, Anton Deguet, Wen Liu, Sue Sue U-Thainual, Lye Lin Lock, Seth Billings, Chin-Hsing Kuo, Zihan Chen, Tutkun Sen, Yixin Gao, Berk Gonenc, Carol Reiley, Robert Grupp, Jonathan Bohren, Anand Malpani, Shahin Sefati, and Hongliang Ren. I am also grateful for my Korean friends: Jin Kyun Park, Jonghyun Lim, Junghoon Lee, Jinseob Kim, Wooram Park, Hyun Jae Kang,

ACKNOWLEDGMENTS

Byungyu Ahn, Sungmin Kim, Jeongil Ko, Yoon Su Kim, and Doyoung Chang, with my very special thanks to Nathan Cho and his family (Annabelle, Stella, and Sharon). I also thank the JHU Computer Science department and LCSR staff members who have always been of great help. They are: Cathy Thorton, Debbie Deford, Laura Graham, Emily Evans, Elisa Ahmanson, Alison Morrow, and Jamie Meehan.

Finally, I would like to thank my families for their support, endless love, and being with me throughout the challenging years: Mom, Dad, Hyun Yang (Jamie and Soyoon), Yu Jin, Mother-in-law, Father-in-law, Sun Woo, Sungyoung, and lastly, my eternal companion of the life, lovely Sunmee.

Contents

List of Figures

List of Tables

List of Codes

Chapter 1

Introduction

For the past three decades since the first reported robotic surgical procedure in 1985 (Kwoh *et al.*, 1985[1]), a variety of medical robot systems have been developed both in academia and industry. These robot systems have helped to improve clinical results and enabled surgical procedures and techniques that would not have been possible otherwise in various areas. Prominent examples of such application areas include neurosurgery, orthopaedic surgery, laparoscopic surgery, radiosurgery, and telesurgery. Commercial products are also being used in the modern operating room or interventional suite.[2] For example, the da Vinci system (Intuitive Surgical, Inc., Sunnyvale, CA, USA) was used in 80% of radical prostatectomies performed in the U.S. in 2008.[2]

Broadly, research in medical robotics comprises three areas (Taylor, 2006[3]): (1) *modeling and analysis* of images, patient anatomy, and surgical plans, (2) *interface technology* relating the "virtual reality" of computer models to the "actual reality" of the patient, operating room, and surgical staff, and (3) *systems science* permitting these components to be combined in a modular and robust manner with safe and predictable performance. So far, much of the advancement of the domain has been driven by innovations and improvements in the first two areas, whereas the third area – especially safety – has not received as much attention as the other two areas. However, the importance of the systems science is rapidly increasing as the size and complexity of medical robot systems significantly increase to achieve challenging medical functionality requirements, while at the same time meeting non-functional requirements such as safety, performance, and reliability.

Within medical robotics, safety has been perceived as the crucial property of medical robot systems. The pioneering work with focus on safety is found in the literature as early as 1991 (Taylor *et al.*, 1991[4]), as well as other early work on safety (e.g., Kazanzides, 1992;[5] Davies, 1993[6] and 1996;[7] Taylor, 1996[8]). There exist other works that presented software-based approaches to safety (e.g., system architecture and state-based approach by Kazanzides *et al.*, 1992;[5] verification and validation-based software design method by Fei *et al.*, 2001[9]). Most of these works addressed safety issues in an application- and system-specific manner, or in the form of general safety design guidelines.

Despite these early efforts and the recognition of the importance of safety, system designers still find the following two issues challenging: (1) *inability to capture the knowledge and experience with safety of medical robot systems*, and (2) *significant amount of engineering effort* to build medical robot systems with safety. Currently, there is no safety standard that specifically governs the design of medical robot systems; rather, developers conform to existing medical device standards, such as IEC-60601 and IEC-62304. Similarly, medical robots are subject to the same regulatory approval processes as other medical devices. By definition, medical robots are examples of safety-critical systems because human lives depend on their correct operation. As in other safety-critical domains, developers must invest significant engineering effort to ensure that every device they design is safe and meets regulatory requirements. In particular, such considerable engineering effort is unlikely available to academic researchers in practice. Yet, they have to design and develop medical robot systems to be safe in order to perform clinical trials. This obviously leads to a desire to capture the "best practices" that can be reused when designing new systems, thereby quickly building prototypes of safe medical robot systems with less engineering efforts.

Safety has been one of the active research topics in various areas and disciplines, and has been investigated extensively outside the medical robotics domain. Examples include the traditional safety-critical application systems domain, the dependable computing domain, and the software engineering domain that includes safety engineering and component-based software engineering. The existing body of work from these domains could provide substantial references and guidance for safety research in medical robotics. However, they have to be carefully adopted; safety of medical robot systems is distinctly different from that of other areas because: (1) humans must be present in the robot's workspace, (2) one of the humans is usually anesthetized and cannot escape, and (3) the robot may be holding a sharp instrument and be required to "injure" the human to perform the surgical intervention. Thus, these domain characteristics should be considered when adopting prior knowledge

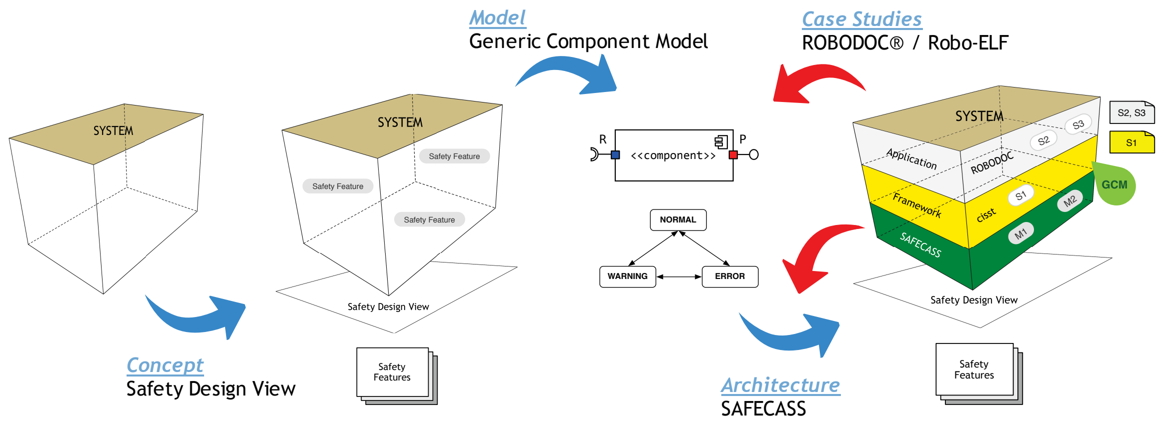


Figure 1.1: Conceptual flow of this dissertation

and experience from outside medical robotics.

1.1 Proposed Solutions

This dissertation addresses the aforementioned challenges by essentially improving *reusability* and *testability* of safety. These improvements would facilitate reuse of prior experience and knowledge on safety while reducing engineering effort to build safe medical robot systems. Our proposed methods to improve these properties are comprised of four key elements: *concept*, *model*, *architecture*, and *case study*. These elements are the key topics of each chapter (Chapter 3-6) and each element is built upon what comes before.

Fig. 1.1 conceptually depicts the progression through the dissertation. Starting from the concept, we progressively develop our methods through the model and the architecture, and validate the methods using the case study. The high-level description of each element is as follows:

- **Concept** (Chapter 3): The concept represents the *Safety Design View (SDV)*, a conceptual framework that defines the design space of safety features of medical robot systems. This design space identifies essential components of safety features, while at the same time capturing the system designer's design decisions on the deployment options.
- **Model** (Chapter 4): The model refers to the *Generic Component Model (GCM)*, an abstract component model that can explicitly represent the operational status of component-based systems using its state-based semantics without relying on particular component models.
- **Architecture** (Chapter 5): The architecture is a safety-oriented layered architecture that enables reuse of safety features by decomposing a safety feature into a reusable mechanism and a configurable specification. This architecture is used to implement a run-time environment for the GCM, called the *Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)*.
- **Case Studies** (Chapter 6): The case studies demonstrate how to apply the proposed methods to a set of safety features of two existing medical robot systems: a commercial medical robot system for orthopaedic surgery (the ROBODOC® System), and a research system for minimally invasive endolaryngeal surgery (the Robo-ELF System). We empirically evaluate and validate our proposed methods through these two case studies.

To illustrate our methods, we introduce an example of a simple safety feature that checks

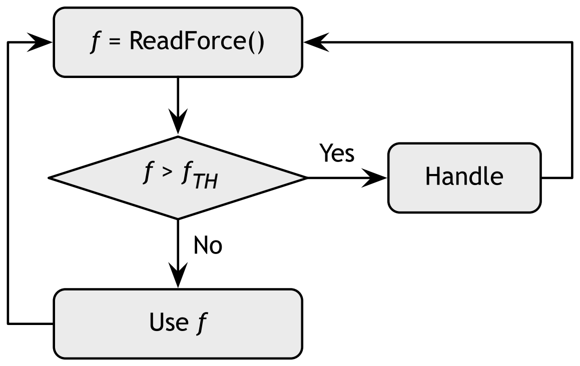


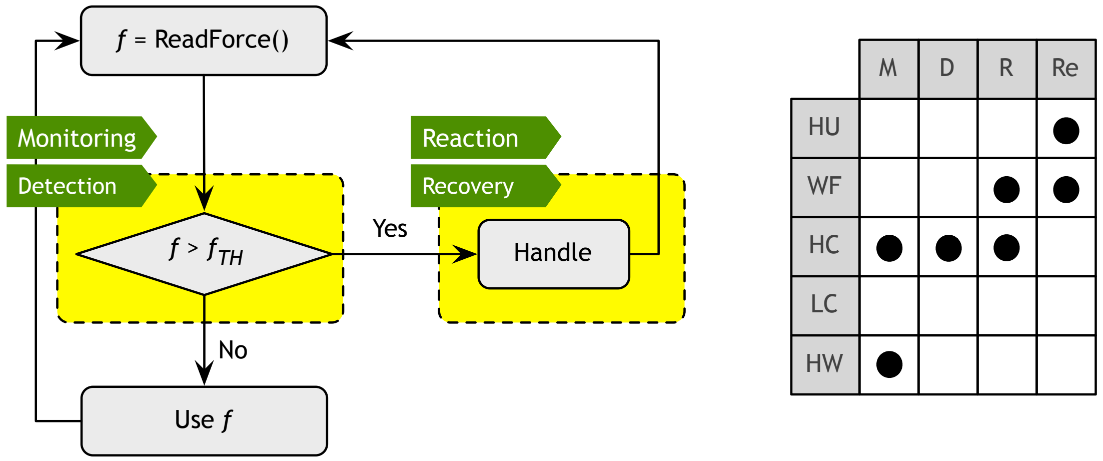
Figure 1.2: Simple example: Design of force sensor-based safety feature. Force feedback, f, is read from the force sensor and is checked against the pre-defined threshold, f_{TH}. Typically, an error event is generated if f exceeds f_{TH}, and the error is handled accordingly.

force feedback from a force sensor and generates an error event when the force feedback exceeds a pre-defined threshold. Fig. 1.2 shows the typical design of this safety feature. Despite its simple design, this force sensor-based safety feature is one of the most widely used safety features in the medical robotics domain, as described in Chapter 2.

When developing this safety feature, the system designers are likely to encounter practical issues in terms of testability and repeatability. For example, it would be extremely difficult to manually test what happens if f is *exactly* equal to f_{TH}, or becomes $f_{TH} \pm 0.0001$ (boundary conditions). Although such test conditions may be simulated once or twice, it is not practically feasible to repeatedly perform regression testing over time, possibly with varying f_{TH} values (e.g., to find the optimal threshold value that leads to the least false positives). Furthermore, it is not possible to repeatedly test sensor failure cases that lead to permanent damage, such as internal hardware failure (e.g., damaged sensor), external sensor cabling issues (e.g., physically disconnected cables), and irregular magnetic or electric interference (e.g., electric shock). Although these issues can be lessened with the help of additional test code or a dedicated testing library, they tend to be application- or system-specific and thus are not likely reusable across systems. Thus, by applying our methods, we aim to improve testability of this system.

To begin with, we apply our **concept** – the Safety Design View (Chapter 3) – to this safety feature for structured understanding of the design of it. We first identify the four essential components of run-time safety mechanisms: *monitoring*, *detection*, *reaction*, and *recovery*. These four components are defined by the *Mechanism View* (Sec. 3.2.1). The comparison of f against f_{TH} represents monitoring and detection, and the handling of excessive force feedback indicates reaction and recovery. This is denoted by the green tags in Fig. 1.3. Next, we identify where each of these components is implemented in the system. For consistent and systematic representation, the SDV defines a canonical architecture of medical robot systems, which is called the *System View* (Sec. 3.2.2). It is essentially a layered architecture and is comprised of four layers: *Hardware*, *Control*, *Workflow*, and *Human* layers. For demonstration purposes, we assume that the system designer chose a particular deployment option, as shown on the right side of Fig. 1.3. The SDV is a two-dimensional plane formed by combining the Mechanism View on the horizontal axis with the System View on the vertical axis (Sec. 3.3). In this manner, the SDV allows us to capture and describe the design of safety features in an explicit, consistent, and structured manner, thereby facilitating sharing of safety experience and knowledge.

Next, we apply our **model** – the Generic Component Model (Chapter 4) – to the example. The GCM describes the run-time status of component-based systems using its state-based



	M	D	R	Re
HU				●
WF			●	●
HC	●	●	●	
LC				
HW	●			

Figure 1.3: Simple example with the Safety Design View (concept). On the left side, the four essential components of run-time safety mechanisms are tagged in green. On the right side, the Safety Design View presents system designer's decisions on the deployment options, i.e., which elements are deployed to which layers of the system.

semantics (Sec. 4.3). Its three essential elements are *states*, *events*, and *filters*. The three abstract states – *Normal*, *Warning*, and *Error* – and transitions between the states define the state machine. A set of state machines collectively represents the current operational status of the system. The state transitions are initiated by events, which are generated by filters. The GCM also defines a mechanism for error propagation (Sec. 4.3.7) that systematically notifies other components of the occurrences of errors.

Fig. 1.4 depicts the new design of the example after applying the GCM, where the major changes are represented in the yellow box. The force threshold check is wrapped as a filter and an error event is explicitly defined as `E_FORCE_ERROR`. This event is generated when excessive force feedback is detected and initiates a state transition from Normal to Error, and results in error propagation to other connected components. The state-based semantics of the GCM defines this sequence of actions. It should be noted that the GCM is defined at

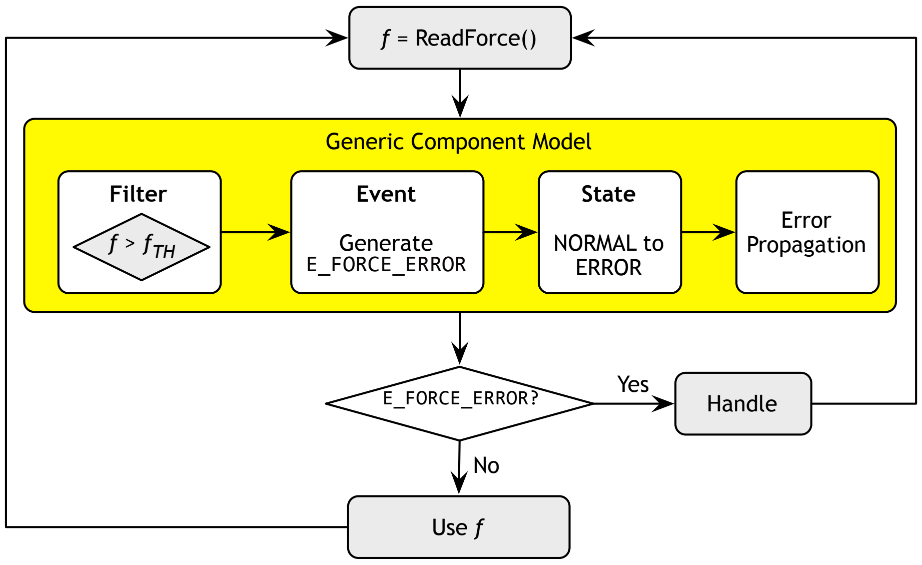


Figure 1.4: Simple example with the Generic Component Model (model). The design of existing safety features are represented in terms of the three elements of the state-based semantics of the Generic Component Model: filters, events, and states. They are represented in the yellow box.

the model level using the minimal structural elements (components and interfaces). That is, it does not contain any code-level specifics and is not dependent on a particular component model.

For these reasons, the GCM is neither executable nor verifiable at run-time. Thus, we provide a run-time environment for the GCM, called the Safety Architecture for Engineering Computer-Assisted Surgical Systems (Chapter 5). Based on the domain characteristics of medical robotics (Sec. 5.3), the design requirements of the run-time environment are identified (Sec. 5.4). Our approaches to achieve the design requirements are the *framework independence* and *safety design decomposition* (Sec. 5.5). Considering these elements all together, we design and implement a safety-oriented layered architecture, called the SAFECASS-based **architecture** (Sec. 5.6). It provides a set of tools that can facilitate the system development process.

Now that the GCM becomes executable within the SAFECASS, we can actually execute the GCM-enabled example. Fig. 1.5 illustrates this change. Note that the yellow box now represents the SAFECASS, instead of the GCM. In this design, one key change is that we now maintain the specifications of the GCM elements (i.e., events and filters) in a separate text file (the JSON format). Such SAFECASS artifacts are used to deploy safety features into the system at run-time and allow us to easily change the system behavior by just modifying parameters of this artifact. For example, the following code snippet can be used as a SAFECASS artifact for the simple force sensor-based safety feature (some parameters are arbitrarily chosen or simplified for demonstration purposes; see Chapter 5 for more details):

Code 1.1: Example of JSON specification for the force sensor-based safety feature

```
{
    "component": "Force",
    "event": [
        {   "name"            : "E_FORCE_WARNING",
            "severity"        : 10,
            "state_transition": [ "N2W" ]
        },
        {   "name"            : "/E_FORCE_WARNING",
            "severity"        : 10,
            "state_transition": [ "W2N" ]
        },
        {   "name"            : "E_FORCE_ERROR",
            "severity"        : 20,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/E_FORCE_ERROR",
            "severity"        : 20,
            "state_transition": [ "E2N", "W2N" ]
        },
        {   "name"            : "E_SENSOR_FAILURE",
            "severity"        : 30,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/E_SENSOR_FAILURE",
            "severity"        : 30,
            "state_transition": [ "E2N", "W2N" ]
        }
    ],
    "filter" : [
        {   // common fields
            "class_name"           : "FilterThreshold",
            "target"               : {
                "type"             : "s_A",
                "component"        : "Force"
            },
            "type"                 : "ACTIVE",
            // fields specific to threshold filter
            "argument" : {
                "input_signal"     : "Force",
                "threshold"        : 5.0,
                "tolerance"        : 0.5,
                "event_onset"      : "E_FORCE_ERROR",
                "event_completion": "/E_FORCE_ERROR"
            }
        }
    ],
    "service" : [
        {   "name"     : "InterfaceProvidingForce",
            "dependency" : {
                "s_R" : [ ],
                "s_A" : true,
                "s_F" : true
            }
        }
    ]
}
```

To deploy the safety features defined in this artifact, it is necessary to modify the code-level implementation. Assuming one particular implementation for the simple force sensor-based safety feature, the following code snippets show the original and new code structure side-by-side (the variables and functions that begin with a capital letter represent class member variables and functions; see Chapter 6 for more details):

Before:

```
void Force::Run(void)
{
  f = ReadForce();

  if (f > f_threshold) {
    // Handle excessive force
  } else {
    // Update/use latest reading f
  }
}
```

After:

```
void Force::RunNormal(void)
{
  f = ReadForce();

  // Update/use latest reading f

  UpdateStatus();
}

#define ON_EVENT(evt_name)\
  if (e == Coordinator->GetEvent(evt_name))

void Force::RunWarning(const SC::Event * e)
{
  ON_EVENT("E_FORCE_WARNING") {...}

  RunNormal();
}

void Force::RunError(const SC::Event * e)
{
  ON_EVENT("E_SENSOR_FAILURE") {...}
  ON_EVENT("E_FORCE_ERROR") {
    // Handle excessive force
  }

  UpdateStatus();
}

void Force::UpdateStatus(void)
{
  if (sensor_failure_detected) {
    Coordinator->GenerateEvent(
        "E_SENSOR_FAILURE");
  }
}
```

As Chapter 5 and 6 describe in detail, these changes not only allow us to *read* key information of the system (e.g., deployed filters, outstanding events, current states), but also to *write* data into the system at run-time (e.g., fault injection, event generation) using the services and tools that the SAFECASS provides. The ability to fully access the key data of the system at run-time is crucial to improving testability of safety.

The benefits of our proposed methods in terms of reusability and testability are highlighted when we compare the original design of the example in Fig. 1.2 and the SAFECASS-enabled design in Fig. 1.5. These benefits include:

- **Improved testability**: The abilities to read key data from the system and to write test data to the system enable fault injection and event generation at *run-time* without code compilation or system restart. These features are provided in two different forms: the interactive standalone utility (the *console* utility; Sec. 5.6.7.1) and the application programming interfaces (APIs).

- **Configurable, flexible, and traceable safety specification**: The SAFECASS arti-

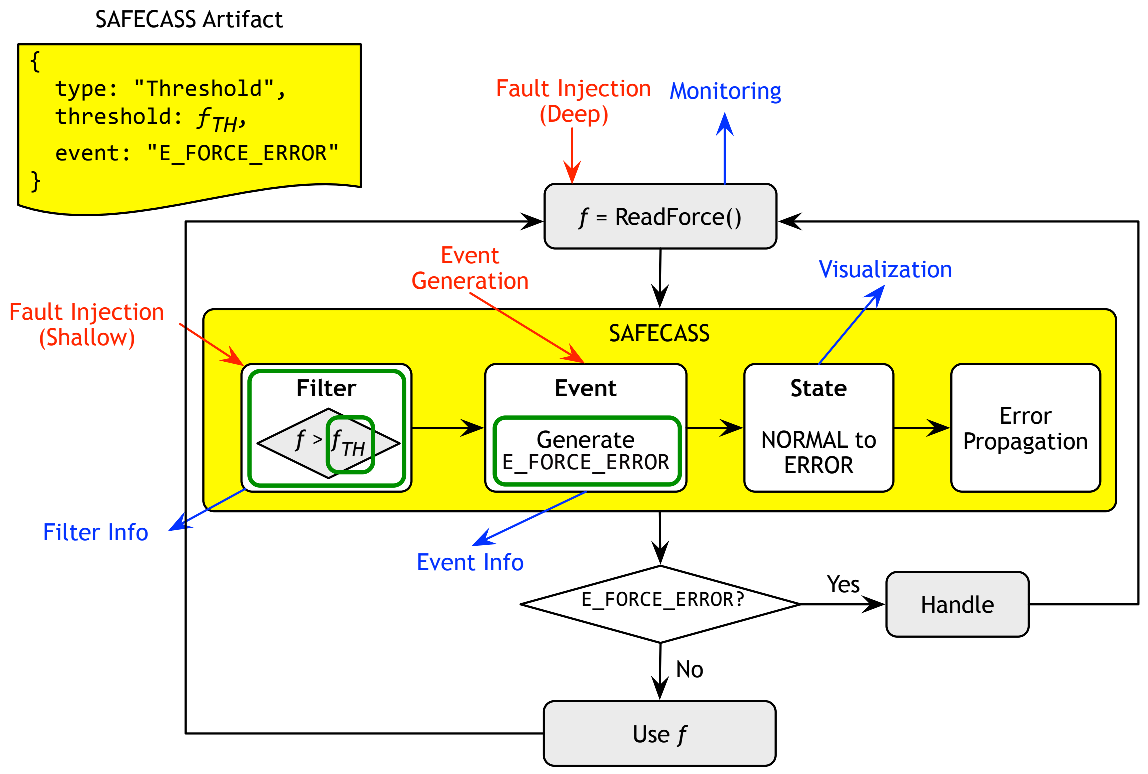


Figure 1.5: Simple example with SAFECASS (architecture). Read and write operations are represented in blue and red, respectively. The green boxes indicate the parameters that the SAFECASS artifacts define and thus can be easily modified to change the design of the safety feature.

facts contain safety specifications used to deploy safety features to the system. Because safety specifications are decoupled from code and are separately maintained, it becomes possible to easily change the design of safety features, to track the changes of safety specifications over time (e.g., using version control software), and to perform automated testing based on specifications (e.g., unit testing, regression testing).

- **Reusable safety mechanisms**: The state-based semantics of the GCM and the SAFECASS are designed to be application-, framework-, and component model-independent. Once their correctness is verified (via, for example, formal methods), they can be reused later without having to verify their correctness again, thereby reducing engineering effort when building new systems with safety.

Through our **case studies** (Chapter 6), we empirically validate these benefits by applying the proposed methods to two existing medical robot systems: (1) a commercial robot system for orthopaedic surgery, called the ROBODOC® System (Sec. 6.2), and (2) a research robot system for minimally invasive laryngeal surgery, called the Robotic Endo-Laryngeal Flexible (Robo-ELF) Scope System (Sec. 6.3). The commercial ROBODOC system (THINK Surgical, Inc., Fremont, CA, USA) has a solid set of safety features that have obtained the U.S. Food and Drug Administration (FDA) approval and European Union (EU) CE marking, and has been in clinical use since 1992. The Robo-ELF includes several safety features and received JHU Institutional Review Board (IRB) approval for clinical use after the FDA determined that an Investigational Device Exemption (IDE) was not required. With our full access to the source code of both systems, we apply the SAFECASS to a set of safety features of those systems at the code-level, while maximally preserving their original design. The idea is to evaluate and validate the benefits and effectiveness of the proposed methods by comparing the original design with the new, SAFECASS-enabled design in terms of testability and reusability.

For each robot system, we perform the safety design refactoring process with the following steps:

1. **Understanding of the system**: We provide a brief introduction to each robot system that includes a target surgical application, system components (hardware and software),

and system design requirements. We also represent the system design using the System View (Sec. 3.2.2) to better understand the system design rationale.

2. **Identification of safety features**: Based on the design documents accessible to us (e.g., published academic literature), we present a summary of safety features of the system. The safety features of the ROBODOC and the Robo-ELF are summarized in Secs. 6.2.1.2 and 6.3.1, respectively.
3. **Application of SAFECASS**: We integrate SAFECASS with each system in two steps: (1) defining safety specification in the JSON[10] format, and (2) code-level design refactoring.

In our case studies, a subset of identified safety features of the ROBODOC is selected for illustration purposes, whereas the entire set of safety features of the Robo-ELF is presented. Although there are variations on how to modify the design of safety features of each robot system, the safety specifications are similar to the JSON example shown in Code listing 1.1, with code-level changes similar to the ones presented above.

1.2 Thesis Statement

A structured understanding of safety designs and a safety-oriented layered architecture with a state-based semantics can facilitate research and development of component-based medical robot systems, thereby enabling sharing and reuse of knowledge and experience on safety.

1.3 Thesis Contributions

This dissertation presents our methods – the concept, the model, and the architecture – that reformulates safety as a visible, reusable, and verifiable property, rather than an invisible, hard-to-reuse, and hard-to-test property that is deeply embedded in the system. The methods address challenges in safety research by defining a new perspective on safety and by providing a software environment that facilitates the design and development process of component-based medical and surgical robot systems. Each chapter describes contributions of the chapter in more detail, but we provide a summary of the major contributions reported in this dissertation here:

Safety Design View: Developed a conceptual model that captures the design rationale of safety features and the system designer's decisions in a structured, systematic manner.

- Identification of the four essential components of safety features
- Identification of the canonical architecture of medical robot systems that can systematically capture the system designer's decisions on deployment options
- Definition of the design space of safety features in medical robotics

Generic Component Model: Proposed a generic model, based on the dependability formalism, with component model-independent semantics and a programming model that supports error propagation.

- Proposal of a generic model that can represent the operational status of component-based robot systems at run-time in an explicit, systematic, and structured manner
- Design of a state-based semantics that enables error propagation across the component boundary
- Design of an event mechanism with the concepts of onset and completion events, outstanding events, and rules for prioritizing events

Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS): Designed a safety-oriented layered architecture that provides a run-time environment for the Generic Component Model.

- Proposal of a safety-oriented architecture for component-based robot systems
- Implementation of a run-time environment for the Generic Component Model using the SAFECASS-based architecture
- Development of the *cisst* framework extensions for the SAFECASS, including *cisst*-specific safety features that improve the system development process in practice

Case Studies: ROBODOC® and Robo-ELF: Empirical evaluation of the proposed methods and architecture using safety features of a commercial surgical robot system for orthopaedic surgery (ROBODOC) and a research system for minimally invasive endolaryngeal surgery (Robo-ELF).

- Empirical validation that the design and implementation of the SAFECASS meets its design requirements

- Empirical validation of the effectiveness and applicability of the SAFECASS-based approach using safety features of two surgical robot systems, one for orthopaedic surgery and the other for endolaryngeal surgery

1.4 Organization

The body chapters of this dissertation follow the flow of the key elements of our proposed methods, i.e., *concept*, *model*, *architecture*, and *case study*. Fig. 1.6 shows the overall flow of this dissertation along with the key topics and the contributions of each chapter.

Chapter 2 presents a comprehensive review on safety and safety-related topics in three parts. The first part provides a brief overview on safety *outside the robotics* domain, which include the dependable computing domain and the safety engineering domain. Topics covered include fault categories and classification, fault detection and diagnosis, fault tolerance, safety engineering, and component-based software engineering. The second part reviews safety research *in the robotics* domain, where safety is mostly considered in the context of the physical human robot interaction. The third part presents an extensive survey on safety research *within the medical robotics* domain, along with a set of domain-specific safety features, medical device standards, and safety-related activities.

Chapter 3 proposes the Safety Design View (SDV), a conceptual framework that can

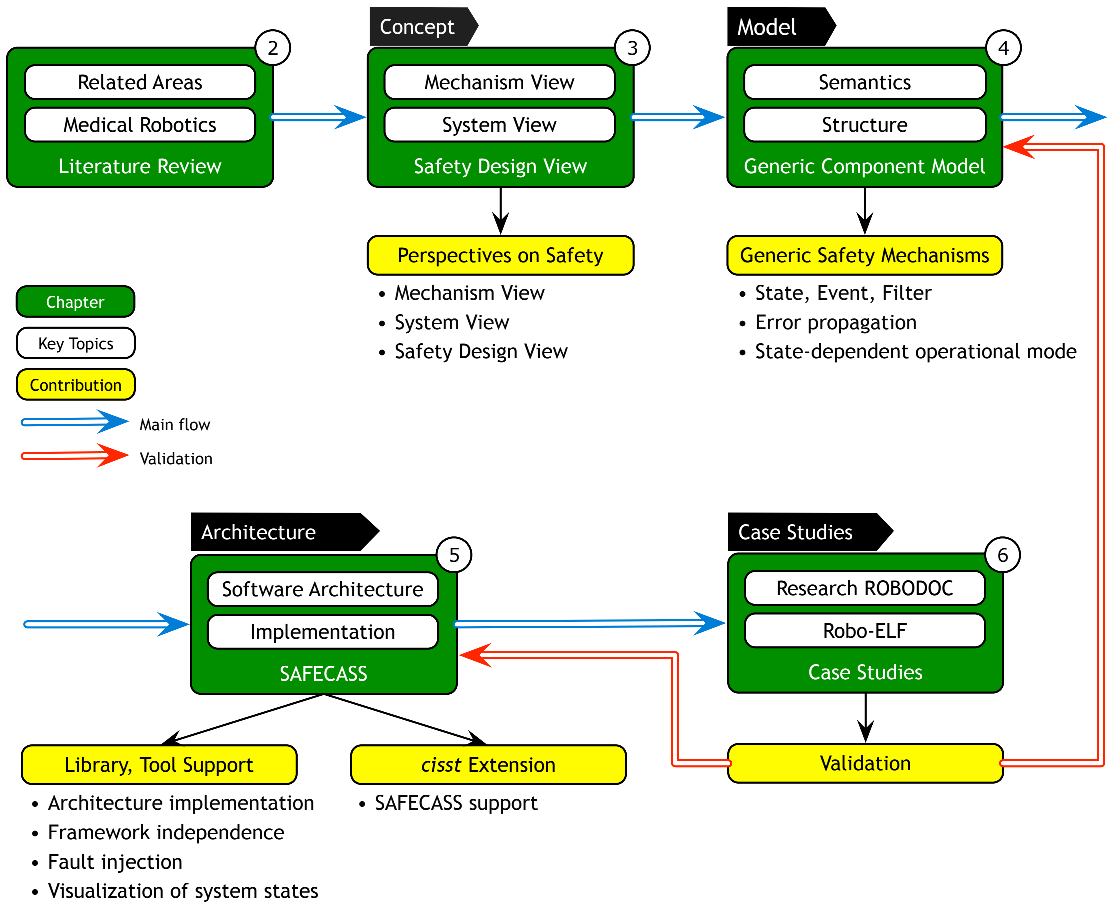


Figure 1.6: Flow of this dissertation with contributions of each chapter. The numbers on the top right corners of the green boxes represent chapters.

capture and describe both the design-time and the run-time characteristics of safety features of medical robot systems in a systematic and structured manner. SDV identifies essential elements and enabling components of safety features of medical robot systems, thereby allowing us to define and describe safety features of medical robot systems in a generic, consistent, and structured manner.

Chapter 4 describes the Generic Component Model (GCM), an abstract component model with minimal structural elements. The GCM is generic enough to be specialized for other component models, yet it is expressive enough to represent the complete description of the system status without relying on a particular component model. Its state-based semantics can explicitly capture and describe the operational status of component-based robot systems at run-time, and supports error propagation across the component boundary.

Chapter 5 presents the design and architecture of the Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS), which provides a run-time environment for the GCM. Starting from the four design requirements that consider the domain characteristics, this chapter proposes a safety-oriented layered architecture called the SAFECASS-based architecture. It also presents the detailed design and implementation of SAFECASS in terms of the essential elements of the GCM. By actually building a software framework, this chapter shows that it is possible to build a run-time environment for the GCM that meets the four design requirements.

Chapter 6 illustrates two case studies with which we empirically evaluate the proposed methods and architecture. In these case studies, we demonstrate in detail how the SAFECASS and the SAFECASS-based architecture can be applied to existing systems to improve the design of safety features and to benefit from the proposed methods. In this chapter, we use the *cisst* component-based framework and two surgical robot systems, one for orthopaedic surgery (the ROBODOC System) and another one for minimally invasive endolaryngeal surgery (the Robo-ELF System). Based on this empirical evaluation, we show that the current design and implementation of SAFECASS effectively achieves its four design requirements, thereby facilitating research and development of safety systems for medical robots.

Chapter 2

Literature Review

Since an industrial robot was first used for a CT-aided stereotactic neurosurgery as a surgical tool holder in 1985,[1] a variety of medical and surgical robot systems have been developed both in academia and in industry. In the medical robotics domain, safety has been considered as the most crucial property from the beginning, and the pioneering work with focus on safety is found in the literature as early as 1990 (Taylor *et al.*, 1990[11]). However, safety as a research problem has not received much attention yet, and prior works on safety in the medical robotics domain approached safety in an application- and system-specific manner.

In contrast, safety has been one of major research topics in various areas and disciplines outside the medical robotics domain, such as the traditional safety-critical application systems domain, the dependable computing domain, and the software engineering domain. A system is *safety-critical* if the failure of the system can lead to consequences that are determined to be unacceptable.[12] Examples of such systems include medical devices, aerospace systems, automotive systems, nuclear power plant control systems, rail-way control systems, air traffic control systems, and weapon control systems. Although the domain characteristics of medical robotics are different from those of other domains[i], the existing body of work in those areas could be substantial references for safety research in medical robotics.

This chapter consists of three parts. The first part explores prior works on safety *outside the robotics* domain (Sec. 2.1). This part presents an overview of safety-related works in different domains, but does not attempt to provide an extensive survey. The second part reviews safety research *in the robotics* domain (Sec. 2.2), where safety is mostly considered in the context of the physical human robot interaction (pHRI). Lastly, the third part presents an extensive survey on safety research *within the medical robotics* domain (Sec. 2.3).

2.1 Outside Robotics

Safety is a system property[13] that requires a systems approach, and thus safety has been a research topic in various system- or safety-related areas. Among those areas outside medical robotics, this section highlights some of the prior works in three different areas: *dependable and secure computing* (Sec. 2.1.1), *software engineering* (Sec. 2.1.2), and *component-based software engineering* (Sec. 2.1.3).

[i]Refer to Sec. 5.3 for domain-specific characteristics of medical and surgical robot systems.

2.1.1 Dependable Computing

In the dependable and secure computing community, safety is considered as one of the attributes of *dependability*, an integrating concept that encompasses safety at a higher level. Based on the domain expertise, Avizienis *et al.* (2004) established the dependability formalism that extensively defines the basic concepts and taxonomy of dependable and secure computing.[14] According to this formalism, dependability of a system is defined as the ability to avoid service failures that are more frequent and more severe than is acceptable, and its *attributes* include availability, reliability, safety, integrity, and maintainability. Although this abstraction to define one integrating concept for these attributes appears to be somewhat debatable[ii], this dependability formalism and semantics have been widely adopted, as proved by the citation count of the paper being more than 3,200[iii].

The *threats* to dependability are *faults*, *errors*, and *failures*, and they successively form the *fundamental chain of dependability and security threats*. The starting point of this chain is faults, which are major threats to system dependability. It has been shown that the lack of error and fault handling coverage drastically limits the improvement of dependability. Although safety is not necessarily achieved by just preventing faults, errors, or failures, proper handling of them makes a system more dependable and thus improves system safety.

[ii] In safety engineering, there was an opinion that did not agree with the idea of abstraction for safety: *"However, attempts to integrate several qualities into one abstraction (like dependability, which has been proposed as a combined measure of reliability, safety, security, availability and just about every other quality) seem misguided. These global abstractions have only disadvantages, since they inhibit understanding and control."* (*Safeware*, Leveson 1995[15])

[iii] as of October 2014 from Google Scholar

This sections presents an overview on the following three topics in more details:

- Fault categories and classification (Sec. 2.1.1.1) : *"What is a fault and what types of faults can happen?"*
- Fault detection and diagnosis (Sec. 2.1.1.2) : *"How to detect faults and what needs to be identified?"*
- Fault tolerance (Sec. 2.1.1.3) : *"How to tolerate faults to keep the system running?"*

2.1.1.1 Fault Categories and Classification

Lipow (1979)[16] reported fault categories that occur during software development phases as logic, data handling, interface, data I/O, computational, database, data definition, and others. Marick (1995)[17] surveyed fault categories focusing on software (e.g., data handling, logic error, I/O failure) and showed quantitative summaries on relative occurrences of the faults. Compared to these software-oriented classifications, Avizienis *et al.* (2004)[14] presented an extensive and comprehensive classification of faults, which covers not only software-related faults, but also faults from various aspects of the system (e.g., hardware, environments, human-machine interface, human factors) across all the development phases. They classified faults according to the eight basic viewpoints (a.k.a., the *elementary fault* classes) and identified 31 *likely fault combinations* that are divided into three partially overlapping groups: development faults, physical faults, and interaction faults. In our prior work (Jung 2011[18]), we presented a hierarchical fault model for component-based software systems that can be used to design and analyze possible faults of component-based robotic systems in a

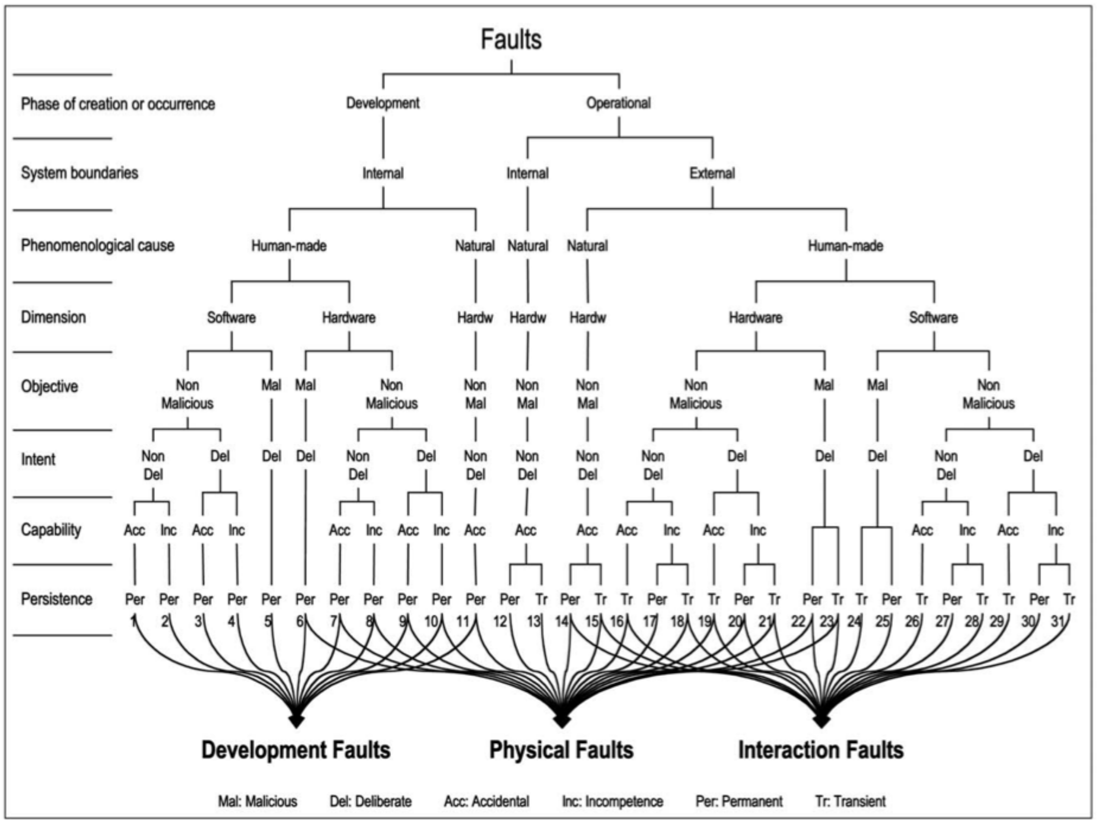


Figure 2.1: Classes of 31 likely fault combinations (from Avizienis (2004),[14] ©2004 IEEE)

structured manner.

2.1.1.2 Fault Detection and Diagnosis

With the growing demand for dependability of modern technical systems, a wide variety of fault detection and diagnosis (FDD) methods have been developed in diverse disciplines such as chemical engineering, control engineering, automation engineering, and reliability engineering.[19]

Zhang and Jiang (2008)[19] reported an extensive bibliographical review on reconfigurable fault-tolerant control systems with 376 references starting from 1971. The review also

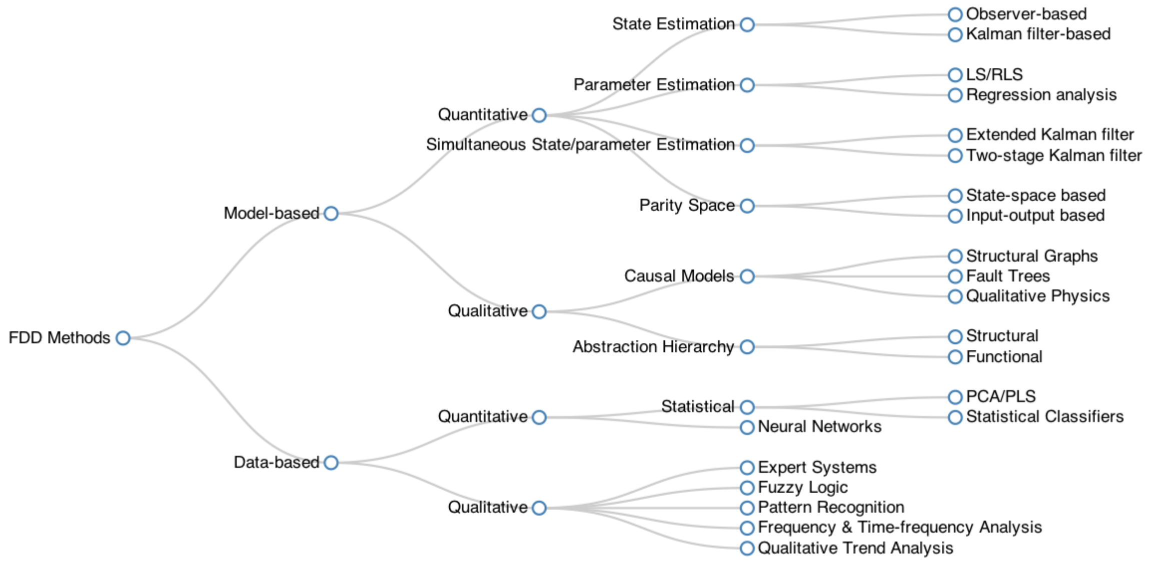


Figure 2.2: Classification of FDD methods (adapted from Zhang (2008)[19])

presented a classification of FDD methods, as in Fig. 2.2. Venkatasubramanian *et al.* (2004) reviewed the three different categories of FDD in a series of three articles: quantitative model-based methods,[20] qualitative models and search strategies,[21] and process history-based methods.[22] Isermann (1997,[23] 2005[24]) presented an introduction to the field of FDD and described the general scheme of different supervision methods. This scheme is an abstraction of various FDD and supervision methods, and identifies a set of essential elements to construct supervisory systems with FDD [iv]. His book on the fault-diagnosis systems[25] comprehensively covers fault-related topics from fault definition to fault detection, fault tolerance, and related applications.

[iv]The concept and structure of this abstract scheme inspired the design of the safety-oriented software architecture described in Chap. 5.

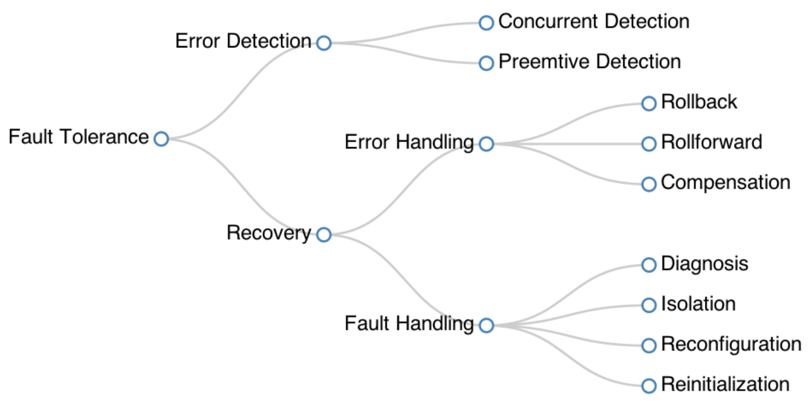


Figure 2.3: Fault tolerance techniques (adapted from Avizienis (2004)[14])

2.1.1.3 Fault Tolerance

The concept of fault tolerance was formulated by Avizienis in 1967:[26] *"We say that a system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults."*[27] The dependability formalism defines four categories to attain the attributes of dependability and security:[26] fault prevention, fault tolerance, fault removal, and fault forecasting. Fault tolerance aims to avoid service failures in the presence of faults by error detection and system recovery, and Fig. 2.3 shows the techniques for fault tolerance.

Fault tolerance has been studied in areas where the continuous operation of the system is important, such as high-confidence computing domains and mission-critical application domains. One of the early works on fault tolerance is the design and construction of an experimental computer – called the Self-Testing And Repairing computer – of which fault tolerance design includes dynamic (standby) redundancy, replaceable subsystems, and a program rollback provision to eliminate transient errors.[28] Randell (1975)[29] discussed a method for structuring complex computing systems using recovery blocks, conversations, and fault-tolerant interfaces to facilitate the provision of dependable error detection and recovery facilities. Avizienis (1997)[26] presented a guideline for designing fault-tolerant systems with the bottom-up approach that included specification, implementation, evaluation, and modification. Zhang and Jiang (2008)[19] also extensively reviewed fault tolerant control techniques and methods in the area of automatic fault tolerant control (AFTC).

In the medical robotics domain, the dependability formalism and semantics have not yet been widely adopted, although some prior works introduced the concept of dependability (e.g., Dowler 1995,[30] Duchemin 2004,[31] Sanchez 2014[32]). To benefit from the comprehensive and precise definition of concepts and terminologies of the dependability formalism, we follow the formalism throughout this thesis.

In the medical robotics domain, fault tolerance has not yet been recognized as an important system property. One reason is that it is often sufficient for medical robot systems to be fail-safe because the robot can be generally brought to a safe state by powering off the motors, and the medical intervention can continue via the conventional or manual method if the system fails to operate correctly.[33] Although switching to the conventional method is feasible in general, it is desirable if the system can tolerate faults or errors of the system and continue the tasks. It would be even more ideal if faults or errors can be prevented in advance. If no conventional method is available or the human cannot continue the task without the robot, fault tolerance may be required as an essential property of medical robot systems.

2.1.2 Software Engineering for Safety

In software engineering, the pioneering work on safety is found as early as 1986. Leveson (1986)[34] attempted to survey software safety with the description of outstanding issues and research topics, and laid the foundation for safety as a separate research topic with many challenging problems.[35]

Leveson (1995)[15] is the standard reference for system safety.[36] Recognizing safety as an *emergent* property[13], system safety emphasizes the importance of a *systems* approach in dealing with safety issues and this approach is called *Safeware*[15]. Safeware presents the safe design techniques in order of the precedence, as in Fig. 2.4, and these techniques are grouped into four: *hazard elimination*, *hazard reduction*, *hazard control*, and *damage reduction*. Safeware also describes various traditional models and techniques for hazard analysis such as checklists, hazard indices, fault tree analysis (FTA) , management oversight and risk tree analysis, event tree analysis (ETA), cause-consequence analysis, hazards and operability analysis (HAZOP), interface analysis, failure modes and effects analysis (FMEA), failure modes, effects, and criticality analysis (FMECA), fault hazard analysis (FHA), state machine hazard analysis (SMHA), task and human error analysis techniques, and evaluation of hazard analysis techniques.[15]

Kelly (1998)[37] proposed an approach to the development, presentation, maintenance and reuse of the safety arguments within safety cases using a graphical notation, called the *Goal Structuring Notation (GSN)*. GSN provides a structured approach to managing and presenting

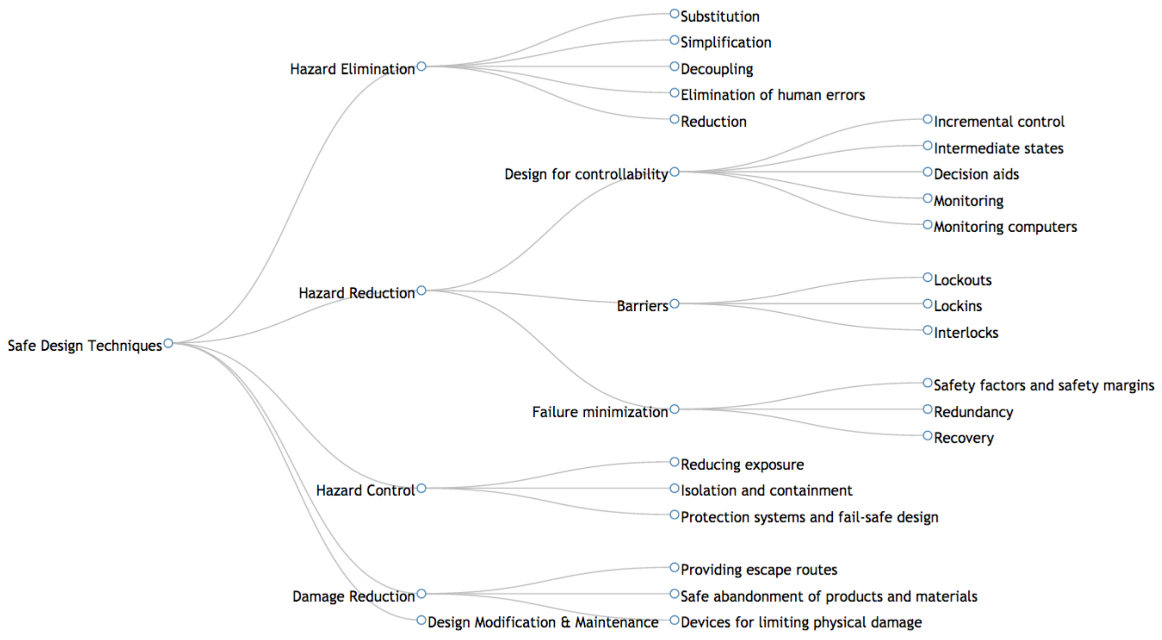


Figure 2.4: Safe design techniques in order of their precedence (adapted from Leveson (1995)[15])

safety cases. Safety-critical industries, such as aerospace and automobile industries, have adopted this approach to improve the structure, rigor, and clarity of safety arguments.[38] One interesting concept is the *Safety Case Patterns*[39] that enables the systematic reuse of common structures in safety case arguments across different systems.

In the International Conference on Software Engineering (ICSE) 2000 edition of The Future of Software Engineering (FoSE), Lutz (2000)[36] summarized six key areas in software engineering for safety as follows:

1. Hazard analysis
2. Safety requirements specification and analysis
3. Designing for safety
4. Testing
5. Certification and standards

6. Resources

Based on this review of the state-of-the-art in these six key areas, she provided a road map for software engineering for safety with six challenges that have to be addressed: (1) further integration of informal and formal methods, (2) constraints on safe product families and safe reuse, (3) testing and evaluation of safety-critical systems, (4) run-time monitoring, (5) education, and (6) collaboration with related fields.

Leveson (2004,[40] 2011[41]) presented a new approach to safety, called *STAMP* (Systems-Theoretic Accident Model and Processes), based on modern systems theory. The rationale behind this new approach is that the traditional models and techniques in safety and reliability engineering (e.g., FTA and FME(C)A) have changed very little, whereas technological advancements are making fundamental changes to the way modern systems operate and interact with the environment.[41]

Continuing a roadmap for the software engineering for safety by Lutz in 2000,[36] Heimdahl (2007)[35] presented a follow-up discussion on the challenges to the future of software engineering for safety in FoSE 2007. The four issues addressed are: (1) education and training of software engineering professionals, (2) software certification, (3) model-based development, and (4) data intensive systems.

In the latest edition of the FoSE (2014), Hatcliff *et al.* (2014)[42] presented a comprehensive and critical review of the state-of-the-art and challenges of modern software engineering for safety, including current practices, the desired goals, and gaps and barriers to reaching the desired goals. Based on this review, they provided possible research directions in software engineering for safety and safety certification as follows: (1) development of the measurable “confidence”, (2) further understanding of evidence, (3) improvement of requirements specification to drive the development and verification, (4) effective ways of performing compositional certification, (5) specification for functional and performance timing requirements, (6) development of tools to facilitate qualification and verification, and (7) more scalable, more practical formal methods.

2.1.3 Component-based Software Engineering

As robotics technology advanced and enabled tasks that had not been possible earlier, the scale and complexity of robot systems have significantly increased. Traditional programming models such as object-oriented programming were not adequate for such large and complex systems. Within the robotics community, there was an early recognition of these system issues.

In the early 1990s, Component-based Software Engineering (CBSE)[43] emerged as an effective programming model that can deal with various aspects of large and complex systems. The fundamental philosophy of CBSE is *software reuse*, which is the use of existing software to construct new software while promoting the development of maintainable, reliable, and affordable software systems.[44] In the software engineering community, CBSE has been proved as an effective approach to many system development issues.

Noting the benefits of CBSE, the robotics community started to adopt CBSE since early to mid 2000,[45] and the community’s experience with CBSE showed that CBSE is highly effective in achieving advanced tasks with complex requirements. Proved by its successful and wide adoption by various component-based software frameworks, CBSE has become the de facto standard programming model in robotics.[46–49] A short list of robotics software frameworks that provide component-based development environments includes Orocos,[50] Orca,[51] ROS,[52] cisst,[49] OPRoS,[53] and OpenRTM.[54] The robotics literature provides comparative studies and reviews of different robot software frameworks (Shakhimardanov *et al.*, 2007,[55] 2010a,[56] and 2010b;[57] Elkady and Sobh, 2012[47]). However, the semantics or formalism for error propagation that can explicitly define, capture, and handle errors has not yet been introduced to the robotics community.

In CBSE, there has been a stream of research in the area of safety analysis and evaluation of component-based software systems (e.g., how to model errors, how to represent failure propagation between components).

Fenelon (1994)[58] introduced modular concepts to specify the failure behavior of components, called the Failure Propagation Transformation Notation (FPTN). The basic entity is a FPTN-Module which contains specifications for failure propagation, failure transformation, and detection and generation of internal failures. Its failure semantics defines timing failures (e.g., too early or too late), invalid values, commission, and omission.

Kaiser (2003)[59] proposed the Component Fault Trees (CFT), an extension of the traditional Fault Trees (FT),[60] to allow the encapsulation and distribution of partial FTs within the boundary of components, thereby enabling compositional hazard analysis for component-based systems. Each instance of CFT is independent from each other, and a CFT is connected to another CFT via input and output ports.

Grunske (2005)[61] investigated how to adapt traditional safety analysis and assessment techniques to component-based systems and evaluated different safety analysis techniques such as failure propagation and transformation notation, CFT, and parametric contracts. In another work in 2005,[62] he proposed the State-Event Fault Trees (SEFT) to facilitate a quantitative safety analysis of component-based systems. SEFT has a state-event semantics that can describe the stochastic behavior of a component at a low level using states and events. In 2006, he integrated component-based safety evaluation techniques and failure propagation model with the SaveCCM component model for the early estimation of failure and hazard probabilities.[63]

Domis (2008,[64] 2009[65]) proposed the *Safe Component Model (SCM)* that exploits one of the basic principles of CBSE: separation of concerns. Focusing on the hierarchical structure of components, SCM separates specification from realization, and functional properties from nonfunctional properties. With this approach, SCM enables the tight integration of safety analysis into the component-oriented and model-based development process.

2.2 Robotics

In modern personal robotics, tasks that robots perform often involve humans. Compared to the early generation of robots (e.g., industrial robots) where their workspace was physically isolated from humans, modern personal robots operate closely with humans and their workspace often has significant overlap with that of humans. Naturally, this leads to more attention to the issue of safety, which was publicly recognized within the robotics community:

> *"We need to build systems that are safe even in the event of a failure of sensors, actuators, wiring, software, and computers. We cannot afford to take unnecessary risks in laboratories and demonstrations. Just because it has not happened does not mean that it cannot happen–one serious injury or worse would set our field back significantly."*
>
> – Peter Corke, *IEEE Robotics & Automation Mazagine (2011)*[66]

In robotics, there have been a few different approaches to safety. The physical human robot interaction (pHRI)[67] is one area that explores the safety aspects of robot manipulators to minimize injury or impact on the human due to physical contacts or collisions with the robot. Topics of interests in that area include collision detection and reaction, impact characterization, injury analysis, and control strategy or architecture. Heinzmann and Zelinsky (2003)[68] proposed a control strategy for robot manipulators that provides quantitative safety guarantees and limits the potential impact force in case of collisions. Bicchi and Tonietti (2004)[69] reported a quantitative analysis of the inherent trade-off between safety and performance based on the severity of injuries due to collisions. Haddadin systematically investigated the area of collision detection and impact/injury minimization due to collisions through a series of works (2008,[70] 2009,[71] 2011[72]), and recently published a book on these topics.[73] Recently, he introduced robotics research on this area to the safety-critical systems domain.[74]

Another area of work in robotics in terms of safety is FDD and fault tolerance. In the early 1990s, Visinsky and Walker investigated fault detection and fault tolerance of robot systems. In 1993, they proposed a layered control framework that consists of the servo, interface, and supervisor layers to enable hierarchical fault detection and fault tolerance for different robots.[75] They also conducted a survey (1994)[76] on fault tolerant research in the robotics domain, exploring possible research topics for robotic fault tolerance such as fault detection and error recovery. Another approach to fault detection includes Verma's work on the run-time fault detection and diagnosis using particle filters in 2004.[77] They presented a set of algorithms that improved the accuracy of fault detection and identification under noisy environments using variable resolution particle filters and unscented Kalman filters. Carlson (2004)[78] reported a summary of discussions and activities of the ICRA 2004 workshop on fault detection, identification, and recovery (FDIR) for robots. The workshop identified safety as one of the five key areas that can further enhance the FDIR performance, along with adaptation, knowing when to ask for help, overcoming wireless communications problems, and unsupervised learning.

One interesting approach is the integration of international standards with the development process for robot systems. Doukas (2006)[79] applied the IEC 61499 function block to a PID control loop for a robot arm, and evaluated the applicability and effectiveness of the function block for robot application systems. Hanai (2012)[80] presented a system development process that follows the guideline of the IEC 61508 (e.g., appropriate safety integrity level) with the use of SysML and the IEC 61499 function blocks. This standard-based approach is also found in the medical robotics domain (refer to Sec. 2.3.2 for more details).

Woodman (2012)[81] proposed a safety-oriented system development methodology for personal robot systems, called the *safety-driven control system architecture*. Built on top of the traditional hazard analysis techniques, the safety protection system is configured based on a safety policy to prevent the controller from activating actuators that may lead to unsafe events. Recently, Tadele (2014)[82] presented a survey on safety in robotics in three respects: collision-focused safety criteria and metrics, mechanical design and actuation, and controller design.

Our observation is that there is a growing body of work on safety in robotics as safety is getting more attention within the robotics community. The research theme in robotics so far mostly centers around the manipulator safety (e.g., collision detection and avoidance, impact analysis and minimization). Although the domain-specific characteristics of medical robot systems are different from those of (non-medical) personal robot systems, the medical robotics domain can benefit from the existing body of work in robotics because manipulator safety is one of the safety features that has been frequently used in the medical robotics domain (refer to Sec. 2.3.1).

2.3 Medical Robotics

An industrial robot was first used for a stereotactic neurosurgery application as a surgical tool holder in 1985, and a wide variety of medical robot systems have been developed since then. In the late 1980s and early 1990s, industrial robots were primarily used for surgical applications because of their accurate and precise movements. From the early 1990s, more application- and domain-specific requirements were incorporated into the design of medical robot systems, and this led to the development of custom robot systems for particular applications. Such custom robots have successfully enabled a wide range of surgical tasks and solved challenging *functional* problems that conventional or manual tasks had suffered from. For example, motion scaling combined with force sensing and tremor reduction allows surgeons to perform microsurgery tasks at much smaller scale with higher precision.[83] In contrast, the *non-functional* properties of medical robot systems – particularly safety – have not received much attention in the research community, although there have been discussions and prior works on system- or application-specific safety.

Medical robot systems have domain-specific characteristics that are distinct from other safety-critical systems or other non-medical robot systems, and thus their safety should be considered with these domain-specific characteristics. In medical robotics, there have been domain-specific design techniques and approaches to safety, and Sec. 2.3.1 presents a survey of these efforts. Although many review or survey articles on medical robotics have been published [v], there is no survey with particular emphasis on the safety aspect of medical and surgical robot systems to the best of our knowledge. Sec. 2.3.2 shows a list of prior works that tried to incorporate medical device standards into the system development process, or discussed the standards. In addition, Sec. 2.3.3 summarizes recent projects or activities that specifically focus on the safety aspects of medical and surgical robot systems.

2.3.1 Survey on Safety

This survey reviews prior works on the safety of medical and surgical robot systems. The scope of literature that we reviewed is limited to *academic* articles and books. Although various commercial medical robot systems exist in the market, the detailed design techniques and methods for safety are not publicly accessible, and their experience on safety is rarely shared with the academic community. In addition, we only selected (1) articles that describe or discuss safety, safety designs, or safety features of the system, (2) articles of which the main concept centers around safety, and (3) review or survey articles. This survey does not attempt to collect an extensive list of medical robot systems that have been developed so far.

2.3.1.1 Domain-specific Safety Features

We reviewed about 100 articles and book chapters and identified 8 classes of domain-specific safety features that have been frequently used in the domain. Table 2.1 shows those 8 classes

[v] A sampling of review/survey or tutorials/book chapters includes Howe (1999),[84] Davies (2000),[85] Cleary (2001),[86] Taylor (2003),[87] Pott (2005),[88] Taylor (2006),[3] Kazanzides (2009),[33] Troccaz (2009),[89] Wolf (2009),[90] Dogangil (2010),[91] OToole (2010),[92] Moustris (2011),[93] Beasley (2012),[2] and Rosen (2012).[94]

of safety features. The *Class* column defines the *class ID* of safety features, which the survey result table in the next section (Table 2.2) uses to describe classes of safety features discussed or described in the article. Each class is categorized by: (1) the *Type* field where **D** represents the design techniques or methods that are considered at *design-time* and **R** indicates the *run-time* safety features, and (2) the *HW, SW,* and *HCI* fields that represent the characteristics of safety features (HW: hardware-based, SW: software-based, HCI: Human-computer interface). The *Examples* column shows examples of particular safety design features.

Each class of safety features is defined as follows:

1. Mechanical Constraints: This class of safety features imposes mechanical or physical constraints on the design of robot manipulators or the way that the robot manipulator moves. One representative example of this class is to *limit the maximum velocity of the end effector* by using high gear reduction ratios or low power actuators. This type of safety feature is found in prior works such as Taylor (1995),[95] Stoianovici (1998),[96] and Zhu (2000).[97]

2. Hardware/Software Interlocks: Interlocks in general are commonly used to enforce correct sequencing or to isolate two events in time.[15] For example, an interlock ensures that an event does not occur inadvertently or while a particular condition exists, or that an event occurs before another event. Two examples of this class of safety features are the *safety circuits to power off the robot* in case of emergency and the *dead-man switch*[7]. Examples of prior works that adopted this class of safety features include Degoulange

(1998),[98] Duchemin (2001),[99] Dombre (2003),[100] and Bast (2006).[101]

3. Redundant Sensing and/or Computation: Redundancy involves deliberate duplication to improve reliability.[15] Multiple devices or hardware components that perform identical functions can be deployed to the system. Software can contain two types of redundancy: *data* (e.g., CRC, checksums, parity bits, sequence numbers) and *control* (e.g., algorithmic redundancy; multiple versions of the same algorithm). Examples of this class of safety features include *separate monitoring subsystems*, *redundant sensors* such as dual position/velocity encoders, and *consistency checks* (e.g., consistency of measured distances between fiducials). Typically, safety features of this class perform continuous checking of measured quantities at run-time in the background while the system operates, and generate error events if any inconsistency is detected. A short list of prior works with the redundant design includes Taylor (1991),[4] Kazanzides (1992a),[5] Guthart (2000),[102] and Hagn (2008).[103]

4. Human-Computer Interface (HCI): One of the principal requirements that many medical robot systems have widely adopted is that the surgeon must be the "decision maker" at all times. By relying on the surgeon's experience and knowledge, the robot system can operate more safely. To support this feature, the system should allow the surgeon to interact with the system to control the robot, and provide information about the current status of the system and the environment, so that the surgeon has comprehensive information and awareness of the situation and makes decisions to stop the robot or to abort the procedure. During this interaction, any potential chance of human errors has to be minimized.

To interact with the system, the system can be designed to enable *intraoperative control by the surgeon* ("hands on" control), or can provide *devices for the surgeon* such as keyboards, joysticks, touch screens, or pedals. For the surgeon's situational awareness, the system can *visualize the current progress* or *pre-planned surgical plans*, or the *system health status*. The system can also provide the surgeon with *sensory feedback* such as visual, auditory, and tactile feedback. To avoid or eliminate human errors, the user interface design techniques such as the *confirmation of selected actions* or the *simple user interface with clear messages* can be used. This class of safety features has been widely used, as seen in Fig. 2.5. Examples of articles with this safety feature are Kobayashi (1999),[104] Guthart (2000),[102] and Guiochet (2002).[105]

5. Environment Sensing: From the early days in medical robotics, a force sensor has been frequently used to provide safety, an improved and intuitive human machine interface, or tactile feedback. A short list of prior works that used a force sensor include Taylor (1989),[106] Kazanzides (1992b),[107] Lueth (1998),[108] and Rodriguez (2005).[109]

6. Software Constraints: The idea of software constraints is conceptually similar to that of the mechanical constraints. Essentially, the robot control software imposes *virtual constraints* on the robot in terms of various quantities such as position, velocity, acceleration, force, and torque, in order to limit the workspace or movement of the robot. The same idea can also be applied to the *singularity avoidance*. This class of safety features was used by Taylor (1991),[4] Troccaz (1993),[110] Harris (1997),[111] and Sanchez (2014).[32]

7. Diagnostic Tests: Safety features of this class are usually performed as part of the application or the workflow during the procedure. For example, *safety checks at power-up* can be performed after power shutdown due to errors in order not to power up the robot system unless particular safety conditions are satisfied. Registration integrity tests during the procedure, such as *verification point checks* using external trackers and *fiducial registration error (FRE) checks*, are other examples. Prior works that used this class include Taylor (1990),[11] Davies (1996),[7] Duchemin (2001),[99] and Laible (2004).[112]

8. Software Engineering Techniques As the robot systems become larger and more complex, different types of software engineering techniques have been adopted within the medical robotics domain. Some works tried to *integrate the international standards* with the system design process (e.g., Varley (1999),[113] Rovetta (2000),[114] Korb (2005),[115] Guiochet (2012);[116] refer to Sec. 2.3.2 for further details). Other works adopted the *state-based design* to benefit from its explicit and declarative characteristics (e.g., Guiochet (2002),[105] Laible (2004),[112] Gary (2006),[117] Lum (2009)[118]). Recent work has also begun to apply *formal methods* to the software of medical robot systems in order to verify the correctness or safety property of the software (e.g., Muradore (2011),[119] Kazanzides (2012),[120] Kouskoulas (2013).[121]

Table 2.1: Category of domain-specific safety features

Class	Safety Feature	Type	HW	SW	HCI	Examples
1	Mechanical constraints	D/R	✓			Low speed or small motor torque (slow motion) Robot at distance (workspace limitation) Lock-down during intervention (brakes, non-backdrivable) Manipulator safety
2	Hardware/software interlocks	R	✓	✓		Safety circuits for power cut-off Power enable interlocks (software and/or hardware) Dead-man switch Safety timeout (watchdog)
3	Redundant sensing and/or computation	R	✓	✓		Separate monitoring subsystem Redundant sensors (position or velocity encoders) Consistency checks (redundant registration checks)
4	Human-computer interface (HCI)	R	✓	✓	✓	Intraoperative control by surgeon (“hands on” control) Intraoperative monitoring by surgeon (“decision maker”) Visualization (display of plans, progress, or current status) Devices for surgeon to interact with system Confirmation of selected actions Sensory feedback (visual, auditory, tactile) Simple HCI; clear messages
5	Environment sensing	R	✓			Force/torque sensor Vision and/or distance sensing
6	Software constraints	R		✓		Dynamic constraints (safety volume, virtual fixture) Singularity avoidance Online monitoring

Table continued

Table 2.1 – continued from previous page

Class	Safety Feature	Type	HW	SW	HCI	Examples
7	Diagnostic tests	R		✓		Initial tests (safety check at power-up) Fiducial registration error (FRE) tolerance
8	Software engineering techniques	D		✓		Integration of safety standards State-based design (state variable, state machine) Use of formal methods Testing (unit-tests, workflow tests)

2.3.1.2 Survey Results

Table 2.2 compiles a complete list of the literature reviewed for this survey. Although there exist early work in medical robotics (e.g., Garbini *et al.* (1987),[122] Watanabe *et al.* (1987),[123] Kosugi (1988)[124]), they do not meet our literature selection criteria, as described in Sec. 2.3.1, and thus are not included in the table.

The *Reference* column indicates the first author and publication year of each article. Multiple names or publication years may appear if multiple articles have been published for the same system. The *Application* shows the surgical procedure or clinical area for which the system is designed. This field is in italics if it represents the main topic of the article (e.g., formal methods applied, verification of the safety of virtual fixtures). The *Status* represents the development status and is defined as follows:

- **D**: Early in the development process (e.g., prototype, phantom study, in vitro study)
- **C**: Performed cadaver study or in vivo experiments (e.g., animal study)
- **H**: Completed clinical (human) trial
- **P**: Released as a commercial product

The *Robot* is the name of the system or the manipulator used, or the names of software packages or system architectures. The *DOF* shows the degrees-of-freedom of the robot. In the case of telesurgery systems that consist of the master and the slave subsystems, this field represents "master DOF/slave DOF". The *Type* represents the class of the system, according to the classification of medical robot systems that has been used in the domain (e.g., Troccaz,[125] Davies,[85] Wolf[90]):

- **P**: Passive (unpowered) robots or motorized tool holders
- **S**: Semi-active or semi-autonomous robots
- **A**: Active or autonomous robots
- **T**: Remote manipulators for telesurgery (master/slave)

The *Safety Features* lists the class IDs, as defined in Table 2.1, for the safety features described or discussed in the article.

Table 2.2: List of articles or books reviewed

Reference	Application	Status	Robot	DOF	Type	Safety Features
Shao 1985[1] Kwoh 1988[126]	stereotactic neurosurgery	H	PUMA 200	6	P	1, 4
Davies 1989[127]	prostatectomy (TURP)	D	PUMA 200	6	A	3,4
Lavallee 1989,[128] 1992[129]	stereotactic brain surgery	H	n/a	6	A	1,4,6
Taylor 1989,[106] 1990,[11] 1991,[4] 1994[130]	orthopaedics	D	IBM 7576 SCARA	5	A	2,3,4,5,6,7
Davies 1991[131, 132]	prostatectomy (TURP)	D	PUMA 200	6	A	1,3,4,6
Drake 1991[133]	stereotactic brain surgery	H	PUMA 200	6	P/A	1,2,3,4,6
Taylor 1991[134]	craniofacial osteotomy	D	Modified SCARA	3+6	A	1,3,4,5,6,7
Kienzle 1992[135]	orthopaedics	C	PUMA 560	6	A	1,4,5
Kazanzides 1992,[5, 107] 1993,[136] 1995,[137] 1999,[138] Cain 1993,[139] Mittelstadt 1993[140]	orthopaedics	P	ROBODOC®	5	A	1,2,3,4,5,6,7,8
Villotte 1992,[141] Glauser 1993,[142, 143] 1995[144]	stereotactic neurosurgery	D (1992) C (1995)	Minerva	6	P	1,6
Fadda 1993[145]	orthopaedics	D	PUMA 560	6	A	3,5,6

Table continued

Table 2.2 – continued from previous page

Reference	Application	Status	Robot	DOF	Type	Safety Features
Matsen 1993[146]	orthopaedics	C	PUMA 260	6	A	1,4,6
Ng 1993[147]	prostatectomy (TURP)	H	SARP	4	A	1,4,6
Troccaz 1993,[110] 1996,[125] Schneider 2001[148]	*mechanism design*	D	PADyC	1,2,3,6	S	6
Kavoussi 1994[149]	laparoscopic surgery	H	AESOP	n/a / 6	T	4,5
Sackier 1994[150]	laparoscopic surgery	D	AESOP	6	A	1,4,6
Davies 1995[151]	orthopaedics	D	Two-link manipulator	6	S	2,5,6
Masamune 1995[152]	stereotactic neurosurgery (MRI compatible)	D	*custom*	6	A	1
Taylor 1995[95]	laparoscopic surgery	C	LARS	6	A	1,2,3,4,5,6
Ng 1996[153]	prostatectomy with ultrasound guidance	D	SARUD (SARP+US)	n/a	A	1,2,3,4
Funda 1996[154]	laparoscopic surgery	C	PLRCM	8	A	6
Brandt 1997,[155] 1999[156]	orthopaedics	D	CRIGOS	6	S/A	1,4,6
Davies 1997,[157] Harris 1997[158]	orthopaedics	D	Acrobot	4	S	2,4,5,6
Harris 1997[111]	prostatectomy (TURP)	H	Probot	4	A	1,3,4,6

Table continued

Table 2.2 – continued from previous page

Reference	Application	Status	Robot	DOF	Type	Safety Features
Cadeddu 1998[159]	endourology	H	PAKY	7+1	P	1,4
Degoulange 1998[98]	ultrasound device holder	D	HIPPOCRATE	6	A	1,2,3,4,5,6,7
Lueth 1998[108]	maxillofacial surgery	C	OTTO (PUMA 500 + MSS SurgiScope)	6+(6/7+1)	A	1,4,5,6
Stoianovici 1998[96]	image-guided needle access	H	PAKY+MINI-RCM	7+3	P/A	1
Cavusoglu 1999[160]	laparoscopic telesurgery	D	Telesurgical Workstation	6 / 6	T	2,4
Kobayashi 1999[104]	laparoscopic neurosurgery	D	laparoscopic manipulator	2	P	1,2,4
Pierrot 1999[161]	ultrasound device holder	C	Hippocrate	6	A	1,2,3,4,5,6,7,8
Reichenspurner 1999,[162] Ghodoussi 2002[163]	telesurgery	H	ZEUS	2*5 / 2*7+6	T	4
Taylor 1999[164]	microsurgery	D	Steady Hand	7	S	1,2,3,4,5
Tombropoulos 1999[165]	stereotactic radiosurgery	D	CARABEAMER (planner)	6	A	1,6
Varley 1999[113]	endoscopic surgery	P	EndoAssist	n/a	n/a	1,2,4,7,8
Guthart 2000[102]	MIS	P	da Vinci™	2*7 / 2*6+4	T	2,3,4,5,6,7,8
Rovetta 2000[114]	telesurgery	H	*custom* (industrial robot-based)	n/a / 6	T	1,2,3,4,6,8

Table continued

Table 2.2 – continued from previous page

Reference	Application	Status	Robot	DOF	Type	Safety Features
Zhu 2000[97]	teleoperated ultrasound diagnosis robot	D	*custom*	6 / 6	T	1,4,5,6
Duchemin 2001[99]	reconstructive surgery	D	SCALPP	6	S/A	1,2,3,4,5,6,7
Engel 2001[166]	craniofacial surgery	D	Staubli RX90 (modified)	6	A	2,3,4,5,6,8
Fei 2001[9]	urology	D	URObot	7+4	A	1,3,4,8
Gonzales 2001,[167] Guiochet 2002[105]	tele-echography	D	TER	6 / 2	T	2,4,5,8
Jakopec 2001,[168] 2003[169]	orthopaedics	H	Acrobot	3	S	1,4,5,6
Masamune 2001[170]	image-guided needle access	D	*custom*	7+3	A	1
Dombre 2003[100]	reconstructive surgery	C	Dermarob	6	A	1,2,3,4,5,6,7
Korb 2003[171]	craniotomy	D	RoboCKA (Staubli RX90)	6	A	1,2,3,4,5,6,8
Cleary 2004,[172] Gary 2006,[117] 2011[173]	image-guided surgery	D	IGSTK (software)	-	-	8
Laible 2004[112]	*multi-purpose tool holder*	D	n/a	6	P/A	1,2,3,4,5,6,7,8
Korb 2005[115]	biopsy	D	B-Rob II	4	P	1,3,4,6,8
Plaskos 2005[174]	orthopaedics	D	Praxiteles	2	A	1,2

Table continued

Table 2.2 – continued from previous page

Reference	Application	Status	Robot	DOF	Type	Safety Features
Rodriguez 2005[109]	orthopaedics	H	Acrobot	3+3	S	1,2,4,5,6
Bast 2006[101]	craniotomy	D	CRANIO	6	A	1,2,3,4,5,6,8
Fodero 2006,[175] Lum 2006,[176] 2009[118]	MIS	C	RAVEN	n/a / 2*7	T	1,2,3,4,6,8
Hagn 2008[103]	endoscopic surgery, neurosurgery, maxillofacial surgery	D	DLR MIRO	7	T/S/A	1,3,4,5,6
Muradore 2011[119]	*puncturing task*	-	-	-	-	8
Guiochet 2012[116]	*rehabilitation robotics*	D	MIRAS	-	-	8
Kazanzides 2012[120]	*correctness of concurrent data exchange mechanism*	D	*cisst* (software)	-	-	8
Kouskoulas 2013[121]	*safety of virtual fixtures*	D	Neuromate	-	-	6,8
Sanchez 2013,[177] 2014[32]	single-port laparoscopic surgery	D	ARAKNES	6(7) / 2*(6+1)	T	1,2,4,6

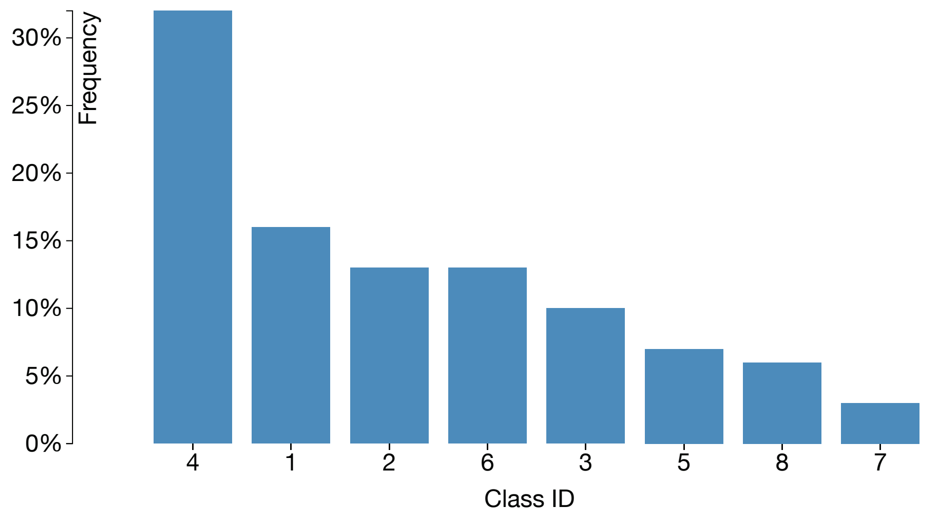


Figure 2.5: Relative frequency of each class of safety features used. The x axis represents the class ID of safety features, and the y axis shows a relative ratio of occurrences of each class.

Fig. 2.5 shows how many times each class of safety features are mentioned among the articles reviewed. The x axis shows the class ID of the safety feature and the y axis represents a relative frequency, i.e., a ratio of the number of cases where each class of safety features was used, to the total number of cases. According to this figure, the most widely reported safety feature is class 4 (HCI) and the least frequently reported is class 7 (self-diagnostic tests). However, this does not imply that class 4 is more effective or safer than class 7 because the figure shows only the *relative* distribution derived from the set of articles reviewed. Rather, this figure represents the *relative availability* of prior works that a system designer can refer to when developing safety features for a new medical robot system.

2.3.2 Medical Device Standards

Currently, there is no safety standard that specifically governs the design of medical robot systems;[33] rather, developers conform to existing standards, such as IEC-60601 and IEC-62304, that apply to medical devices. Similarly, medical robots are subject to the same regulatory approval processes as other medical devices. As in other safety-critical domains, developers must invest significant engineering effort to ensure that every device they design is safe and meets regulatory requirements. This obviously leads to an effort to follow the standards as design guidelines, or to integrate the standards with the system design and development process, in such a way that the conformance to the standards helps to reduce engineering effort for the regulatory approval processes.

Table 2.3 shows a summary of prior works that applied relevant medical device standards to the development process of medical robot systems, or presented discussions about the standards within the medical robotics domain.

Varley (1999)[113] described a set of development methodologies around IEC 1508, which were used in practice to develop an endoscopic camera manipulator (the EndoAssist by Armstrong Healthcare) that has been approved for commercial use both in the U.K. (by the Medical Devices Agency) and in the U.S. (by FDA).

Rovetta (2000)[114] developed a telerobotic surgery system that performed a prostate biopsy on a human patient. The system had a set of safety features, such as emergency stop, enabling device, safety stop, reduced speed, and interlock, and they presented how each

Table 2.3: Medical device standards used in medical robotics

Name	Description	References
IEC 60601	Medical electrical equipment	Rovetta,[114] Fei[9]
IEC 60812	FMEA and FMECA	Kazanzides[33]
IEC 61025	FTA	Laible,[112] Korb[115]
IEC 61508	Functional safety of electrical/electronic/programmable electronic safety-related systems	Varley,[113] Guiochet[116]
IEC 62304	Medical device software – Software life cycle processes	Rovetta[114]
ISO 10218	Robots and robotic devices – Safety requirements for industrial robots	Rovetta[114]
ISO 14971	Medical devices – Application of risk management to medical devices	Kazanzides,[33] Guiochet[116]
510(k)	Premarket notification (required for U.S. FDA approval)	Fei[9]
MDD	Medical Device Directive (93/42/EEC) Required for EU CE Mark	Laible,[112] Sanchez,[177] Guiochet[116]

safety feature complies to relevant safety standards.

Fei (2001)[9] proposed a systematic method (hazard identification and safety insurance control) to analyze, control, and evaluate system safety from software, hardware, and policy perspectives. Their software development cycle consist of multiple phases from the system requirements to the system integration, where requirements, design, and tests form a closed loop to ensure safety of each phase.

Laible (2004)[112] presented an architecture of a fail-safe control for robotic surgery, which includes a set of safety functions for error detection and error reaction. The safety requirements of the system are derived from the Medical Device Directive (MDD) and FTA.

Korb (2005)[115] showed the development process of a biopsy robot for clinical studies, and described how they applied the risk analysis and safety assessment methods (FTA) to the robot system. They also presented the seven surgical robot risks (7 SRRs) that represent the main critical topics of surgical robots in general. The 7 SRRs include (1) image processing and planning, (2) registration and tracking, (3) movement, (4) reliability of control software, (5) vigilance, (6) hygienic considerations, and (7) clinical workflow.

Starting with a discussion about the characteristics of medical robot systems, Kazanzides (2009)[33] presented a tutorial overview of safety design for medical robots and discussed three safety considerations: safety requirements, risk assessment, and safety design.

Guiochet (2012)[116] applied a set of safety standards to the design process of an assistive robot system for standing, walking, and sitting, and presented a safety case argumentation based on the Goal Structuring Notation (GSN).[38]

Sanchez (2013)[177] presented the European standards *93/42/CEE* and *2007/47/CE* that classify medical devices into 4 classes depending on the level of risk (Class I: low level of risk, Class IIa: average level of risk, Class IIb: high level of risk, and Class III: critical level of risk), and described how their design framework (the ARAKNES platform) complies with those European directives as a Class IIb system.

2.3.3 Activities

In Europe, there have been large scale projects that focus on safety and/or place significant emphasis on safety. Table 2.4 presents a list of such projects in three different areas: medical robotics, (non-medical) robotics, and the safety-critical domain.

The ARAKNES (Array of Robots Augmenting the KiNematics of Endoluminal Surgery)

Table 2.4: Safety-related European projects

Area	Project Name	Duration	Description	URL
Medical Robotics	ARAKNES	2008-2012	Array of Robots Augmenting the KiNematics of Endoluminal Surgery	`araknes.org`
Medical Robotics	SAFROS	2010-2013	Patient safety in robotics surgery	`safros.eu`
Robotics	PHRiENDS	2006-2009	Physical Human-Robot Interaction: DepENDability and Safety	`phriends.eu`
Robotics	RoboSAFE	2013-2016	Trustworthy robotic assistant	`robosafe.org`
Safety-critical	OPENCOSS	2011-2015	Open Platform for EvolutioNary Certification of Safety-critical Systems	`opencoss-project.eu`

is an European project that aims to develop a robot system for endoluminal surgery that can transfer laparoscopy techniques into single-port laparoscopy. As part of the project, the ARAKNES platform with a set of safety features for both hardware and software was developed and presented (Sanchez 2013[177] and 2014[32]).

The SAFROS (Patient Safety in Robotics Surgery) is another safety-oriented project in medical robotics. The goal of the project was to develop technologies for patient safety in robotic surgery, and its aims included the development of metrics for patient safety and methods that conform to safety requirements, and the application and verification of the developed methods to surgical scenarios. The results of the project were presented as the *SAFROS method* that addressed various components of surgical robot systems, such as medical imaging and segmentation, operating room supervision, and image registration.

Outside medical robotics, the PHRiENDS and RoboSAFE are two safety-oriented projects in robotics. The PHRiENDS (Physical Human-Robot Interaction DepENDability and Safety) aimed to develop technologies and key components for robots that can physically but safely interact with humans within the context of intrinsically safe physical human robot interaction (pHRI). The RoboSAFE project addresses safety issues with autonomous robot systems that interact with humans. The goal is to develop a holistic methodology for verification and validation that enables the design of safe and trustworthy robotic assistants using three approaches: formal verification, simulation-based testing, and formal user evaluation.

In the general safety-critical application domain, the OPENCOSS (Open Platform for EvolutioNary Certification of Safety-critical Systems) is one active project that targets the automotive, railway, and aerospace industries. It is an initiative to reduce development cost due to the safety (re)certification of safety-critical systems using model-based approaches and incremental techniques. The main idea is to reuse safety arguments and safety evidence of existing components to make the safety certification process more cost-effective and scalable.

2.4 Conclusions

This chapter presented a comprehensive review on safety and safety-related topics in different domains, including dependable computing, software engineering, and robotics. It is important to understand existing methods and techniques for safety from related domains because they have been proved and used as working solutions for real-life problems and systems in history. However, these methods and techniques may not be directly applicable to medical and surgical robotics due to fundamental differences in domain characteristics.

In medical robotics, we reviewed prior work that addressed or involved medical device standards. Additionally, we extensively reviewed the academic literature from 1985 to 2014, focusing on safety, and presented the eight domain-specific safety features as follows: (1) mechanical constraints, (2) hardware/software interlocks, (3) redundant sensing and/or computation, (4) human-computer interface, (5) environment sensing, (6) software constraints, (7) diagnostic tests, and (8) software engineering techniques.

One of our observations on safety research is that there exists an increasing attention to *non-functional properties* of complex safety-critical systems. This applies to not only traditional safety-critical domains, but also to the robotics domain, and particularly to the medical robotics domain. Another observation is that most of recent approaches to safety increasingly adopt *systematic* methods, rather than ad-hoc or application-specific methods. This trend is consistent with the traditional recognition that safety is not a component or subsystem property but an emergent property that requires systems approaches.[13,15] In addition, recent work in robotics has also begun to progressively adopt *software engineering techniques*, such as software architecture, formal methods, and system design with safety-related standards. As medical and surgical robot systems evolve, it is likely that these trends will be further accelerated and become more obvious as the system complexity will increase significantly. Thus, a set of new approaches and methods to deal with safety would be essential for more dependable medical robot systems. For the rest of this dissertation, we use these trends as motivating and underlying design principles.

Chapter 3

Safety Design View

3.1 Introduction

A variety of medical and surgical robot systems have been developed both in academia and industry. Commercial medical robot systems are being used in modern operating rooms and academic researchers build prototypes of medical robot systems for clinical experiments. These systems directly operate on a human, or even inside the human body, and thus the system must be designed with safety. As described in the literature review (Sec. 2.3.2), however, there is no safety standard for medical robot systems, and the system developers conform to existing medical device standards (e.g., IEC-60601, IEC-62304).

Building safe medical robot systems in accordance with the medical device standards typically requires a significant amount of engineering effort. Throughout the development process, system designers acquire knowledge and experience in safety engineering. If such expertise could be systematically collected in a structured manner, it would facilitate sharing of experience on the design and development of safe medical robot systems, and help us to reduce engineering efforts for building new systems. When sharing the design of safety features, it is important to understand what are the essential elements of safety features and what properties or aspects of safety features have to be described. This leads to the question of inherent characteristics of safety features, which can be addressed with the following questions:

- *"What constitutes safety features?"*: What are the essential functional elements of safety features? What should be considered when designing new safety features?
- *"How can safety features be more effectively described, presented, and shared in a systematic way?"*
- *"What is the design space (alternative design options) for safety features?"*

There exists a body of previous work on safety of medical robot systems, as reviewed in Sec. 2.3.2. But, the prior work mostly focused on system- or application-specific safety features, and did not place much emphasis on the basic concepts and essential elements of safety features. The conceptual foundation of safety would be essential to establish a common ground for safety research in medical robotics.

This chapter proposes a conceptual framework that identifies essential elements and enabling components of safety features of medical robot systems with consideration of run-time aspects of the systems. The starting point is to recognize safety as a system property.[15] We treat safety as an *emergent property* that has meaning only when considered at the system-level, not at the individual component level.[13] We also take into account issues related to the deployment of safety features, which could practically have a significant impact on their run-time characteristics or performance. These considerations lead us to define the *two views* of safety features (Sec. 3.2). Based on these two views, we define the design space of safety features, which can simultaneously present the mechanisms and run-time aspects of safety features (Sec. 3.3).

The conceptual framework is based on many years of experience building and observing medical robot systems both in industry and academia, as well as a review of the medical robotics literature (Sec. 2.3). With the proposed framework, the goal is to: (1) systematically understand the design and characteristics of safety features, (2) enable the accumulation of prior experiences in a structured manner, and (3) facilitate sharing of knowledge and experience on safety within the community.

The remainder of this chapter discusses the benefits, limitations, and opportunities for further improvement of the proposed views (Sec. 3.4), future works (Sec. 3.5), as well as a summary of contributions described in this chapter.

3.2 Two Views of Safety Features

When designing safety features for a medical robot system, we consider two different aspects: *functional components* and *deployment options*. The functional components describe what a safety feature does and how it improves the safety of the system, and are usually derived from a set of safety requirements. The deployment options are about how to actually implement safety features and how and where to deploy them in the system. The design of safety features should take into account both aspects together because they are not independent of each other. Thus, we propose two views that reflect each aspect: the *Mechanism View* and the *System View*. These two views enable structured descriptions of safety features with consideration of run-time characteristics.

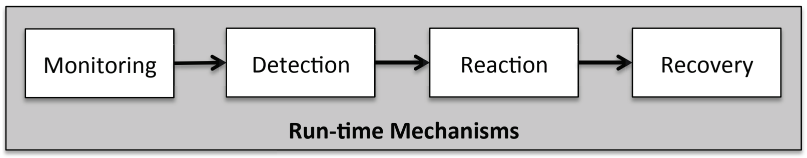
In this section, we describe the definition and characteristics of each view, together with an example that illustrates how each view is applied to some representative safety features that have been frequently used in the medical robotics domain.

3.2.1 Mechanism View

The Mechanism View (Fig. 3.1) defines functional components of safety features, and identifies the objective or behavior of a safety feature. Essentially, this view *decomposes* safety features into functional components.

3.2.1.1 Definition

The *run-time mechanisms* decompose safety features into four functionally essential components that provide a basis for understanding and presenting safety features. The four essential components are *monitoring*, *detection*, *reaction*, and *recovery*. These four components are based on our experience and observation in the domain, but similar concepts or models are also found in the literature. In medical robotics, Laible (2004)[112] presented a fail-safe



(a) Essential components of run-time safety mechanisms

Fail-safe
Fault tolerant

(b) Characteristics: Fail-safe systems focus on monitoring and detection, and fault tolerant systems put more emphasis on reaction and recovery.

Figure 3.1: Mechanism View: Four essential components of run-time safety mechanisms with their characteristics

design and an architecture of a commercial robot system for neurosurgical applications. This system obtained the CE marking and was certified by a German notified body (TÜV Product Service). Its safety functions consist of two tasks – *error detection* and *error reaction* – and each task is classified into two categories: *one-time tests* and *monitoring*. In the control systems domain, Isermann (1997)[23] presented the generic scheme of fault detection and diagnosis systems with supervision methods. This scheme defines tasks similar to the four essential components that we define: *monitoring* (“monitoring”), *diagnosis and decision* (“detection”), and *actions* that include stop operation, reconfiguration, maintenance, and repair (“reaction” and “recovery”).

Each of the four components are defined as follows:

3.2.1.1.1 Monitoring

The monitoring mechanism (*"What to and how to monitor?"*) reads quantities or states of interest from the system. It is the starting point of run-time safety because it makes run-time data available to the system and allows the system, including the human operator, to be aware of its current status and the surrounding environment. One important requirement of the monitoring mechanism is its minimal run-time overhead on the target object being monitored. Otherwise, the monitoring mechanism may introduce adverse run-time impact or burden (i.e., performance degradation) on the target.

In safety engineering, Leveson (1995)[15] presented a set of requirements of monitors in general as part of the safe design techniques:

- Detect problems as soon as possible after they arise and at a level low enough to ensure that effective action can be taken before hazardous states are reached
- Be independent from the devices they are monitoring
- Add as little complexity to the system as possible
- Be easy to maintain, check, and calibrate

3.2.1.1.2 Detection

The detection mechanism (*"How to detect events?"*) determines whether any event happened based on event specifications, and if it happened, it may include a process or subsystem that identifies detailed information about the event, such as severity, location, or timing. When designing and implementing a detection mechanism, it is crucial to minimize latency between the time when an event *actually* happened and the time when it is *determined* to have happened.

3.2.1.1.3 Reaction

The reaction mechanism (*"What to do when events are detected?"*) defines initial and immediate responses to any erroneous or undesired event. Widely used methods in the domain are the *fail-safe* emergency pause ("E-pause"), which stops robot motion, or emergency stop ("E-stop"), which disables robot motor power. Both methods subsequently generate and propagate emergency events to the rest of the system. This approach has been accepted as a working solution, mainly due to the characteristic of the medical intervention where a fail-safe system is often sufficient.[33]

The reaction mechanism is usually implemented as part of the control loop of the system, thereby avoiding time delay in the reaction mechanism due to human intervention.

3.2.1.1.4 Recovery

The recovery mechanism (*"How to recover from events?"*) represents policies, strategies, or methods to recover from erroneous states of a system. This mechanism may depend on human operators (e.g., workflow modifications such as switching to a conventional surgery when the robot fails), or automatically restore its normal state if the system is able to handle or tolerate such undesired events.

In the dependability computing domain, recovery techniques are considered as part of fault tolerance techniques. As depicted in Fig. 2.3, there are two types of recovery techniques: *error handling* that includes rollback, rollforward, and compensation, and *fault handling* that involves diagnosis, isolation, reconfiguration, and reinitialization.

3.2.1.2 Characteristics

Fig. 3.1b shows characteristics of safety features that the Mechanism View can reveal. A safety feature is considered to be *fail-safe* if the focus is on the monitoring and detection mechanism, and to be *fault tolerant* if more emphasis is placed on the reaction and recovery mechanisms. This distinction is not mutually exclusive, though. Kazanzides (2009)[33] previously pointed out the differences between fail-safe and fault tolerant systems, noting that a fail-safe system is often sufficient for medical interventions, but fault tolerance may be required for more advanced surgeries. One interpretation of this statement from the Mechanism View's standpoint is that safety features of advanced medical robot systems may have better support for the reaction and recovery mechanisms.

3.2.1.3 Example

The use of a force sensor, and force threshold check, is one of the most widely used safety features in the medical robotics domain. A force sensor software module (e.g., an object or component) periodically *monitors* force feedback from the environment. If the module *detects* excessive force beyond a predefined threshold, it initiates the emergency pause or stop as an immediate *reaction*, which then stops or powers off the robot as quickly as possible. In the meantime, a signal is generated to inform the system of the event so that other parts of the system can take appropriate *reactions* (e.g., emergency alert for users, transition to E-pause or E-stop state). When the cause of the excessive force event is removed, the system can *recover* from the emergency state to continue its previous task.

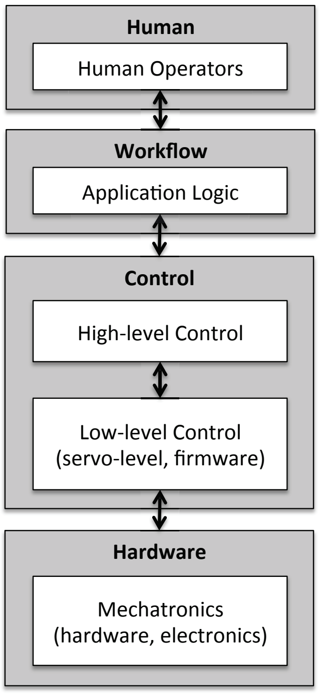
3.2.2 System View

The System View presents a hierarchical structure of medical robot systems, as shown in Fig. 3.2. This hierarchical structure forms a *layered architecture* that has been used not only within the medical robotics domain (e.g., Kazanzides (1992)[5] and Ng (1996)[153]), but also more widely adopted in the robotics domain (e.g., Visinsky (1993),[75] Hagn (2008),[103] and Kortenkamp (2009)[178]).

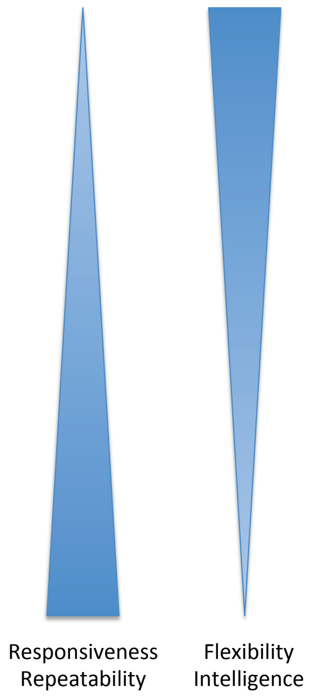
3.2.2.1 Definition

The System View captures design decisions on the deployment of safety features by identifying the layers in which functional components of the safety features are actually implemented. Those design decisions are important factors that determine the effectiveness and/or performance of safety features, and eventually the safety of the system.

As depicted in Fig. 3.2a, the System View defines a *canonical robot system architecture* with the four layers: *Hardware*, *Control*, *Workflow*, and *Human* layers. This canonical robot system architecture is based on the *generic architectures of medical robot systems*, as shown



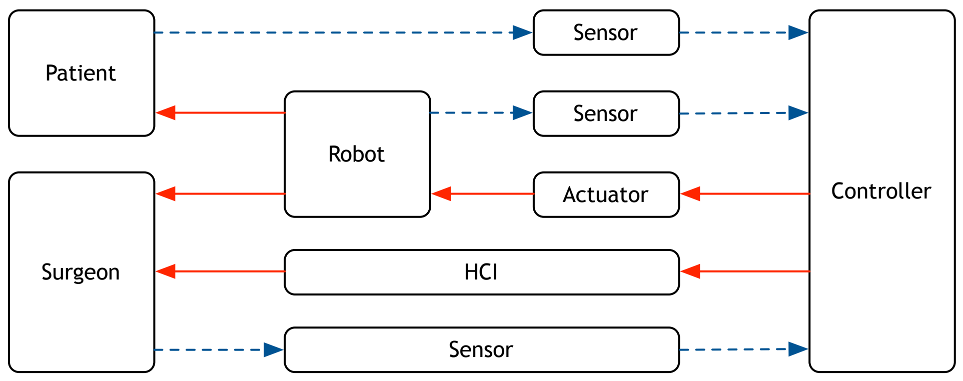
(a) Layered architecture of medical robot systems



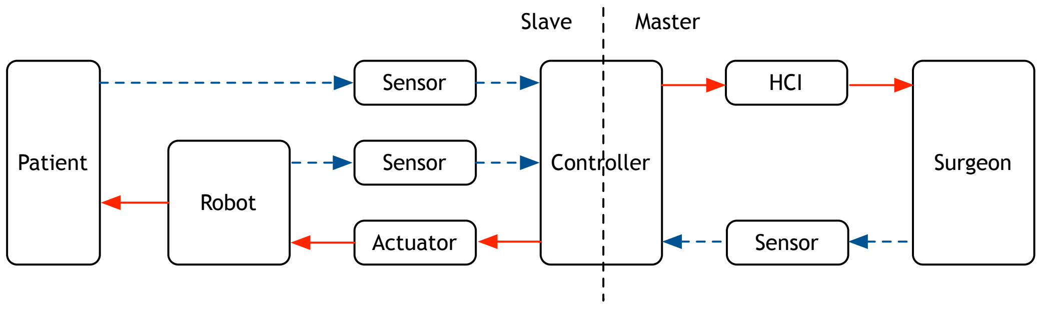
(b) Characteristics of each layer

Figure 3.2: System View: Layered architecture of medical robot systems with characteristics of each layer

in Fig. 3.3, that reflects our experience and observation on the design and architecture of various medical robot systems. The generic architectures include two different schemes: one for autonomous or cooperative control-based systems (Fig. 3.3a), and the other one for teleoperation systems (Fig. 3.3b). Although these block diagrams of the generic architectures represent highly simplified medical robot systems, they identify a set of essential components of the system and data flow among those components in a generic manner, providing an abstraction for the canonical robot system architecture. Conceptually similar schemes and architectures are found in the medical robotics literature (e.g., Speich and Rosen (2004),[179] Rosen (2012)[94]).



(a) Co-operative control or autonomous systems



(b) Teleoperation systems

Figure 3.3: Block diagrams of generic architectures of medical robot systems. Dashed blue arrows represent data flow *towards* the controller, whereas solid red arrows show output from the controller. In terms of safety, the focus is on the red arrows that could potentially lead to hazards.

The definition of each layer is as follows:

3.2.2.1.1 Hardware Layer

The hardware layer represents mechanical and electrical components of a safety feature as well as the use of, or reliance on, physical devices such as various sensors (e.g., force/torque sensor, accelerometer) and external tracking devices (e.g., optical tracker). One representative safety feature that is typically deployed to this layer is electronic circuits for E-Stop or E-Pause.

3.2.2.1.2 Control Layer

The control layer implements the control loop between the hardware and applications and provides a set of *application-independent*, but *robot-specific*, services for the upper layers (the workflow and human layers). This layer consists of two sub-layers: **high-level control** and **low-level control**. The high-level control performs tasks such as motion or trajectory planning and typically runs at hundreds of Hz. The low-level control refers to the servo-level control and is often implemented on dedicated devices (e.g., firmware on controller boards).

3.2.2.1.3 Workflow Layer

The workflow layer implements *application-specific* logic or data (e.g., surgical planning, patient-specific data processing) on top of the services that the control layer provides. When developing multiple surgical scenarios or procedures that use the same robot, the separation of the workflow layer from the lower layers (hardware and control) facilitates the development process by enabling the reuse of resources and services from the control layer. It also improves the system safety by isolating errors of the workflow from the lower layers.

3.2.2.1.4 Human Layer

The human layer represents activities or interactions with the human, such as human intervention (e.g., decision making or supervision), and thus involves issues related to human-machine interface. Of note, the human at this layer only includes people who *use* the system (e.g., surgeons, system operators, medical personnel), and does not include patients.

3.2.2.2 Characteristics

Each layer of the System View has its own characteristics that have a profound influence on the characteristics of the run-time behavior of the safety features. We consider four characteristics of each layer, as illustrated in Fig. 3.2b.

Responsiveness: The hardware layer uses dedicated hardware and electronic circuits that are optimized for specific requirements. In contrast, humans have inherent physiological limitations in sensing and processing of sensory information (e.g., limits on the temporal resolution for visual stimuli, bandwidth limits on the range of audible sound). This makes the hardware layer the most responsive layer, and the human layer the least responsive one. This concept is also found in the medical robotics literature. For example, Engel (2001)[166] presented the concept of "short reaction time" as one of the four design principles of a robot system for craniofacial surgery, and described safety features of the system in terms of reaction time.

Repeatability: Repeatability is one important property of safety features because extensive and repetitive testing can prove that the safety features work as designed, and thus meet their safety requirements. Like responsiveness, the human layer has the least repeatable characteristic due to physiological fatigue, whereas the other layers can be heavily and thoroughly tested via automated unit-testing frameworks.

Flexibility: Flexibility can be considered from two respects: design flexibility (e.g., how easily can we change parameters or behaviors?) and adaptation flexibility (e.g., how well can a layer adapt to changes in environment or conditions?). For both cases, hardware provides the least flexible options to update parameters or to change logic, whereas humans can adapt to changes in the surrounding environment. In this sense, the hardware layer is least flexible and the human layer is most flexible.

Intelligence: The hardware layer has highly specialized and limited "intelligence" (e.g., sensors, electronic elements, firmware) and thus can only handle changes of the surrounding environment that were anticipated during design. Humans, however, have experiences and expertise that can deal with unexpected events, and thus some safety features should rely on a human's decisions or supervision for safer operations.

3.2.2.3 Example

We consider the force sensor-based safety feature again that was used for the Mechanism View example, but from a different perspective. This feature can be deployed to the system in different ways, depending on the vertical distribution of each run-time mechanism. For example, we can deploy all run-time mechanisms, i.e., monitoring, detection, reaction and recovery, to the high-level control layer. The implication of this option is that this design does not rely on a human's decisions, and the safety checking would be done as part of the control loop. Another option is to deploy the first three mechanisms to the high-level control layer and to rely on the human for the recovery mechanism. In this case, the recovery mechanism becomes less responsive, harder to test, but more adaptable to changes and can take advantage of the human's experience and intelligence.

3.3 Safety Design View

The two views defined in the previous section allow us to look at two different aspects of safety features separately. Now we combine these two views into a two-dimensional plane, as shown in Fig. 3.4, with the Mechanism View on the horizontal axis and the System View on the vertical axis. This two-dimensional plane forms the design space of safety features and we call it the *Safety Design View (SDV)*.

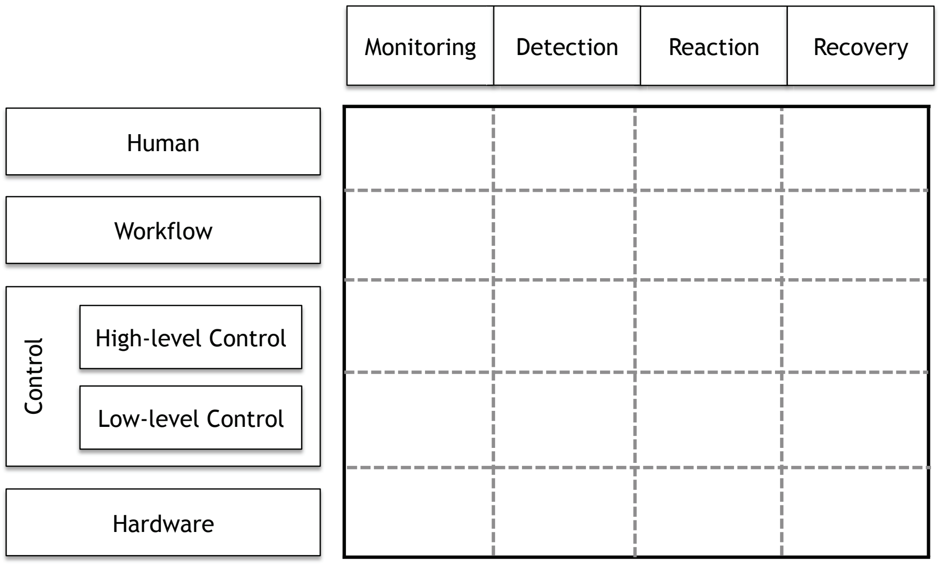


Figure 3.4: Safety Design View: Design space of safety features. The Mechanism View and System View are used as the x and y axis, respectively.

3.3.1 Definition and Characteristics

SDV is a domain-specific view to elucidate the characteristics of safety features of medical robot systems in a consistent and systematic manner. SDV represents the design space of safety features, as in Fig. 3.4, and we use the term “SDV” and the “design space” interchangeably.

SDV coherently presents the two different aspects of safety features – functional components and deployment decisions – based on the Mechanism View and the System View. The characteristics of the two views also apply to SDV in the orthogonal directions to each axis. For example, characteristics of the high-level control layer in the vertical axis apply to the entire row of the layer, and the design requirements or issues with the monitoring mechanism apply to the entire column. This coherent presentation of safety features in the design space allows SDV to capture design decisions on how to deploy safety features into the system by identifying the distribution (or combination) of functional components of safety features in the design space.

3.3.2 Case Study: Safety Features of ROBODOC®

To illustrate how to apply SDV to an actual system, we selected a commercial robot system for orthopaedic surgery, the ROBODOC® system (THINK Surgical, Inc., Fremont, CA, USA; formerly, Curexo Technology Corporation), as a case study. This system has a solid set of safety features that obtained FDA approval and CE marking, and has been in clinical

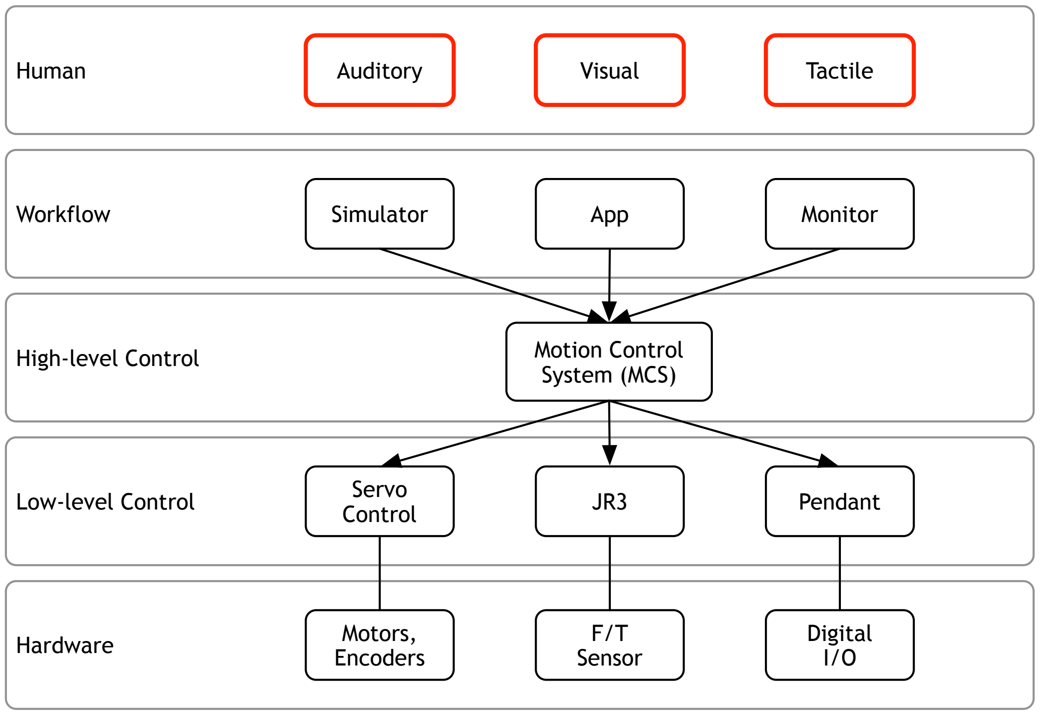


Figure 3.5: Simplified System View of ROBODOC (not all the components of the system are shown)

use since 1992. More importantly, a relatively large number of academic publications about the system and safety designs are available.

The system view of ROBODOC is depicted in Fig. 3.5 where only the key elements of the system are shown. Other elements of the system that are not shown here include the base, bone motion monitor (BMM), digitizer, cutting motor, and irrigation. ROBODOC fits the canonical system model, where the low-level control is performed on dedicated joint control boards, the high-level control (e.g., Cartesian motion and force control) is a real-time loop on a PC, and the application (workflow) runs in non-real-time on the same PC. The surgeon interfaces with the system via graphical menus and a hand-held control pendant; in addition to the buttons for selecting menu items, the pendant includes pause and stop buttons that freeze robot motion and turn off robot motor power, respectively.

For this case study, we selected a subset of the safety features of the ROBODOC system, which have been also repeatedly used in the other medical robot systems, based on our safety survey results (refer to Sec. 2.3.1 for details). Sec. 6.2.1.2 presents a more complete list of the safety features of ROBODOC. The safety features that we consider in this section include:

1. **Force threshold checks** (e.g., Kazanzides (1992b),[107] Dombre (2003)[100])
2. **Redundant sensors** (e.g., Cain (1993),[139] Degoulange (1998),[98] Engel (2001)[166])
3. **E-pause and/or E-stop** (e.g., Kazanzides (1992a),[5] Guiochet (2002)[105])
4. **Hardware or software limiting of speed or torque** (e.g., Kazanzides (1993),[136] Jakopec (2003),[169] Laible (2004)[112])
5. **Dynamic constraints** (a.k.a., Safety volume, virtual fixture) (e.g., Kazanzides (1995),[137] Davies (1995),[151] Zhu (2000)[97])

In order to complete the SDV, it was first necessary to define some conventions. For example, many safety features rely on hardware to measure a physical quantity, such as position or force, and this feedback is acquired by one of the software layers (typically low-level or high-level control). In this case, we place solid dots in the monitoring (M) column corresponding to the rows for the hardware layer (HW) and the appropriate software layer (e.g., LC or HC). With this convention, we applied SDV to these safety features and Fig. 3.6 shows the results.

The force limit checking (Fig. 3.6a) shows a typical run-time safety mechanism where

	M	D	R	Re
HU				●
WF			●	●
HC	●	●	●	
LC				
HW	●			

(a) Force limit check

	M	D	R	Re
HU				●
WF			●	
HC		●	●	
LC	●	●	●	
HW	●			

(b) Redundant sensors

	M	D	R	Re
HU				●
WF			●	
HC			●	
LC			●	
HW	●	●	●	

(c) E-Stop (hardware-based)

	M	D	R	Re
HU				●
WF			●	
HC	●	●	●	
LC				
HW	●			

(d) E-Pause (software-based)

	M	D	R	Re
HU				●
WF			●	
HC	●	●	●	
LC				
HW				

(e) Software speed limit

	M	D	R	Re
HU				●
WF			●	
HC	●	●	●	
LC				
HW	●			

(f) Safety volume

Figure 3.6: Representation of safety features of ROBODOC using Safety Design View. The horizontal and vertical axes correspond to the Mechanism View and the System View, respectively.

monitoring is done at the hardware layer, whereas detection and reaction occur at the control layer, and recovery relies on the human. Compared to this, the redundant sensor (e.g., encoder mismatch) has more mechanisms implemented in the low-level control layer (Fig. 3.6b). ROBODOC included both an E-stop and an E-pause, where the former is initiated by hardware and the latter by software. On SDV, it is straightforward to discern the differences between them, as in Figs. 3.6c and 3.6d. The motor speed limit is implemented as part of the high-level control loop (Fig. 3.6e) because that is where the trajectory generation was performed (this safety feature limits the commanded joint speed, which could otherwise become excessive near a kinematic singularity); for torque-controlled robots, a torque limit would be used instead. The safety volume (Fig. 3.6f) appears similar to the motor speed or torque limits, but also relies on the monitoring mechanism in the hardware layer to check the actual robot position measured by the encoders.

In Fig. 3.6, we found a “pattern” in the reaction and recovery mechanisms of the safety features. This was, of course, part of the ROBODOC system design and SDV “captured” it. For example, we note that all reactions involve at least one control layer (typically to stop motion or initiate power-off), but also affect the workflow by stopping the normal sequence to display an error message to the user. In all cases shown, the human (surgeon) is involved in the recovery action. For the force threshold check, the workflow also initiated part of the recovery by automatically backing away along the measured force direction. It is possible for recovery to be performed without human involvement, where the system silently recovers from an unsafe condition, but that was not the case for any of the above safety features.

3.4 Discussion

Our approaches to understanding and representing safety can be summarized as three elements: the Mechanism View, the System View, and SDV. They are designed to better present safety features of medical robot systems by simultaneously identifying run-time safety mechanisms and capturing design decisions on deployment options. This section discusses some of the design details of these elements as well as some limitations that we experienced throughout this work.

We note that not all safety features can be represented by the four components of the mechanism view. For example, one possible safety feature is to design the robot with low-power motors so that it has limited speed and/or torque. This is essentially a safety feature that is implemented by changing a *property* of the system (it is different from a software-imposed speed or torque limit, as presented in Fig. 3.6e). In our experience, these types of safety features are typically confined to hardware design properties and are therefore outside the scope of this work.

The current definition of the system view does not yet consider another type of human – *patients* – because the focus is on engineered systems. However, patient safety is also a crucial aspect and there is a body of work in this area, i.e., physical interactions between the robot and human, such as Haddadin (2009,[71] 2012[180]).

The locations of black dots in the design-space of safety features (SDV) captures design decisions about the deployment options of safety features, which reflects system designers' experience and expertise. One possible use of SDV is to document and collect representative safety features using SDV, and establish "canonical templates" of such safety features, which describe or define how to design, implement, and deploy safety features as "best practices" or design guidelines.

In the design-space of safety features, one limitation is that it can capture the data and control flow *horizontally*, but is hard to do *vertically*. The Mechanism View has a notion of "flow" from monitoring to recovery, but the System View does not. Sometimes safety features behave in specific orders and may need specific timing requirements. For example, if the workflow layer detects a safety violation, it may initiate part of the reaction, but would likely need to request the control layer to stop motion or power off the motors. The ability to capture this sequence of actions between the system layers is a possible extension to SDV.

Through this work, we noticed that the effectiveness of SDV depends on the SDV user's experience and the degree of understanding of safety features, as well as the design and architecture of the system. Because SDV is based on the concept of abstraction (of mechanisms) and layers (of deployment options), the deeper the understanding is, the more effective SDV becomes.

Despite its limitation in terms of expressiveness, SDV was effective and helpful for clearly describing, documenting, and conveying the idea and design of safety features, based on our experience. Especially, representing safety features of ROBODOC using SDV (Sec. 3.3.2) led to in-depth design discussions of its current safety design.

3.5 Conclusions and Future Works

We presented the Safety Design View (SDV), a conceptual framework that can capture and describe both the design-time and the run-time characteristics of safety features of medical robot systems in a systematic and structured manner. SDV is based on two views, the Mechanism View and the System View, each dealing with safety mechanisms and design decisions on the deployment of safety features. The goal of SDV is to: (1) explicitly and intuitively describe safety features in a consistent and structured manner, (2) collect "good" practices on the design of safety features, and (3) facilitate sharing of knowledge and experience on safety within the community.

There are active discussions for safety standards of medical robotics in the community. Prior to these discussions, or for more effective discussions, it is our hope that such a common ground as SDV could be helpful. Also, it is possible that SDV may become an additional method for conveying the safety design within a medical device company, and between the company and regulatory agencies.

SDV has two possible directions for further works: one is to further elaborate and improve the design and definition of SDV, and the other is to review and organize our safety survey results into a set of "canonical templates" of safety features.

3.6 Contributions

The contributions of this thesis described in this chapter are as follows:

1. Mechanism View (Sec. 3.2.1)

– *Identification of four essential components of safety features*

We identified (1) four essential functional components that define run-time mechanisms of safety features, and (2) the Mechanism View that consists of those four components. The Mechanism View is not restricted to the medical robotics domain, but is applicable to any other domain.

2. System View (Sec. 3.2.2)

– *Identification of canonical architecture of medical robot systems that captures system designer's decisions*

We identified (1) the canonical architecture of medical robot systems, and (2) the System View that captures design decisions on the deployment of safety features, which determine the effectiveness of safety features, and eventually influence the safety of the system. The System View contains domain-specific components, and is applicable to the medical robotics domain as well as the general robotics domains.

3. Safety Design View (Sec. 3.3)

– Definition of the design space of safety features in medical robotics

We defined a domain-specific view, Safety Design View (SDV), that can explicitly describe the characteristics of safety features of medical robot systems and can capture deployment decisions. SDV enables the description of safety features in a consistent and structured manner, allows to collect best practices on the design of safety features, and facilitates sharing of knowledge and experience on safety with the community.

Chapter 4

Generic Component Model

4.1 Introduction

The previous chapter described the Safety Design View (SDV) that lays the conceptual foundation for this thesis. Specifically, the mechanism view identifies the four essential components of safety features as monitoring, detection, reaction, and recovery. In addition, the system view defines the four layers of the canonical architecture of medical robot systems as the hardware, control, workflow, and human layers. As will become clearer throughout this thesis, SDV allows us to define and describe safety features of robot systems in a generic, consistent, and structured manner.

Component-based software engineering (CBSE) is now well established in various domains, including the automotive,[181] avionics/embedded,[182] and robotics[45] domains. Some key characteristics of the component-based approach include (1) generic abstraction in software design, implementation, and deployment, (2) flexible configuration of software (potentially run-time reconfiguration), and (3) promotion of reuse of third-party software.[183] As proved both in academia and in industry, system designers have benefited from these characteristics when designing and developing large, complex software systems.

In robotics, numerous component-based software packages have been released to facilitate robotics research.[45,47] Representative examples include Robot Operating System (ROS)[52] and OROCOS (Open Robot Control Software).[50] These software frameworks have mostly focused on the *functional* aspects of robot systems, whereas the *non-functional* properties are not their primary interest. Attention to these non-functional properties, especially safety, is gradually increasing within the robotics community (e.g., Corke, 2011;[66] Woodman *et al.*, 2012;[81] Tadele *et al.*, 2014;[82] Jung *et al.*, 2014[184]).

Although some robotics packages (e.g., OROCOS) maintain a state machine that distinguishes normal states from error states, and other packages (e.g., OPRoS[53]) provide a facility for fault management and recovery, support for systematic safety is still in its infancy. In particular, most of the software frameworks in robotics suffer from the key limitation that they are unable to model the effects of errors coming from the *outside* in a systematic and structured manner. This is primarily because error semantics of those frameworks are confined *within* the component boundary.

The literature outside the robotics domain provides a body of work on component-based semantics for safety, which explicitly models error propagation *across* the component boundary. They are called the *model-driven safety analysis techniques* and the goal is to identify all possible cases that can potentially lead to hazardous situations, and to demonstrate that such probabilities are sufficiently low.[61] Prior work and the state-of-the-art in this field are presented in Sec. 4.2, along with a review of state-based approaches in medical robotics and error propagation semantics. However, those model-driven analysis methods may not be directly applicable to component-based robot systems where component models are typically defined at the *code* level by the component framework; model-driven engineering is not yet widely adopted by the robotics community.[48] Thus, to apply model-driven safety analysis to robot systems, it would be necessary to reverse engineer existing code to obtain the models. Although it is technically possible to perform reverse engineering, it would require considerable effort. Recent work in robotics has begun to apply model-driven engineering to the development of component-based robot systems to benefit from its maintainable and deterministic characteristics.[48]

In medical robotics, a programming model that is commonly accepted by the medical robotics community does not yet exist, although there are experience reports on software engineering techniques such as agile methods (Gary *et al.*, 2011[173]) and CBSE (Jung *et al.*, 2014[49]). The ideal model would be based on the component-based approach, as in robotics, with support for error propagation. It would be even more desirable if such a model is inherently generic, extensible, and customizable so that it can be specialized in a flexible manner to adapt to existing component models in robotics, thereby achieving reusability and interoperability. Our work aims to develop such a model.

In this chapter, we describe a generic, abstract component model, called the Generic

Component Model (GCM). As presented in Sec. 4.3, GCM is essentially an abstraction of various component models with minimal structural elements, yet it is expressive enough to represent the complete description of the system status without relying on a particular component model. GCM enables a structured approach to designing and implementing safety features, both at design-time and run-time. The basis of our approach is a state-based semantics for component-based robotic systems, which explicitly represents the operational status of the system at run-time in a systematic and structured manner with support for error propagation. Sec. 4.4 discusses our approach in terms of the requirements for model-based safety analysis techniques, addresses the room for improvement, and identifies differences between our approach and the model-driven safety analysis techniques, mainly due to domain-specific characteristics. Sec. 4.5 wraps up this chapter, and Sec. 4.6 provides a summary of contributions of the work described in this chapter.

4.2 Related Works

Our work described in this chapter is a *state*-based semantics for the *safety* of *component-based* robot systems, which can explicitly represent the run-time status of the system with support for *error propagation*. The four key concepts are in italics: state-based, safety, component-based, and error propagation. With these key concepts, this work has three areas of related work: (1) safety analysis techniques, (2) state-based approaches to safety, and (3) error propagation. There has been a body of research on these areas in the literature of different domains, such as safety engineering, safety-critical systems engineering, and component-based software engineering. This section provides an overview of prior work on these topics.

Sec. 4.2.1 presents an overview on various safety analysis techniques, including both traditionally established techniques and state-of-the-art methods. Sec. 4.2.2 describes state-based approaches to safety in the medical robotics domain, and Sec. 4.2.3 provides a brief review on the error propagation semantics of the component models in robotics.

4.2.1 Safety Analysis Techniques

Many different safety analysis techniques have been proposed and are in use. Each method has its own formalism with different coverage and goals, leading to different characteristics (e.g., quantitative vs. qualitative, inductive vs. deductive). In the literature, there exist articles that provide an overview, current practice, or comparative study of various safety analysis techniques[i]. This section presents a high-level overview of these techniques; more detailed descriptions of each technique can be found in the literature.

Fig. 4.1 shows a classification of various safety analysis techniques based on the review of the literature. Safety analysis techniques are divided into two categories: the *traditional* methods that do not consider system architectures, and the *model-driven* methods that take a system model or an architecture into account.

[i]The literature includes Leveson (1995),[15] Grunske *et al.* (2005),[61,62] Lisagor *et al.* (2006),[185] Grunske and Han (2008),[186] Bloomfield and Bishop (2010),[187] Jamboti and Liggesmeyer (2012),[188] and Hatcliff *et al.* (2014).[42]

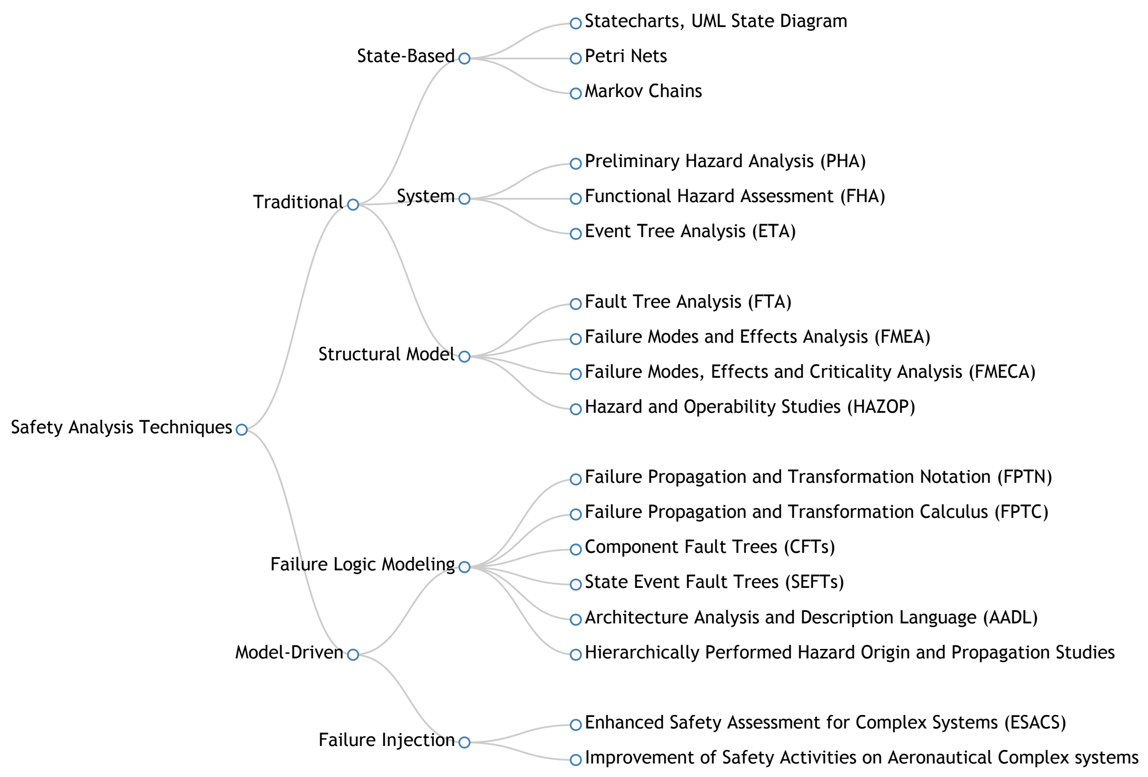

Safety Analysis Techniques
Traditional
State-Based
Statecharts, UML State Diagram
Petri Nets
Markov Chains
System
Preliminary Hazard Analysis (PHA)
Functional Hazard Assessment (FHA)
Event Tree Analysis (ETA)
Structural Model
Fault Tree Analysis (FTA)
Failure Modes and Effects Analysis (FMEA)
Failure Modes, Effects and Criticality Analysis (FMECA)
Hazard and Operability Studies (HAZOP)
Model-Driven
Failure Logic Modeling
Failure Propagation and Transformation Notation (FPTN)
Failure Propagation and Transformation Calculus (FPTC)
Component Fault Trees (CFTs)
State Event Fault Trees (SEFTs)
Architecture Analysis and Description Language (AADL)
Hierarchically Performed Hazard Origin and Propagation Studies
Failure Injection
Enhanced Safety Assessment for Complex Systems (ESACS)
Improvement of Safety Activities on Aeronautical Complex systems

Figure 4.1: Classification of safety analysis techniques in the literature (based on Grunske *et al.* (2005)[61] and Lisagor *et al.* (2006)[185])

4.2.1.1 Traditional Methods

The traditional methods refer to a set of established safety analysis techniques that were originally developed for mechanical or electrical systems, and then have been widely adopted in industry, such as automotive, aerospace, and robotics.

The traditional methods are divided into three subclasses: (1) the *system-level* techniques, (2) the *structural model* techniques, and (3) the *state-based* techniques. The system-level techniques regard the system as a black-box, i.e., look at the system on a coarse and abstract level to examine the effects of system-level failures. Because these techniques are applied to the system level, an internal structure of the system or a system architecture (e.g., whether the system architecture is monolithic or component-based) is of no concern. Techniques of this class include Preliminary Hazard Analysis (PHA), Functional Hazard Assessment (FHA), and Event Tree Analysis (ETA). In contrast, the structural model techniques, such as Fault Tree Analysis (FTA), Failure Modes and Effects Analysis (FMEA), Failure Modes, Effects and Criticality Analysis (FMECA), consider the internal structure of the systems[ii]. FTA and FME(C)A are among the most widely used techniques in industry. The state-based techniques model system safety using custom states, rather than a two states abstraction (normal vs. failed). These techniques use state-based methods to analyze and verify the safety properties of the system. Examples of this class include Statecharts/UML state diagram, Petri Nets, and Markov Chains. Other examples include Atlee and Gannon's work (1993)[189] on model checking to verify safety properties for event-driven systems. Heimdahl and Leveson (1996)[190] presented the Requirements State Machine Language (RSML) that automatically analyzes state-based requirements specifications in terms of completeness and consistency.

These traditional safety analysis techniques predate CBSE and do not consider the characteristics of component-based systems, such as the hierarchical structure, reuse of components, and component composition. Although the traditional techniques can be directly applied to component-based system without modification,[61] the lack of support for the inherent characteristics of CBSE makes the traditional safety analysis methods less

[ii]For example, the FMEA analysis process begins with identifying the structural elements in a hierarchical manner, and the decomposition of the system structures is deductively performed down to the basic functional elements of the system.

effective for analyzing large and complex component-based systems.[61,62]

4.2.1.2 Model-Driven Methods

Since the traditional safety analysis techniques were developed before the emergence of CBSE, they cannot leverage the characteristics of CBSE and this leads to challenges when applying traditional methods to component-based systems. Grunske *et al.* (2005) presented a summary of such challenges as follows:[61]

1. **Composition of the safety property**: Although safety is a system property that has to be considered as a whole, the behavior of individual components in terms of safety are also essential. Thus, safety analysis techniques for component-based systems need to consider system safety using *component behavioral models* in order to construct system-level safety cases from component-level quality properties such as correctness, availability, and reliability.
2. **Failure propagation**: Component failures occur due to, not only internal failures, but also *failures propagated from* other component(s).
3. **Integration of development process**: Safety analysis techniques must integrate into the overall development process of the system as tightly as possible, for reuse and compatibility purposes.
4. **Complexity**: When analyzing large and complex component-based systems, they easily suffer from the state-explosion problem. It is important to find a proper level of

abstraction that makes a model *expressive enough and yet analyzable*[iii].

The model-driven safety analysis techniques address these challenges by incorporating the characteristics of CBSE into the model, and this approach has been getting more attention from researchers and practitioners.[185, 191] These techniques are further classified according to their approach: the *failure logic modeling* approach and the *failure injection* approach.

The failure logic modeling approach models the failure behavior of the system in an incremental fashion as part of the design process. The key idea of this approach is to break down the complex system-level analysis into the manageable failure behavior of individual components. Then, artifacts of the traditional methods (e.g., FTs, FME(C)A tables) can be automatically generated based on the results of the modeling process. Examples of these techniques are Failure Propagation and Transformation Notation (FPTN) (Fenelon *et al.*, 1994[58]), Hierarchically Performed Hazard Origin and Propagation Studies (HiP HOPS) (Papadopoulos *et al.*, 2001[192]), Component Fault Trees (CFTs) (Kaiser *et al.*, 2003[59]), Fault Propagation and Transformation Calculus (FPTC) (Wallace, 2005[193]), State Event Fault Trees (SEFTs) (Grunske *et al.*, 2005[62]), and Architecture Analysis and Description Language (AADL) (Feiler *et al.*, 2006[182]). Grunske and Han (2008)[186] presented a comparative study of existing safety evaluation methods with AADL from three perspectives: modeling support, process support, and tool support. Jamboti and Liggesmeyer (2012)[188] also presented another comparison of model-based safety analysis techniques where they proposed a structured way of combining CFTs into an integrated CFT (iCFT). Domis and Trapp (2008)[64] proposed

[iii]Grunske *et al.*, (2005) states that "techniques on a practical granularity level and with a limited scope (i.e., expressing just the facts of interest) are necessary."

the Safe Component Model (SCM) that separates non-functional properties from functional properties, and handles specification and realization separately.

The failure injection approach uses formal design models and modern model checking techniques to automatically deduce failure modes of the system that can potentially lead to unsafe conditions. This approach is motivated by the increasing acceptance of formal models in industry, such as SCADE (Safety Critical Application Development Environment, Esterel Technologies) or Simulink (Matlab), as well as the advances of formal verification methods.[185] Examples of safety analysis techniques with this approach includes the Enhanced Safety Assessment for Complex Systems (ESACS) project and the Improvement of Safety Activities on Aeronautical Complex Systems (ISAAC) project.

4.2.2 State-based Approaches in Medical Robotics

In medical robotics, state-based approaches have been used for safety, mostly as a means to construct a system with deterministic and structured behaviors. Kazanzides *et al.* (1992)[5] presented a state-based approach devised for a commercial orthopaedic surgery robot, called ROBODOC®. This state-based approach controls the procedural flow of the system by the values of state variables. This approach was empirically proven to be effective for reducing the complexity of the application program, and also facilitated structured error or exception handling mechanisms.

Cleary *et al.* (2004,[172] 2006[117]) adopted the state machine as the fundamental design principle for inherent safety, determinism, repeatability, and testability, and applied it to the development of an open-source component-based software system for image-guided surgery applications, called the Image Guided Surgery Toolkit (IGSTK). Other examples that adopted the concept of states and state machine as part of the system design principle include Varley (1999),[113] Guthart and Salisbury (2000),[102] Guiochet and Vilchis (2002),[105] Korb *et al.* (2003),[171] Laible *et al.* (2004),[112] and Fodero *et al.* (2006).[175]

There is also a set of prior work that applied the traditional safety analysis techniques, such as FMECA and FTA, to the design process of medical robot systems. A short list of examples includes Pierrot *et al.* (1999),[161] Guiochet and Vilchis (2002),[105] Korb *et al.* (2003),[171] Laible *et al.* (2004),[112] and Korb *et al.* (2005).[115] However, no prior work that adopted the model-driven safety analysis techniques is found in the medical robotics literature, to the best of our knowledge. This is partly because CBSE has not yet been adopted as a standard programming model within the medical robotics domain, although there is a recent report on experiences with CBSE within medical robotics (Jung *et al.*, 2014[49]).

4.2.3 Error Propagation Semantics

As described in the previous sections, the model-driven safety analysis techniques for component-based systems include a semantics for error or failure propagation. Among these techniques, the AADL Error Annex[194] has recently attracted much attention because of its success and wide adoption in the avionic and automotive domains.[186] The AADL Error Annex allows system designers to annotate AADL components with dependability information, such as error propagation policies and fault-tolerance policies. This information is defined by the AADL error models that also define the general hardware and software component error models.

In contast, no component model with the concept of error propagation is found in the robotics domain, despite a proliferation of component-based robot software frameworks. Some component models in robotics, such as Orocos and OPRoS, maintain a state machine that makes a distinction between normal and non-normal states. However, those component models are introspective, i.e., they only consider errors generated within the component boundary, and do not take into account (the effects of) errors propagated from other components. This lack of error propagation semantics limits the expressiveness of the component model because components typically require data and/or services from other components. Recently, Tadele (2014)[82] recognized the significance of component interaction and error propagation semantics in the context of risk assessment and safety analysis.

Fig. 4.2 illustrates an overview of the state-of-the-art and emerging areas of different domains – medical robotics, robotics, and outside robotics – in terms of three topics: (1) the adoption of CBSE, (2) support for error propagation semantics, and (3) research on the model-based safety analysis techniques. The check mark represents that a topic has been already adopted or established in a domain, and the circle indicates emerging topics or areas for improvement. In medical robotics, it is our understanding that there is no consensus on a standard programming model within the community, and that error propagation and model-driven safety analysis techniques have not yet been investigated. In robotics, CBSE is a de facto programming model. In spite of the wide adoption of CBSE by researchers and practitioners, no component model with support for error propagation is found. Outside robotics, i.e., in the general software engineering domain including CBSE, the state-of-the-art research activities center around the model-driven safety analysis techniques.

This figure leads to two obvious areas for contributions: (1) the introduction of the concept and semantics of error propagation to the robotics domain, and (2) the proposal of CBSE and the error propagation semantics as an effective programming model to build large, complex, and safety-oriented medical robot systems.

Domain	Topics		
	CBSE	Error Propagation	Model-driven Safety Analysis
Medical Robotics	○	○	
Robotics (Non-medical)	✔	○	
Outside Robotics (General SW Eng)	✔	✔	○

Figure 4.2: Room for improvement for each topic in each domain (SW Eng: Software Engineering, ✔: already established or adopted, ○: emerging areas, i.e., room for improvement)

4.3 State-Based Semantics for Component-based Software Systems

The essential basis of Component-based Software Systems (CBSS) is a *component model*. A component model defines what components are (*semantics*), how components are defined, constructed, and represented (*syntax*), and how components are composed, assembled, or deployed (*composition*).[195] There is a wide variety of component models. A short list of such component models outside the robotics domain includes Fractal,[196] Palladio,[197] AUTOSAR,[181] and AADL.[182] Due to its wide variety, there exists a field of study in CBSE on the identification and classification of component models (e.g., Lau and Wang, 2007;[195] Birkmeier, 2009[198]).

In robotics, component models are defined by robot software frameworks or middlewares that provide the run-time environment for robot systems. A few examples of such software packages include ROS,[52] Orocos,[50] BRICS,[48] cisst,[49] and OPRoS.[199] However, the error semantics of those component models are confined *within* the boundary of components, and do not explicitly handle errors caused by other components through component interaction, i.e., error propagation. Error propagation is a prerequisite for improved safety under component-based robot systems because errors or failures of one component may cause adverse effects on other components via connections between the two components.

The goal of the work described in this section is to define a generic semantics that explicitly captures and presents the operational status of component-based robotic systems at *run-time* with support for error propagation. In Sec. 4.3.1, we first define the Generic Component Model (GCM) that enables us to present the concepts without delving into the details of a particular component model or implementation details. The concept of two layers that enable the separation of concerns is introduced in Sec. 4.3.2. The GCM defines the *states* and *state machine* (Sec. 4.3.3) that are adapted from the *fault-error-failure model* of the dependability formalism.[14] Each key element of the GCM is assigned with an instance of this state machine and its run-time state is represented by state variables (Sec. 4.3.4). Sec. 4.3.5 presents the *event* that plays a key role in initiating state transitions. Sec. 4.3.6 introduces the *filter*, a unit of computation that provides a generic representation of arbitrary algorithms and generates events based on the results of computation. Sec. 4.3.7 describes the *error propagation* semantics that defines how an event from one component is propagated to other components in a systematic and structured manner. Sec. 4.3.8 presents the *state-dependent operational modes* that enable different component behaviors depending on the component state.

4.3.1 Generic Component Model

The Generic Component Model comprises the *structural elements* and the *state-based semantics* to represent the *operational status* of component-based robot systems in an explicit, structured, and component model-independent manner.

The starting point is to identify a *minimal but essential set of structural elements* that can be used as an abstraction of various component models. Such an abstraction with the minimal structural elements would enable us to describe and represent the *operational status* of the system in a generic and component model-independent manner. This leads us to visit the definition of a component. One widely accepted definition is by Szyperski (2002):[43] *"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 third-party composition."* Lau and Wang (2007)[195] presented another concise definition: *"A generally accepted view of a software component is that it is a software unit with provided services and required services. The interface of a component consists of the specifications of its provided and required services."* The commonalities between these two definitions are that a component has *interfaces*, and that these interfaces define *services*. This leads to the definition of the minimal structural elements: a provided interface and a required interface, as depicted in Fig. 4.3. A component may have zero, one, or multiple instances of provided and/or required interfaces. For generality, the minimal structural elements of GCM does not include more fine-grained structural elements than interfaces. However, this abstraction is generic enough to apply to different robotics frameworks. For example, provided interfaces in *cisst*[49] are collections of command execution elements that provide related services; in this case, one GCM provided interface would map to one *cisst* provided interface (collection of services). The same is true for *cisst* required interfaces. In contrast, a ROS[52] node (component) provides individual services and topics and thus each service or topic would correspond to a separate GCM interface (e.g., a topic publisher is considered a

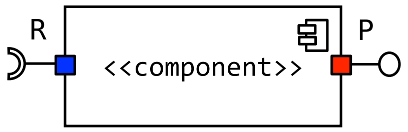


Figure 4.3: The Generic Component Model: The only structural elements are the provided interface ("P") and the required interface ("R").

provided interface and a topic subscriber is considered a required interface).

GCM models service dependencies between interfaces as *connections*, i.e., point-to-point communications. The notion of point-to-point communication is a generic, abstract concept that is independent of a particular type of data communication schemes, such as publish/subscribe, message passing, remote invocations, notifications, shared spaces, and message queuing.[200] For example, *cisst* connections and ROS services can be directly mapped to GCM connections. In case of the ROS topic, multiple GCM connections can be defined between a topic publisher and each of the topic subscriber(s).

With the structural elements, the GCM defines the *state-based semantics* to systematically capture and represent the operational status of components with support for error propagation. The operational status is described in accordance with the state-based semantics, which is in essence represented by the *internal processing pipeline*. As shown in Fig. 4.4, the internal processing pipeline defines how data from other components or the environment are processed, how events are generated, and what initiates states transitions.

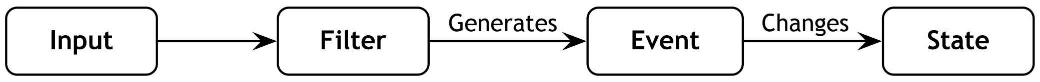


Figure 4.4: Internal processing pipeline of Generic Component Model. *Inputs* from other components or from the environment are processed by *filters*, which generate *events* that may possibly initiate *state* changes.

The internal processing pipeline consists of four elements: *state*, *event*, *filter*, and *input*. These elements are described in more detail in the following sections.

4.3.2 Two Layers and Two Views

One of the design principles of modern software engineering is the *separation of concerns*, which brings benefits such as reduced complexity, improved reusability, and simpler evolution.[201] In the GCM, this principle leads to a system architecture that splits the system into two layers: the *component framework layer* and the *application logic layer*. These two layers are defined within the boundary of a GCM component, as shown in Fig. 4.5.

The *component framework layer* represents the infrastructure of the system, which provides the application with the component-based environment and related services such as component composition and run-time management. Each component framework has its own definitions for the component model, the thread execution model, or data exchange mechanisms, which have nothing application-specific. That is, this layer has the *framework-*

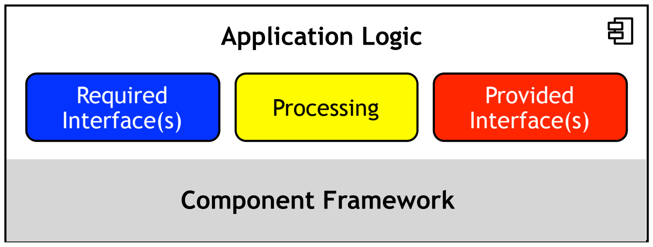


Figure 4.5: Two layers of Generic Component Model. The component framework layer provides *application-independent* but *framework-specific* services, and the application logic layer uses these services to implement *application-specific* logic or algorithms.

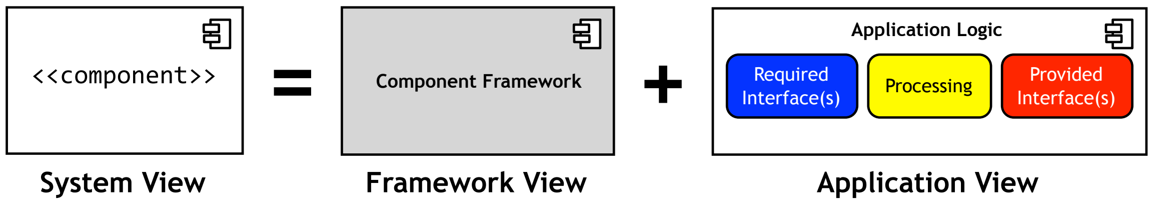


Figure 4.6: Two views of Generic Component Model. The framework view and the application view represent the status of the component framework layer and the application logic layer, respectively. These two views are combined into the system view that presents the overall status of the system.

specific but *application-independent* characteristics.

The *application logic layer* uses services provided by the component framework layer to implement application-specific content via three elements: *required interfaces*, *processing unit*, and *provided interfaces*. The interfaces enable inter-component data exchange and the processing unit implements the application-defined behavior of a component, which leads to the *framework-independent* but *application-specific* characteristics of this layer.

These two layers allow the system designers to consider the system from two different viewpoints, and help the GCM manage events of each layer separately. This separation of layers naturally leads to two different *views* of the system: the *framework view* and the *application view*, as depicted in Fig. 4.6. Each view only considers the component framework layer and application logic layer, respectively, and the characteristics of each view follow those of each layer. These two views are “combined” into the *system view* that represents the overall status of the system (refer to Sec. 4.3.4 for how to combine these two views).

4.3.3 States and State Machine

The GCM allows us to describe the run-time information about CBSS without relying on a particular component model. The question here is how to define the "run-time information" in such a way that it is not only expressive enough to describe the current status of a system in a comprehensive manner, but also generic enough to capture various properties of a system in a consistent way. One approach to achieving both goals – expressiveness and generality – is to use *states* as a means of abstraction.

The semantics of the GCM state machine is based on the *dependability formalism* of the dependability computing domain,[14] which comprehensively defines the fundamental concepts and taxonomy to address the dependability and security issues of computing systems. This formalism includes the *pathology of failure* that defines the creation and manifestation mechanisms of faults, errors, and failures. These are the three major threats to dependability and Table 4.1 shows their definitions. There is a causal relationship among these threats: faults are *activated* and cause one or more errors, errors are *propagated* to the service interface, a service failure occurs, and the failure *causes* faults in the other systems. Salfner *et al.* (2010)[202] also presented a figure that clearly illustrates this causal relationship, as shown in Fig. 4.7.

Table 4.1: Definitions of three major threats to dependability and security[14]

Threat	Definition
Fault	Adjudged or hypothesized *cause* of an error
Error	Part of the total *state* of the system that may lead to its subsequent service failure
Failure	*Event* that occurs when the delivered service deviates from correct service

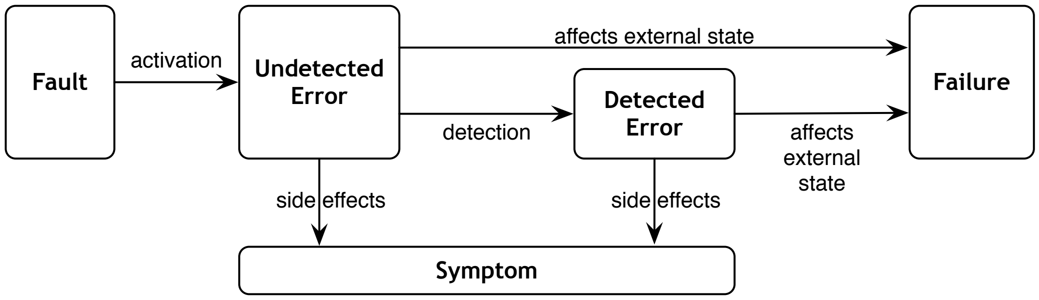


Figure 4.7: Interrelations of faults, errors, symptoms, and failures (adapted from Salfner (2010)[202])

4.3.3.1 States

The GCM defines three states: *Normal*, *Warning*, and *Error*. *Normal* is an initial state where no threat is present and the system is working properly and correctly as specified. *Warning* is an informative state where an error has not yet occurred but may possibly happen, and the system is still working properly up to some extent (e.g., degraded mode or performance). With this state, the system can have a chance to prevent errors in advance by reacting to events accordingly (e.g., fault prevention using prognostic information). *Error* is a state where the delivered service deviates from the correct service. Note that this definition is slightly different from the original definition where there is a distinction between an error

Table 4.2: Definitions of states in Generic Component Model

State	Definition	Description
Normal	A state where no threat is present and the system is working properly as specified.	Initial state; A state which is free from any abnormal event.
Warning	A state where an error has not yet occurred but may possibly happen.	Informational state; Services are provided as in the *Normal* state.
Error	A state where the delivered service deviates from the correct service.	Failure; Cannot guarantee any service.

and a failure. This is because the definition of *services* is different. In the original context, a service is defined for the system, whereas a GCM service is defined for every essential structural element of the system, i.e., components and interfaces.

Table 4.2 summarizes the definitions of the three states with more intuitive descriptions.

4.3.3.2 State Machine

The GCM state machine is an event-driven finite-state machine with the three states – *Normal*, *Warning*, and *Error* – and six state transitions among the states, as shown in Fig. 4.8. Transitions are defined between two different states; given a state, any state except the same state can be the next state. The GCM state machine does not explicitly define transitions to the same state, although the GCM handles such transitions by updating information about the event that initiated the last state transition (refer to Sec. 4.3.5 for more details).

There are two types of state transitions: transitions for *error detection* and transitions

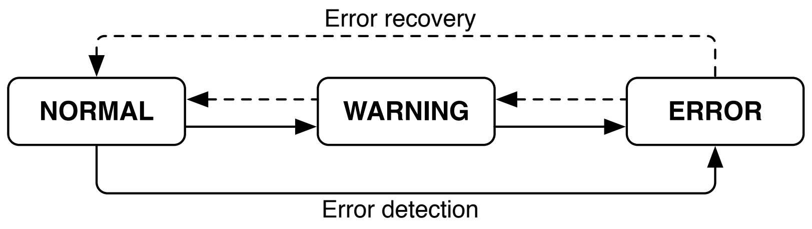


Figure 4.8: State machine of Generic Component Model: Solid arrows represent transitions due to the occurrence of abnormal events, whereas dashed arrows show transitions when recovering from non-normal states.

for *error recovery*. The former includes transitions such as *Normal* to *Warning/Error* and *Warning* to *Error*, and the latter includes the other three transitions, i.e., *Error* to *Warning/Normal* and *Warning* to *Normal*. A transition may or may not occur when an event is generated, and the state machine maintains information about the event that caused the last state transition. More details about this event maintenance mechanism are provided in Sec. 4.3.5.

4.3.4 State Variables

The GCM defines the *state variable* to indicate the current state of the GCM state machine. One state variable is defined for each instance of the GCM state machine. A complete set of state variables represents the instantaneous status of the entire system.

The GCM defines two different types of state variables. The first type is the *actual* state variable that represents an instance of the GCM state machine. State variables of this type are simply called "states" and are marked as $s_X(Y)$ where X indicates the element that the state variable is associated with, and Y denotes arguments specifying the element X and

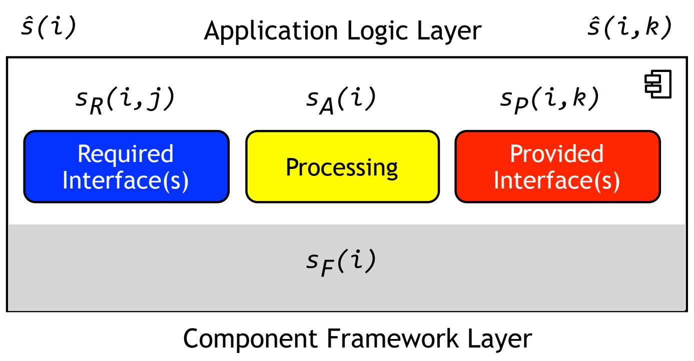


Figure 4.9: State variables of Generic Component Model: $s_R(i, j)$, $s_A(i)$, $s_F(i)$, and $s_P(i,k)$ are the *actual* state variables associated with instances of the GCM state machine, whereas $\hat{s}(i)$ and $\hat{s}(i,k)$ represent the *derived* state variables, for component i, required interface j, and provided interface k

timestamp. The second type is the *derived* state variable that is not associated with an actual instance of the GCM state machine, but is derived from other state variables. To indicate derived state variables, we use the hat symbol notation without a subscript: $\hat{s}(Y)$.

Fig. 4.9 illustrates a complete set of state variables that represent the status of a GCM component. The following sections describe each state variable.

4.3.4.1 Component State

GCM defines the *component state* that represents the current status of a component. Because the GCM has the two views (the *framework* and *application* views) and the overall view (the *system* view) (see Fig. 4.6), the complete description of component states requires three different state variables that correspond to each view. Given the i-th component of a system at time t,

$$\hat{s}(i;t) := \text{A component state in the } \textit{system} \text{ view} \tag{4.1a}$$

$$s_F(i;t) := \text{A component state in the } \textit{framework} \text{ view} \tag{4.1b}$$

$$s_A(i;t) := \text{A component state in the } \textit{application} \text{ view} \tag{4.1c}$$

Note that the hat notation is used for $\hat{s}(i;t)$ to indicate that it is a derived state variable. Each state variable follows the state definition in Table 4.2 and its value is one of *Normal*, *Warning*, or *Error*. For brevity, we henceforth use N, W, and E, respectively. A set of states is also defined to represent multiple component states in a vector form:

$\hat{\boldsymbol{S}}(t) :=$ A set of $s(i;t) = \{\hat{s}(1;t), \hat{s}(2;t), \cdots\}$

$\boldsymbol{S}_F(t) :=$ A set of $s_F(i;t) = \{s_F(1;t), s_F(2;t), \cdots\}$

$\boldsymbol{S}_A(t) :=$ A set of $s_A(i;t) = \{s_A(1;t), s_A(2;t), \cdots\}$

Note that $\hat{s}(i;t)$ has a correlation with $s_F(i;t)$ and $s_A(i;t)$, i.e., $\hat{s}(i;t)$ is a function of $s_F(i;t)$ and $s_A(i;t)$. We represent this relationship using the *state product operator*, $\otimes$:

$$\hat{s}(i;t) = s_F(i;t) \otimes s_A(i;t) \tag{4.2}$$

Given two states s_1 and s_2 as operands, Fig. 4.10a presents the definition of the state product operator. If either s_1 or s_2 is N, the operator takes the other state variable as output. If either is E, the output becomes E regardless of the other state variable. The output is W if both states are W. Fig. 4.10b shows the pattern of the state product operation. Conceptually, the operator behaves in the same way as the "max" operator in terms of severity, i.e., given two states, this operator chooses the one that is more "severe". Fig. 4.10c illustrates the state transformation. When a component queries another component's state, the state is "transformed" into either *N* or *E*. This is because the *Warning* state is an informational state and does not matter to the other components as long as the service is provided correctly. This helps to simplify not only the behavior of components that rely on other component's services, but also the GCM error propagation semantics, as described later in Sec. 4.3.7. The original, non-transformed states are also available.

$s_1 \backslash s_2$	N	W	E
N	N	W	E
W	W	W	E
E	E	E	E

(a) State product operation

$s_1 \backslash s_2$	N	W	E
N	N	W	E
W	W	W	E
E	E	E	E

(b) Pattern of the operation

$s_1 \backslash s_2$	N	W	E
N	N	N	E
W	N	N	E
E	E	E	E

(c) State transformation

Figure 4.10: Definition of state product operation

4.3.4.2 Interface State

The GCM also defines the *interface state* that represents the current status of an interface, such as availability, correctness, and timeliness. One state variable is assigned to each instance of interfaces, and two types of state variables are defined to handle provided interfaces and required interfaces separately. For the i-th component at time t,

$$s_R(i, j; t) := \text{A state of the } j\text{-th } \textit{required} \text{ interface in the } \textit{application} \text{ view} \tag{4.3a}$$

$$s_P(i, k; t) := \text{A state of the } k\text{-th } \textit{provided} \text{ interface in the } \textit{application} \text{ view} \tag{4.3b}$$

As in the component state, each state variable follows the state semantics described in Table 4.2 and its value is one of N, W, and E. A set of states is also defined to represent multiple interface states:

$$\boldsymbol{S}_R(i; t) := \text{A set of } s_R(i, j; t) = \{s_R(i, 1; t), s_R(i, 2; t), \cdots\} \tag{4.4a}$$

$$\boldsymbol{S}_P(i; t) := \text{A set of } s_P(i, k; t) = \{s_P(i, 1; t), s_P(i, 2; t), \cdots\} \tag{4.4b}$$

In contrast to the component state, the interface state is defined only in the *application* view. That is, all services are considered to be part of the application. Although some component-based frameworks may provide framework-level services such as dynamic component composition or component reconfiguration, GCM handles such framework-specific services as part of the application for generality purposes.

4.3.4.3 Service State

Another state variable in the GCM is the *service state*. The service state variable is a derived state variable, and indicates whether a provided interface can provide its service correctly considering its service dependencies. An individual service state variable is defined for each provided interface instance. For the i-th component at time t, the service state is defined as follows:

$\hat{s}(i,k;t)$:= A service state of the k-th *provided* interface in the *application* view

A set of service states is also defined, as for the other state variables:

$\hat{\boldsymbol{S}}(i;t)$:= A set of $\hat{s}(i,k;t) = \{\hat{s}(i,1;t), \hat{s}(i,2;t), \cdots\}$

The definition of a service state variable varies depending on its service dependencies. There can be cases where a service can be provided without relying on other resources, i.e., a service state variable that is independent from other state variables. For example, a service that provides sensor readings for other components without computation does not need any computation resource (e.g., CPU time). Such service states are defined as provided interface states:

$$\hat{s}(i,k;t) = s_P(i,k;t)$$

It is also possible that a provided interface requires other resources such as data from other components or computation time to execute application-specific logic. In such cases, the service state for the provided interface depends on other state variables. For example, a trajectory generator component provides a service that allows another component to specify the next goal position, and generates as output interpolated setpoint positions at a higher rate. This trajectory generator component connects to a PID control component that accepts setpoint positions as its control input. In this case, the trajectory generator component may fail to provide its service if (1) the connection between the trajectory generator component and the PID control component is disconnected (s_R becomes E), (2) the processing thread crashes (s_F becomes E), or (3) an exception is thrown when executing the application-specific logic and the trajectory component fails to generate outputs (s_A becomes E). This service dependency information is represented using the state product operator:

$$\hat{s}(i,k;t) = \left[s_R(i,j;t) \otimes s_F(i;t) \otimes s_A(i;t) \otimes\right] s_P(i,k;t) \tag{4.5}$$

where the first three states between [and] are *optional* terms that are determined by dependencies of the service. The order of states does not matter because the state product operation is commutative.

The service state is the key element that enables error propagation in the GCM. Sec. 4.3.7 provides further details on the GCM error propagation.

In summary, Table 4.3 presents a complete list of state variables in the GCM. This table essentially shows a "snapshot" of the run-time status of the component-based systems at time t, thereby enabling the description of the status of a system in a structured and comprehensive manner.

For convenience, we henceforth omit the parameter t from the list of state variable arguments, except for cases where this would lead to confusion.

Table 4.3: State variables of Generic Component Model (assumed at time t)

Id	Component Name	Component State			Interface State		Service State
		System View	Framework View	Application View	Required (j-th)	Provided (k-th)	
i	Name	$\hat{s}(i)$	$s_F(i)$	$s_A(i)$	$s_R(i,j)$ $\boldsymbol{S}_R(i)$	$s_P(i,k)$ $\boldsymbol{S}_P(i)$	$\hat{s}(i,k)$ $\hat{\boldsymbol{S}}(i)$
Entire System		$\hat{\boldsymbol{S}}$	$\boldsymbol{S}_F$	$\boldsymbol{S}_A$	-	-	-

4.3.5 Event

As described in Sec. 4.3.3.2, the GCM state machine is an event-driven finite-state machine where events can directly change the status of the system by initiating state transitions. In the GCM, the *event semantics* define what an event consists of, and how to define events.

The GCM event has five attributes: *name*, *severity*, *transition*, *timestamp*, and *description*. The *name* is a unique identifier of an event and no duplicate name within a system is allowed. The *severity* is the degree of criticality used for determining the relative priority of events. The *transition* defines a set of possible state transitions that may occur due to an event, and each transition is represented as a pair of the current state and the next state. The GCM state machine (Fig. 4.8) defines six possible transitions: $N \rightarrow W$, $N \rightarrow E$, $W \rightarrow E$ and $E \rightarrow W$, $E \rightarrow N$, $W \rightarrow N$. The *timestamp* is the time when an event is generated, and the *description* can include additional information about an event (e.g., error description in human readable form). The first three attributes – name, severity, and transitions – are

Table 4.4: Attributes of event in Generic Component Model

Id	Attribute	Definition	Determined at
i	Name	Unique identifier	Design-time
	Severity	Relative degree of criticality	
	Transitions	State transitions that may occur due to this event	
	Timestamp	Time when an event was detected or generated	Run-time
	Description	Extra information	

specified by the system designer as part of the system development process, whereas the other two – timestamp and description – are determined at run-time. Table 4.4 summarizes the five attributes of GCM events.

System designers define GCM events after performing a safety analysis that identifies potential hazards of the system. Events are associated with hazards in such a way that hazards can be prevented in advance by detecting associated events. The design-time event attributes – name, severity, transitions – are determined based on the results of the safety analysis and the system designer's knowledge and experience. The GCM event does not specify an algorithm or logic for event detection. System designers define event detection algorithms using the GCM filter, which is described in the next section (Sec. 4.3.6). Once GCM events are defined, all events are registered to the system so that the system will be able to generate and handle these user-specified events.

4.3.5.1 Onset and Completion Event

The GCM event semantics defines two different types of events: *onset* and *completion* events. The onset event refers to the *occurrence* of an event, and the completion event represents the *resolution* or *completion* of the event handling process for an associated onset event[iv]. The idea is to represent an event and related activities in terms of the two individual GCM events and handle them separately. This distinction enables the structured and systematic management of events.

By convention, the name of a completion event has a prefix "`/`". For example, a `sensor_error` event can be handled as a combination of two separate events: a `sensor_error` event – as the onset event – that is generated if any sensor error is detected, and a `/sensor_error` event – as the completion event – that is generated after the system has recovered from the sensor error. In terms of the Mechanism View of SDV (Fig. 3.1), the onset events are used as part of the monitoring, detection, and reaction mechanisms, whereas the completion events are associated with the recovery mechanism.

It is not always possible to handle an event with a combination of an onset and a completion event. Some events do not necessarily cause adverse effects on the system, and thus the completion event is unnecessary or undefined. In such cases, completion events can be undefined because these events do not typically initiate state transitions.

[iv]To avoid confusion, the term "event" is used to refer to events in general, whereas the "GCM event" is used to represent events defined by the GCM event semantics.

4.3.5.2 Outstanding Event

Each instance of the GCM state machine maintains information about an onset event that caused the last state transition, and this event is called the *outstanding event*. The outstanding event is undefined if the current state is N; otherwise, it is set as the latest onset event that caused a state transition to the current state. That is, given a time t and a state variable s that is associated with an instance of the GCM state machine sm, an outstanding event $e_{out}(sm;t)$ of the state machine is defined as follows:

$$e_{out}(sm;t) = \begin{cases} e_i & \text{if } s = W \text{ or } E \\ \textit{undefined (null)} & \text{if } s = N \end{cases}$$

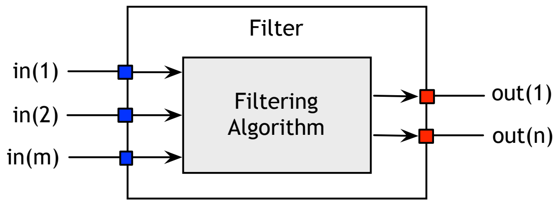
where e_i is the i-th GCM event registered to the system, which has caused the latest onset event that caused the state transition. Each GCM state machine maintains only one outstanding event at a time. The outstanding event changes if (1) a new GCM event of equal or higher severity occurs, or (2) the current outstanding event is resolved. This implies that newer GCM events of lower severity are not handled until the current outstanding event is resolved. Furthermore, the current outstanding event can change even if it is not resolved yet. The rationale is that handling the root cause of higher-priority events may also resolve the problems causing lower priority events.

4.3.6 Filter

The previous sections described the state and event semantics in GCM. The state semantics enables the generic representation of the system status, and the event semantics provides the event-driven mechanism that can explicitly capture and change states. The next question is how to generate events in a generic and consistent manner. The GCM events are generated by an element called a *filter*.

A filter is an abstract unit of computation that can represent any arbitrary algorithm and can generate events as a result of computation. As shown in the internal processing pipeline of the GCM in Fig. 4.4, the filter is the element between the input (from other components or from the environment) and the event. The filter is based on the filter mechanism for the fault detection and diagnosis of component-based robotic systems (Jung and Kazanzides, 2010).[203] To avoid confusion, we henceforth adopt the term GCM filter to distinguish it from the filter of the prior work, which we call the *original* filter. The original filter consists of three generic elements: (1) one (or more) *input(s)*, (2) a *filtering algorithm*, and (3) one (or more) *output(s)*. Essentially, the original filter takes in inputs, executes the filtering algorithm defined and implemented by the system designers, and generates outputs. This filtering mechanism requires a *history buffer*, which is a time-indexed circular buffer that contains a history of data. Fig. 4.11 depicts the original filter and history buffer.

The GCM filter is derived from the original filter with additional features to support the GCM event semantics. This extension includes (1) the specification of a state machine associated with the GCM filter and (2) support for the event semantics of the GCM. After the GCM filter is deployed, events generated by the filter are handled by a state machine associated with the filter. It is possible to associate multiple filters with the same state machine such that different events are generated using different filtering algorithms. As in



(a) Filter

T	in(1)	...	in(m)	out(1)	...	out(n)
⋮						
t(k)	0.50	...	0.4 1.3 4.2	0.3 2.4 0.5 3.2	...	2.3
t(k+1)	0.53	...	0.3 1.9 4.5	0.4 2.6 0.2 3.8	...	1.1
⋮						

(b) History buffer

Figure 4.11: Filter and history buffer (Jung and Kazanzides 2012[203]). A filter is comprised of input(s), a filtering algorithm, and output(s). A history buffer maintains the history of timestamped data.

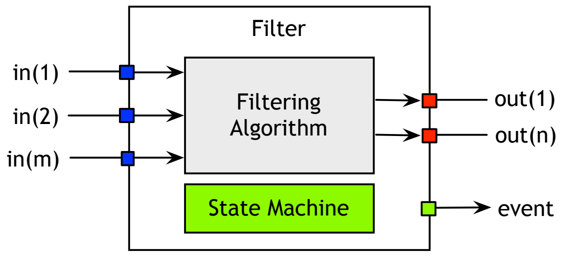


Figure 4.12: Filter of Generic Component Model. The GCM filter extends the original filter and filtering mechanism to support the event semantics of the GCM. This extension includes the specification of a state machine associated with the filter, which handles GCM events that this filter generates. Note that the state machine in the filter is not an instance of the GCM state machine – it is a specification of a state machine associated with the filter.

the original filter and filtering mechanism, GCM filters can also be cascaded into a filter pipeline by using output(s) of a GCM filter as input(s) of another filter. Fig. 4.12 illustrates the GCM filter where the extension is represented in green.

For the i-th component at time t, the p-th GCM filter associated with the component state is represented as follows:

$$f_X^p(i;t;\boldsymbol{in}(i,p;t))$$

where X is F or A for the framework or application view, respectively, and $\boldsymbol{in}(i,p;t)$ is a set of inputs to the p-th filter, i.e., $\{in(i,p,1;t), \cdots, in(i,p,m;t)\}$ (m: total number of inputs defined for the p-th filter). A collective form, $\boldsymbol{F}_F(i;t)$ and $\boldsymbol{F}_A(i;t)$, is also defined to represent a set of GCM filters.

For the j-th provided or required interface of the i-th component at time t, the q-th GCM filter associated with the interface state is defined as follows:

$$f_Y^q(i,j;t;\boldsymbol{in}(i,j,q;t))$$

where Y is R or P for required or provided interfaces, respectively, and $\boldsymbol{in}(i,j,q;t)$ is a set of inputs to the q-th filter, i.e., $\{in(i,j,q,1;t),\ \cdots,\ in(i,j,q,n;t)\}$ (n: total number of inputs defined for the q-th filter). $\boldsymbol{F}_R(i,j;t)$ and $\boldsymbol{F}_P(i,j;t)$ are also defined to represent a set of GCM filters.

The GCM filter semantics defines a particular order in executing filter algorithms. This order considers the two layers of GCM components and state variables shown in Fig. 4.9, and applies to the four groups of filters, $\boldsymbol{F}_F$, $\boldsymbol{F}_A$, $\boldsymbol{F}_R$, and $\boldsymbol{F}_P$. The order of filter execution is:

$$\boldsymbol{F}_F \longrightarrow \boldsymbol{F}_R \longrightarrow \boldsymbol{F}_A \longrightarrow \boldsymbol{F}_P$$

The rationale of this order is as follows:

1. $\boldsymbol{F}_F$: To ensure that the component-based framework works correctly (e.g., no thread crash, timeliness of thread scheduling).
2. $\boldsymbol{F}_R$: To make sure that the required services are available, i.e., the service is being provided by other components properly (e.g., no disconnection, service availability).
3. $\boldsymbol{F}_A$: To check if the application layer has any issue (e.g., no logic error, no application-level exception).

4. $\boldsymbol{F_P}$: To determine the service states of provided interfaces for other components.

Within a group, the order of filter execution is determined by the filter ID. For example, the order will be alphabetic or numeric if the type of the filter ID is defined as a string or number, respectively. One possible extension to the current design is to tag a filter with a property that specifies a priority, and to execute filters based on the priority. This would allow designers to have more fine-grained control over the execution order of the GCM filters at run-time.

4.3.7 Error Propagation

When an event occurs within a component, the error behavior of a system is determined by the error model of individual components and the interactions between components. Error propagation defines how abnormal events are propagated to other components, how states are transformed during the propagation process, and how errors in one component affect the status of the other components. This section describes a semantics for error propagation in the GCM.

The goal of error propagation is to inform the other components of the occurrence of error events so that the system can react to error events and update its run-time status accordingly in a timely manner. The starting point of the error propagation semantics in the GCM is the service state (Eq. 4.5) that comprehensively represents the service availability of a provided interface with consideration of resource dependencies.

The error propagation is *initiated* if a service state, $\hat{s}(i,k)$, changes to the *Error* state

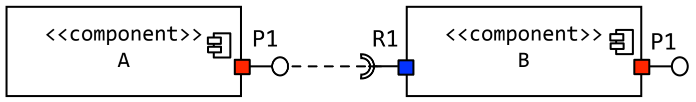


Figure 4.13: Minimal two-component system to illustrate error propagation in GCM

(N or $W \rightarrow E$) or to the *Normal* state (E or $W \rightarrow N$). Changes to the *Warning* state (N or $E \rightarrow W$) do not lead to error propagation because the service in the *Warning* state is still considered to be correct from the external component's perspective, although the service is possibly provided in a degraded mode. Semantically, this is equivalent to the state transformation, as shown in Fig. 4.10c, where W is considered as N from the outside. Once initiated, the error event is forwarded to a (set of) required interface(s) connected to the provided interface associated with the service state. Then, the state(s) of required interface(s) becomes *Error* because its required service is unavailable. In this way, the error events are propagated to other components through connections between components, and the run-time status of connected components are affected by error propagation.

To illustrate error propagation, we consider a minimal system that consists of two connected components, as depicted in Fig. 4.13, where `Component A` has one provided interface `P1` and `Component B` has one required interface `R1` and one provided interface `P1`. We are going to change some of the states of this system and illustrate how this change affects the rest of the system via error propagation, as presented in Figure 4.14.

Initially (t=0), all states are N. We first consider cases where the component state of `Component A` in the system view, $\hat{s}(A)$, changes to W or E.

At t=1, if $\hat{s}(A)$ becomes W, there are two possible cases depending on the definition

State Vars \ Time (t)		0	1	2	3	4	5	6
$\hat{s}(A) = s_F(A) \otimes s_A(A)$		N	$\underline{\text{W}}$	$\underline{\text{E}}$	N	N	N	N
$s_P(A, 1)$		N	N	N	$\underline{\text{W}}$	$\underline{\text{E}}$	N	N
$\hat{s}(A, 1) =$	$\hat{s}'(A, 1) = s_P(A, 1)$	N	N	N	W	E	N	N
	$\hat{s}''(A, 1) = \hat{s}(A) \otimes s_P(A, 1)$	N	W	E	W	E	N	N
$s_R(B, 1)$		N	N	E	N	E	N	N
$\hat{s}(B) = s_F(B) \otimes s_A(B)$		N	N	N	N	N	$\underline{\text{W}}$	$\underline{\text{E}}$
$s_P(B, 1)$		N	N	N	N	N	N	N
$\hat{s}(B, 1) =$	$\hat{s}'(B, 1) = s_P(B, 1)$	N	N	N	N	N	N	N
	$\hat{s}''(B, 1) = \hat{s}(B) \otimes s_P(B, 1)$	N	N	N	N	N	W	E
	$\hat{s}'''(B, 1) = s_R(\hat{B}, 1) \otimes \hat{s}(B) \otimes s_P(B, 1)$	N	N	E	N	E	N	E

Figure 4.14: Illustration of error propagation. State changes that originally initiated error propagation are marked with underline.

of the service state for `P1`, $\hat{s}(A, 1)$. If `P1` has no dependency on other resources for its service, denoted by $\hat{s}'(A, 1)$, the service state does not change. In contrast, the service state becomes W if `P1` relies on $\hat{s}(A)$, and this is denoted by $\hat{s}''(A, 1)$. In both cases, however, error propagation does not occur because $\hat{s}(A, 1)$ does not change to E.

At t=2, $\hat{s}(A)$ changes to E and $\hat{s}''(A, 1)$ becomes E, which then causes error propagation through the connection between `P1` of `Component A` and `R1` of `Component B`. As a result, the required interface state of `Component B`, $s_R(B, 1)$, changes to E. This state change may or may not lead to the change of the service state of the provided interface of `Component B`, $\hat{s}(B, 1)$. Similar to the case of `Component A`, there can be three different definitions of $\hat{s}(B, 1)$, and each is represented as $\hat{s}'(B, 1)$, $\hat{s}''(B, 1)$, and $\hat{s}'''(B, 1)$. At t=2, only $\hat{s}'''(B, 1)$ becomes E due to its service dependency on $s_R(B, 1)$.

At t=3, if $s_P(A,1)$ changes to W, $\hat{s}(A,1)$ becomes W regardless of its definition, but error propagation does not occur. However, error propagation occurs if $s_P(A,1)$ becomes E (t=4), and this causes $\hat{s}'''(B,1)$ to change to E. The same logic applies to cases at t=5 and 6.

The distinction between component states and interface states allows the GCM to handle component states and interface states separately, and this improves the expressiveness of the GCM. For example, when an internal processing thread crashes, $s_P(i,k)$ can still remain N if the service of the k-th provided interface consists of only "read" operations that do not require computation, i.e., the thread execution of component i. However, there can be cases where the component state and the interface state should be considered together, such as visualizing the system-wide health status. In such cases, the definition of the component state can be extended to consider the entire set of states of a component, such that all the states are consolidated into one state. This is called the *extended* component state and is defined as follows[v]:

$$\begin{aligned}\hat{s}_{ext}(i) &= \boldsymbol{S}_R(i) \otimes s_F(i) \otimes s_A(i) \otimes \boldsymbol{S}_P(i) \\ &= (s_R(i,1) \otimes \cdots \otimes s_R(i,j)) \otimes s_F(i) \otimes s_A(i) \otimes (s_P(i,1) \otimes \cdots \otimes s_P(i,k))\end{aligned} \tag{4.6}$$

4.3.8 State-dependent Operational Modes

The essential idea of the GCM is to explicitly define a set of minimal, but meaningful, states so that the run-time status of the system can be systematically captured and described, and

[v]Sec. 5.6.7.2 describes an example of how the extended component state is actually used for visualizing the overall system status.

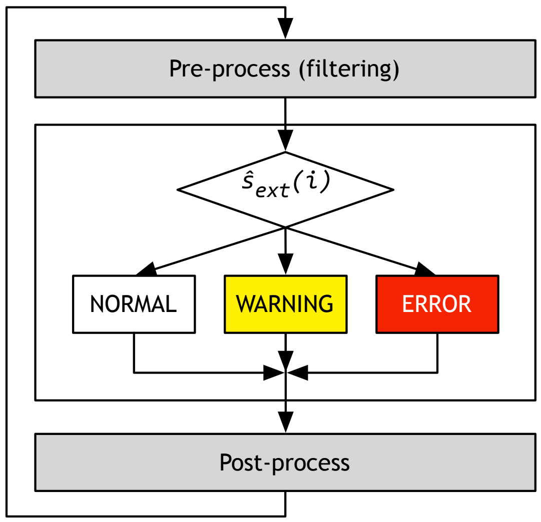


Figure 4.15: State-dependent operational modes in GCM. Three separate modules correspond to the three GCM states, and only one module is executed at a time depending on the current extended component state, $\hat{s}_{ext}(i)$.

possibly controlled for testing and verification of safety features. This section describes how this idea is applied to the code-level structure of a component in order to make the run-time behavior of GCM components be more explicit and more structured.

The idea is to decompose the application logic into *three separate processing modules* that correspond to the three states (N, W, E) and to execute only *one module at a time* depending on the current *extended* component state, $\hat{s}_{ext}(i)$ (Eq. 4.6). We call this scheme the *state-dependent operational modes*, which is depicted in Fig. 4.15. The *Normal* module corresponds to the *Normal* state and implements the default behavior or logic of the component. If $\hat{s}_{ext}(i)$ is W, the *Warning* module is executed to perform prognostic procedures. For example, a data sampling rate of the monitoring mechanism can be increased for more fine-grained monitoring. The goal is to prevent errors that can possibly happen. The *Error* module is executed if $\hat{s}_{ext}(i)$ is E. This module can be used to implement procedures for error recovery. In terms of the four essential elements of the Mechanism View (Fig. 3.1), the *Normal* and *Warning* modules can be primarily used for monitoring and detection, whereas the *Error* module is typically used for reaction and recovery.

This structure decouples code for the default behavior of a component from code for error handling and error recovery. This decoupling facilitates the component testing process. For example, each processing module can be easily tested by directly changing the component state. We can also generate GCM events and inject it into a state machine of interest. In this way, we can verify that (1) the state changes as expected, and (2) the behavior of the component is correct for different states. If the same code should be reused between different modules, system designers in practice can share code between the modules using techniques such as the Template Method pattern (Gamma *et al.* 1994[204]).

4.4 Discussion

The Generic Component Model (GCM) is a generic semantics that can define and describe the run-time status of component-based systems in a systematic and explicit manner. The use of minimal structural elements – components and interfaces – and four types of state variables leads to its *component-model independence*. The GCM also defines the error propagation semantics that consider service dependencies defined by component connections.

In medical robotics, there is no consensus on a standard programming model within the community, and error propagation and model-driven safety analysis techniques have not yet been investigated, as described in Sec. 4.2. The GCM is our proposal for an effective programming model to build large, complex, and safety-oriented medical robot systems with support for error propagation. Its inherently generic, extensible, and customizable design allows us to adapt the GCM to existing component models in a flexible manner, thereby achieving reusability and interoperability.

Unlike conventional safety analysis techniques of which the goal is to analyze the safety property of a system[vi], the primary goal of the GCM is to develop a semantics that can accommodate the design and implementation of safety features of component-based robot systems. Still, we note that there are many overlaps between the design of the GCM and the requirements of model-based safety analysis techniques. Grunske *et al.* (2005)[61] defined six requirements for model-based safety analysis techniques and used them to classify and compare different analysis methods. We summarize these six requirements, followed by the description of the GCM from each requirement's perspective:

1. **Appropriate component-level models**: *Each component must be annotated with an appropriate evaluation model.*

   » The GCM annotates a component with four types of state variables, each representing a state of an instance of the GCM state machines (Fig. 4.9), and describes the system status in terms of three generic states (N, W, and E) and outstanding events. Although the GCM does not specify any particular error behavior of a component,

[vi]According to Grunske *et al.* (2005):[61] *"The primary goal of safety analysis techniques is to identify all failures on the system level that cause hazardous situations and to demonstrate that their probabilities are sufficiently low."*

it provides generic mechanisms for error handling and error propagation between components, and allows system designers to implement application-specific behaviors that refer to the operational state of a component.

2. **Encapsulation and interfaces**: *The notation for the evaluation models should allow encapsulation and composition by interfaces similar to component-based design notations. The interfaces of the safety evaluation models should correspond as closely as possible to the interfaces of the component models.*

   » The GCM is a generic representation of various component models and it can be specialized for a particular component model. During this specialization process, a particular component model is "augmented" by the GCM and a set of GCM state machines is deployed to each component and interface of the system. Thus, the components and interfaces of the GCM are semantically identical to those of the system.

3. **Dependencies on external components**: *Safety analysis techniques must be able to express the dependencies of failures regarding provided services on failures regarding required services and on internal failures of the component.*

   » The GCM service state (Sec. 4.3.4.3) represents its service dependencies on both internal errors and external errors from other components. Based on the service state, errors in one component are propagated to other components along the component connection topology of the system.

4. **Integration of analysis results**: *The goal of safety analysis techniques is not only vi-*

sualizing the system for better understanding, but also running analysis algorithms on it. This requires that the algorithms are composable, i.e., the results from component analysis can be integrated to the results on the system level.

» Although the primary goal is not safety analysis, the GCM represents the system status using a hierarchical structure – system, components, interfaces – with the state product operator, and a combination of the structure and the operator makes the GCM composable. A partial set of system states can be combined with other sets, or can be consolidated even into a single state according to the state transformation semantics, as described in Sec. 4.3.4.1, as long as the two partial sets are in a proper hierarchy (e.g., two different interfaces in the same component, two components in the same system).

5. **Practicable granularity**: *The techniques applied should be on the one hand rich enough in details to express how different kinds of component behavior can influence system safety, but on the other hand coarse enough to allow affordable analysis on the system level.*

   » As described in Sec. 4.3.1, the rationale behind the design of the GCM is that it should be generic and flexible enough to be specialized for particular component models, while being expressive enough to capture and describe key information of the system in terms of safety. Furthermore, the state product operator allows the GCM to represent the system status at various levels of granularity, from a set of raw states in its entirety to just a single state after consecutively applying the state product operator.

6. **Tool support**: *The safety analysis technique should be supported by appropriate and ergonomic tools.*

   » The GCM does not specify any particular requirement for tool support because it is an abstract model that requires specialization for a particular component model. However, its specialization can have tool support, and the next chapter describes a suite of tools in more detail (refer to Sec. 5.6.7). Briefly, the idea is to provide tools that allow system designers to manipulate, i.e., read data from and write data to, the internal processing pipeline of the GCM (Fig. 4.4).

The current design of the GCM is based on two assumptions. The first assumption is that event handling and state transitions consume zero processing time, i.e., *no processing delay*, and they are considered as atomic operations, i.e., *no concurrency issues*. Thus, no *temporal* relationship is defined between events and state changes; only a *causal* relationship is considered. In practice, however, event handling that includes event generation, event delivery – possibly across networks – and event processing can take a significant amount of time. Also, multiple processes and/or threads are typically used together for large and complex systems. These assumptions can potentially lead to timing and concurrency issues. For example, the current GCM semantics does not specify the component behavior if the component state changes from N to E while executing one of the three state-dependent modules. In practice, such timing and concurrency issues are handled by the design and implementation of the GCM specialization process.

The second assumption is that event occurrences and state transitions in the GCM are modeled as *deterministic* events and transitions, rather than probabilistic ones. Probabilistic events are useful for modeling a wider range of events, such as stochastic events. For example, state transitions in State Event Fault Trees (SEFTs) can be modeled as one of three different types: deterministic, stochastic, and triggered transitions.[62] This leads to the GCM's inability to perform probabilistic safety analysis or stochastic modeling of system state changes. Although these issues may limit the expressiveness and applicability of the GCM, they provide opportunities for further improvement. For example, the addition of probabilistic semantics to the GCM would improve its expressiveness and enable probabilistic safety analysis.

4.5 Conclusions

We presented the Generic Component Model (GCM), a generic model that consists of the minimal structural elements (components and interfaces) and the state-based semantics that represent the operational status of component-based robot systems at run-time in an explicit, systematic, and structured manner.

The GCM is generic enough to be specialized for other component models, yet it is expressive enough to represent the complete description of the system status without relying on a particular component model. The essential elements of the state-based semantics of GCM include the state, event, and filter. The filters process data and generate the events that change the state. The use of the three abstract states (*Normal*, *Warning*, *Error*) enables the generic, consistent representation of the operational status of component-based systems. The events are defined based on the results from safety analysis of the system, and the filters provide mechanisms for monitoring and detection.

The error propagation model of the GCM systematically illustrates how the state of a component is affected due to errors from the other component, i.e., error propagation across the component boundary. This model could be useful for component-based software frameworks in robotics, where no existing component model explicitly defines and handles error propagation. In addition, the GCM introduces CBSE and error propagation semantics to the medical robotics domain, as an effective means to improve the design of safety features of medical robot systems.

One recent trend in the robotics community is to support data exchange between different component-based frameworks in order to increase component reuse[vii]. Towards this direction, the GCM can facilitate such cross-framework efforts by providing a standardized semantics and representation of the operational status of component-based systems.

4.6 Contributions

The thesis contributions described in this chapter are as follows:

1. Generic Component Model

– Proposal of generic model for component-based robot systems

[vii]There have been recent reports on such cross-framework data exchange: OROCOS with ROS (Smits and Bruyninckx, 2011[205]), OPRoS with ROS (Kang *et al.*, 2012[206]), and BRICS with ROS.[207]

We proposed a generic model that can be used to represent the operational status of component-based robot systems at run-time in an explicit, systematic, and structured manner. This model is an abstraction of various component models and is inherently extensible and customizable so that it can be specialized for a particular component model. To the best of our knowledge, no prior work in medical robotics presented systematic approaches to describing the run-time status of component-based robot systems.

2. State Machine

– Design of state-based semantics for Generic Component Model

We designed a state machine with three generic states, *Normal*, *Warning*, and *Error*. This state machine is based on the dependability formalism of the dependable computing domain. A set of state machines are deployed to the essential elements of component-based systems in a hierarchical manner, thereby representing the system status explicitly and systematically. This state-based semantics also enables error propagation across the component boundary.

3. Event Mechanism

– Design of event semantics for Generic Component Model

We designed an event mechanism with the concepts of onset and completion events, outstanding events, and rules for prioritizing events. This event mechanism is crucial to state transitions in the Generic Component Model. Separation of the source of state transitions from the state machine itself facilitates testing of the Generic Component Model and improves design flexibility.

4. Error Propagation

– Introduction of error propagation to component-based frameworks in robotics

We introduced the error propagation semantics to the field of component-based robot software frameworks. Although error propagation plays a crucial role on the operational status, there is no component model, in the robotics domain, that has the concept of error propagation across the component boundaries.

Chapter 5

Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)

5.1 Introduction

The previous chapter presented the Generic Component Model (GCM), a state-based semantics that can explicitly describe the operational status of component-based robot systems. The GCM is designed with component model independence using a minimal set of structural elements – components and interfaces – so that it can be specialized for particular component models. The four essential elements built around GCM include states, events, filters, and inputs, and these elements fundamentally define and control the system behavior.

CHAPTER 5. SAFETY ARCHITECTURE FOR ENGINEERING COMPUTER-ASSISTED SURGICAL SYSTEMS

Although the GCM is a generic model that enables us to *semantically* describe the operational status of the system, it does not define any code-level specifics, nor is it directly applicable to robot systems. Without an executable implementation, it is not feasible to dynamically verify that the design of the GCM is correct and to evaluate how effective it is at run-time. This recognition necessitates a software environment that provides a run-time environment for the GCM.

The design space of such a run-time environment is relatively large in that the GCM does not enforce any restriction on the design. However, the design should carefully consider its essential design elements, such as design requirements, design goals to achieve, and the design rationale of the GCM. Another important design element is the *programming model* that determines the inherent characteristics of the system. In robotics, component-based software engineering (CBSE) has been widely accepted as a de facto programming model. Typically, a component model is defined by a robot software framework. As in various areas in robotics, there exists a variety of robot software frameworks (e.g., Orocos, ROS, *cisst*, OpenRTM, OPRoS, CLARAty), and each framework defines its own component model that best fits their design requirements. The challenge here is to define a standard component model that reasonably fits for inherently different robot systems. For this reason, a run-time environment that only targets a particular component model is not likely applicable to other component models. It would be ideal if the same run-time environment is flexible and adjustable, so that it can support different component models in the same manner.

In this chapter, we describe a software framework called Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS) that provides a run-time environment for the GCM, and discuss its design rationale. SAFECASS aims to provide a run-time software environment based on the GCM semantics, while at the same time supporting the domain characteristics of medical robotics. This is achieved by (1) *decomposing* safety features into generic, reusable *mechanisms* and expressive, configurable *specifications*, and (2) combining mechanisms and specifications within a safety-oriented layered architecture, called the SAFECASS-based architecture. As a proof-of-concept, we use the *cisst* component-based framework[49] to demonstrate how SAFECASS achieves component model independence and how component model-*dependent* parts are isolated within SAFECASS and can be implemented by the framework.

The remainder of this chapter is structured as follows: Sec. 5.2 first provides a brief overview of prior work on safety with systematic or framework-based approaches. Sec. 5.3 describes the domain characteristics of medical robotics, which provides the foundations for the design requirements of SAFECASS, as described in Sec. 5.4. Then, Sec. 5.5 discusses our approaches to achieve the design requirements identified. Next, Sec. 5.6 describes the architecture and implementation of the GCM within SAFECASS. After presenting a brief overview of different architectural styles for robot control systems in Sec. 5.6.1, we propose a safety-oriented layered architecture for component-based systems. In the following sections (Secs. 5.6.2 - 5.6.6), we describe the design and implementation of each key element of the GCM in more detail, including the state machine, event, filter, and coordinator. Each of these sections has a separate subsection that describes *cisst*-specific implementations. Sec. 5.6.7 describes tool support within SAFECASS, and Sec. 5.7 presents SAFECASS-based safety features that we introduced to *cisst*. Sec. 5.8 illustrates how the system actually operates, using experimental data collected from a simple example system. Finally, we discuss the design and current implementation in Sec. 5.9, followed by our conclusions in Sec. 5.10.

5.2 Related Works

This section describes prior work on safety that presented systematic, architectural, or framework-based approaches in three areas: (1) in medical robotics, (2) in robotics (outside medical robotics), and (3) outside robotics.

In the medical robotics domain, prior work with systematic approaches to safety include HISIC (Fei *et al.*, 2001[9]) and the Design Framework (Sanchez *et al.*, 2014[32]). The hazard identification and safety insurance control (HISIC) is a systematic method for analyzing and controlling the design process of medical robot systems with safety. It consists of seven design principles that can be applied to software, hardware, and safety policies. HISIC was applied to the development of a 3D ultrasound image-guided robot system. Compared to SAFECASS, HISIC is a general guideline for designing and enhancing system safety, rather than a framework enabling a run-time environment, and HISIC is closer to the safety analysis techniques that we comprehensively described in Sec. 4.2.1. The Design Framework is one of the state-of-the-art safety design guidelines to develop surgical robot systems in accordance with the European directives for medical devices. Starting from the design cycle of a surgical robot, the framework presents design guidelines for electrical, electromechanical, software, and operational safety. The ARAKNES project[32] applied the Design Framework to the development of a surgical robot system for single port laparoscopy. The Design Framework also considers design elements that involve run-time issues, such as hard real-time operating system and control software architecture, and discusses specific implementations of safety features, such as watchdogs, emergency power-off, and kinematic singularities. In contrast to SAFECASS, the Design Framework does not place much emphasis on the reusability of safety features; mostly system- or application-level safety features are discussed. More importantly, the framework lacks the concept of CBSE, which may potentially lead to deployment issues as the scale and complexity of the system increase.

In robotics, the software architecture of robot control systems has been an active area of research.[178] Although there is no single architecture that fits for all cases, the layered, hierarchical control architecture has been increasingly popular because of its flexibility and multiple levels of abstraction. The robotics literature provides an overview of prior work in the robot control architecture (e.g., Kortenkamp and Simmons, 2009;[178] Nesnas *et al.*, 2006[208]). Starting from the sense-plan-act (SPA) paradigm in the late 1960s, the subsumption architecture was developed by Brooks in the mid 1980s, which has been called the most influential work in robotics history.[178] Most of these are driven by functional requirements, whereas non-functional properties, especially safety, are not of primary concern in these architectures. SAFECASS, on the other hand, is designed with a safety-oriented layered architecture where the underlying semantics are based on the Generic Component Model. Recently, Woodman *et al.* (2012[81]) presented the safety-driven control system architecture, called the safety protection system, with its implementation methodology in the area of autonomous personal robotics. The safety protection system verifies the safety constraints and enforces those constraints by controlling the actions of the robot system in order to prevent unsafe operations. Despite conceptual similarities between this approach and SAFECASS in terms of safety policy and architecture, it lacks the concept of components and CBSE. Without CBSE, it is questionable how systematically this architecture and approach would scale up for large and complex modern robot systems.

Outside the robotics domain, such as the real-time embedded systems domain and dependable computing domain, there exists a large body of work that presented architecture- or framework-based approaches to safety. A short list includes CHIMERA II (Stewart *et al.*, 1992[209]), GUARDS (generic upgradable architecture for real-time dependable systems; Powell *et al.*, 1999[210]), CADENA (component architecture development environment for avionics systems; Hatcliff *et al.*, 2003[211]), AUTOSAR (automotive open system architecture; Heinecke *et al.*, 2004[181]), and KARYON (kernel-based architecture for safety-critical control; Casimiro *et al.*, 2013[212]). Although target applications and system requirements vary, there is one common design principle among these: the framework provides generic and adaptable (and possibly verifiable) *base mechanisms* that can be instantiated into actual services or instances using *specifications or policies*. Likewise, this distinction between mechanisms and specifications is one of the fundamental design concept of SAFECASS, as discussed in

Sec. 5.5.2.

In the private sector, there exists a recent report that presented a survey of the architectural and design patterns (Hampton, 2012[213]). This report described and discussed various architectural patterns for safety together with development practices. Although those architecture and design patterns provide an overview of various existing architectural approaches, those architectures lack the domain characteristics of robot systems and the design of the Generic Component Model as well. For these reasons, those architectures are not directly applicable to SAFECASS.

5.3 Domain Characteristics

Medical robots are typically used in the operating room (OR) by highly trained surgeons. The surgeons have specialized skills for particular surgical applications, and use medical robot systems during the procedures. Currently, medical robots are categorized as medical devices and must be approved as medical devices by the regulatory agencies. The conceptual foundation of our approach to safety is based on these domain characteristics. In this section, we describe the domain characteristics of medical robot systems from four perspectives: *environment*, *control*, *application diversity*, and *regulatory requirements*.

5.3.1 Unstructured Environments

Traditional industrial manufacturing robots are placed in a predictable and relatively fixed environment where the workspace is designed considering robots, equipment, and manufacturing processes.[136] Compared to this predictable setting, the OR is an *unstructured* environment that exhibits variations in numerous elements (*variability*), such as patient anatomy, workspace configuration, surgical techniques, and technical skill of the OR team.[140] For example, anatomical structures vary in size and shape depending on the patient, making patient specific information (e.g., CT/MRI images) essential for surgery. Also, the medical personnel have different levels of technical skills and/or background.

In addition, the workspace around the end effector of medical robot systems is *unpredictable*. The robotic devices perform procedures on or inside the patient's body (e.g., internal organs or structures) which has constant movements due to heart beats, body motions, or respiration. Furthermore, the robot can actively make changes to the surrounding environment. During surgery, it is common that internal organs and structures significantly deform when parts of them are resected, punctured, or sutured, and then deviate from pre-planned models. This is a unique characteristic of medical robot systems.

It is often challenging to deal with the unpredictability and variability of the unstructured surrounding environments, especially within information intensive systems that use a variety of different types of data (e.g., high-frequency robot control data with small payload, low-frequency real-time stereo HD vision data with large payload). To deal with these unpredictability and variability issues, medical robot systems not only require diverse sensing techniques that can capture real-time information about the environment, but also rely on domain expertise such as the surgeon's knowledge, skills, and experiences, and patient specific information. For these reasons, medical robot systems are information intensive systems and are often called Computer-Integrated Surgery (CIS) systems.[3,87]

5.3.2 Human-in-the-loop Control

When performing surgical procedures, the surgeon, as the decision maker, desires to have full control over the surgical procedure for safety reasons. In traditional surgeries, highly trained surgeons with procedure specific knowledge made decisions based on their *direct* sensory perception on the patient's body. In robot surgeries, however, surgeons control the robot manipulator relying on the *indirect* sensory perception that the robot provides, or they supervise robot motions to make sure that the robot is in the safe state. To enable such features, one of the widely accepted control schemes in the medical robotics domain is *human-in-the-loop* control, where surgeons actively participate in surgical procedures as part of the control process. This concept is somewhat unique because the control scheme is neither fully automated, nor fully manual, while allowing system users to have full control over the surgery to exploit their expertise and to make decisions during the operation. To enable this control scheme, it is essential that the robot system provides the surgeon with sensory feedback, the current progress of procedures, and the overall status of the system. They also aid the surgeon or medical personnel's decisions during the operation.

5.3.3 Application Diversity

In medical robotics, a wide variety of medical robot systems have been developed in various application areas, such as neurosurgery, orthopaedic surgery, laparoscopic surgery, microsurgery, telesurgery, and so forth (refer to Table 2.2 for more details). Each application has highly procedure-specific requirements, which makes it less feasible to directly apply a robot system designed for one particular application to another. Because of the high development cost that involves approval by regulatory agencies, there have been approaches to increasing the reusability of the system. Examples of such approaches include the separation of the application-specific part from the common base system (e.g., Kazanzides *et al.*, 1992[5]), and the modular design of the robot system that allows different tools to be easily attached to, and detached from, the common robot manipulator (e.g., Guthart and Salisbury, 2000[102]).

5.3.4 Regulatory Requirements: Traceability and Testability

Medical robot systems are considered to be medical devices and thus to be used for clinical tests or released as commercial products, they must be approved by regulatory agencies such as the U.S. Food and Drug Administration (FDA). As part of the approval process, extensive documentation is required to show the design history and traceability between requirements, implementation, and testing. Without a proper tool support, this process needs to be performed manually. In case of large and complex systems, such "paper and pencil" methods are often impractical to analyze every possible case, and are likely to introduce more errors to the analysis process.[186] Ideally, the system should provide tool support for such testing and analysis processes in order to minimize manual labor and the possibility of errors.

Unlike many other safety-critical industries, the medical device regulatory agencies do not, in general, dictate specific design or manufacturing procedures that must be followed. This enables some flexibility for manufacturers to define a quality system that best fits their business and makes it possible to consider the *development and use of a custom framework* to assist with the design, implementation, and testing of medical robot systems.

5.4 Design Requirements

Considering the domain characteristics and the GCM semantics, the design of SAFECASS aims to address the following four requirements:

REQ. 1: Conformity to Generic Component Model

SAFECASS should be designed in accordance with the design rationale of the GCM.

The first requirement of SAFECASS is that a run-time environment that SAFECASS enables should conform with the design rationale of the GCM. Specifically, this requirement refers to (1) *component model independence*, i.e., SAFECASS should be reusable across different component models, and (2) *explicit and structured state management*, meaning that SAFECASS should provide facilities that enable the explicit and structured state management.

REQ. 2: Flexibility and Reusability

SAFECASS should provide *flexible* and *reusable* facilities for defining and deploying safety features to deal with the application diversity and the environment variability.

The second requirement is *flexibility* and *reusability*. As described in the previous section, medical robot systems are used for various applications, such as orthopaedics, neurosurgery, and laparoscopic surgery. Even within one application area, there are many factors that contribute to the variability of the environment, including patient-specific anatomical structures, different surgical techniques or approaches, and workspace configurations. Thus, SAFECASS must be designed to be flexible so that system designers can easily configure and implement safety features for various applications. It is even more desirable if the design of safety features can be systematically captured as a collection of use cases. Such a collection, as prior experience, may form the foundation for the "best practices", thereby facilitating reuse across different systems or different applications.

REQ. 3: Testability

SAFECASS should provide system designers *tools for testing* to verify the system behavior according to the GCM semantics.

The third requirement is *testability*. Testing plays a central role in verifying that the system behaves as specified and its behavior meets the design requirements. The effective design of testing facilities makes it easy for system designers to access any key data of the system. Ideally, a run-time environment that SAFECASS provides should improve testability of the system by enabling individual access to the key elements of the GCM, i.e., states, events, filters, and inputs.

REQ. 4: Traceability

Safety-related artifacts within SAFECASS should be *traceable*.

The last requirement is *traceability*. Traceability between different instances of the same system allows system designers to track changes over time and to identify the root-cause of problems. This feature helps to expedite the regulatory approval process if a previous version of the same system has already obtained clearance. It also facilitates the testing procedure by reducing the size of test suites by identifying incremental changes.

5.5 Approaches

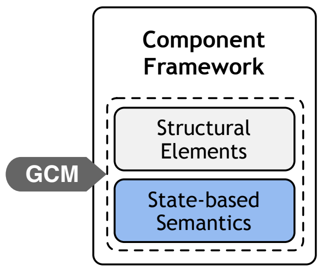
5.5.1 Framework Independence

The GCM is a generic model that consists of the structural elements and the state-based semantics without specifying any code-level implementation details. This enables design flexibility where we can implement a run-time environment for the GCM in different ways.

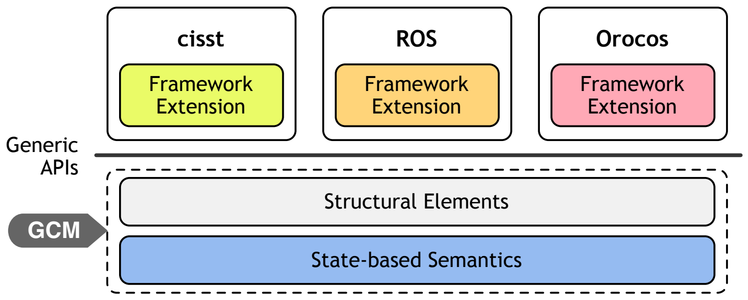
Fig. 5.1 illustrates two inherently different approaches. One option, as shown in Fig. 5.1a, is to choose a particular component framework and *directly modify* it to support the

GCM. Although this approach is straightforward and leads to tight, seamless integration of the GCM with the component framework chosen, this approach requires "deep" changes on the existing component framework and thus binds the GCM to a particular component framework. This makes it hard to reuse the run-time environment for the GCM for the other component frameworks. As a result, it is necessary to develop the entire stack of the run-time environment for the GCM whenever a new component framework is used.

Another option, as depicted in Fig. 5.1b, is to implement a run-time environment for the GCM in the *reusable, component framework-independent* fashion and to bridge the run-time environment with component frameworks via the *generic APIs*. Essentially, the idea is to separate framework-specific parts from a reusable, framework-independent implementation. This necessitates the *framework extension* within the framework, which provides framework-specific functionalities for the GCM and uses the generic APIs to access the information maintained by the GCM (e.g., states, events, filters). Compared to the aforementioned option, this approach requires more design efforts due to layering overhead for the two additional



(a) Framework-dependent design (not reusable)



(b) Reusable and framework-independent design

Figure 5.1: Two different designs of the run-time environment for the GCM

elements, i.e., the generic APIs and the framework extension. However, the generic APIs need to be designed only once as part of the framework-independent implementation, and only the framework extension – rather than the entire run-time environment for the GCM – needs to be implemented for a particular framework. More importantly, the advantages of this approach include:

- **Reusability**: Once the framework-independent run-time environment for the GCM is implemented, the same implementation can be reused for different component frameworks (if the framework extension is provided).
- **Maintainability**: The separation between the framework-independent part and the framework-specific part allows each part to be updated and maintained separately (assuming the generic APIs remain the same).
- **Use of multiple component frameworks**: It is possible to use multiple, different component frameworks within the same system. Despite the different component models, the information maintained by the GCM can be shared across the different component frameworks. This feature allows system designers to represent the operational status of the heterogeneous system in a consistent, systematic manner, and to define and coordinate the system behavior in terms of the GCM.

We chose the second approach mainly to benefit from these advantages. As will be described throughout this chapter, this design choice leads to profound effects on the overall design and implementation of a run-time environment for the GCM.

5.5.2 Safety Design Decomposition

One of the fundamental design principles of SAFECASS is reusability. As will be described later, we apply this principle to various aspects of SAFECASS, such as architecture and code-level designs and the design of safety features. The idea is to *decompose* a safety feature into a *reusable mechanism* and a *configurable specification*, thereby handling each part separately within SAFECASS. This approach, called *safety design decomposition*, has two goals: (1) to make the mechanisms generic, configurable, and adaptable, thereby reusable, and (2) to make the specifications expressive and comprehensive such that various application-specific safety requirements can be captured.

This separation facilitates reuse of safety mechanisms across different applications. One of the most widely used safety features in the domain is to power off the robot when excessive force is detected. According to the mechanism view described in Sec. 3.2.1, this safety feature can be decomposed into four mechanisms: *monitoring* of force feedback, *detection* of excessive force feedback above a pre-defined threshold F_{thresh}, powering off the robot as a *reaction*, and *recovery* from the excessive force feedback event. Although there are a couple of parameters to specify the design of this safety feature (e.g., which signal to monitor, which filter to use for detection), the key parameter is the pre-defined threshold, F_{thresh}, that determines the occurrence of the event. If a system already supports all the four mechanisms, different system behaviors can be easily defined by just changing a single parameter, F_{thresh}. Fig. 5.2 illustrates this concept.

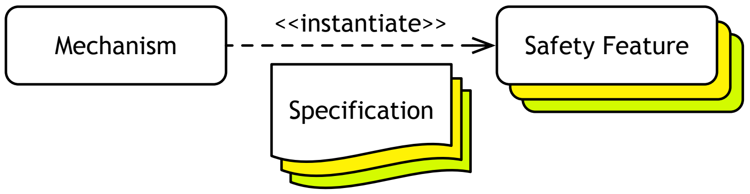


Figure 5.2: Decomposition of safety features into reusable mechanisms and configurable specifications. With the same mechanism, different safety features can be defined and deployed to the system by using different specifications.

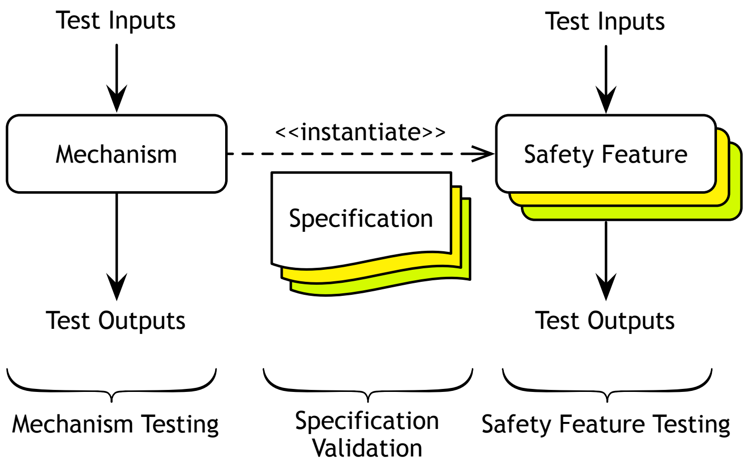


Figure 5.3: Testing of safety features. The decomposition of safety features allows us to test mechanisms and specifications separately, thereby enabling more extensive, modular, and verifiable testing processes.

Another advantage of this design is that it is possible to test safety features in terms of mechanisms and specifications, both *together* and *individually*. Typically, safety features are tested *as a whole* at the *system level* when the system is *online*. Given test inputs, the system is checked if the system behavior is correct in terms of outputs. In Fig. 5.3, this is represented as the *safety feature testing*. Once safety features are decomposed into mechanisms and specifications, we can test each part individually. For example, we can perform unit tests on the mechanism to verify that the mechanism works correctly. Because such testing can use any arbitrary specification, a set of automatically or systematically generated specifications can be used for rigorously testing corner cases and simulating specific error scenarios. The same technique can be applied to the specification as well, where we can validate consistency of specification. These concepts are illustrated as the *mechanism testing* and the *specification validation* in Fig. 5.3. It should be noted that (1) these decomposition-based tests can be performed *individually* at the *subsystem or module level*, even when the parts of the system that are irrelevant to the tests are *offline*, and (2) the conventional tests are still available as before.

5.6 Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)

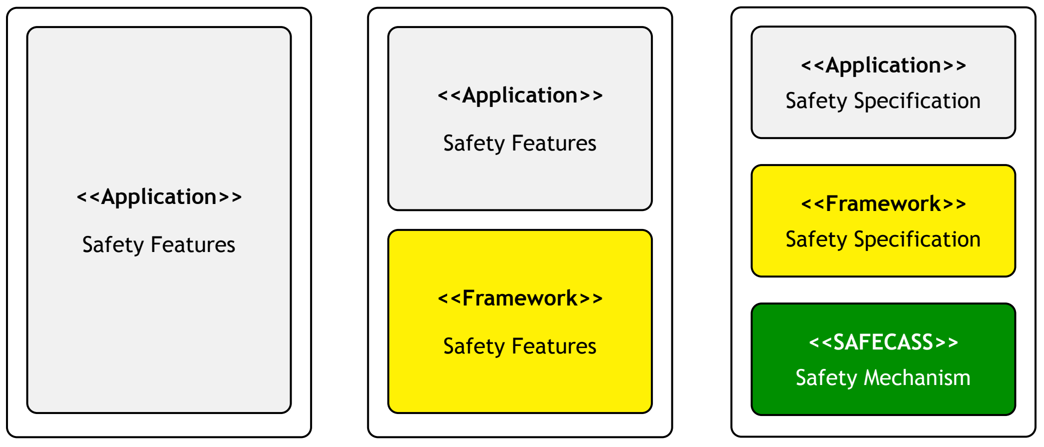
The Safety Architecture for Engineering Computer-Assisted Surgical Systems is developed as an open source C++ framework that consists of the core library, tools, and examples. SAFECASS aims to provide a run-time software environment for systematic safety research in component-based medical robotics. In this section, we present the architecture and design details of SAFECASS with discussions about our design decisions. This section is structured as follows: Sec. 5.6.1 first presents the architecture of SAFECASS with design rationale. Secs. 5.6.2 through 5.6.5 describe in detail how the essential elements of the GCM are implemented in SAFECASS. Sec. 5.6.6 introduces the key entity, called the coordinator, which maintains run-time data of the GCM (e.g., states, events) and coordinates activities within SAFECASS. Finally, Sec. 5.6.7 describes tool support for system introspection and visualization of run-time states.

5.6.1 Architecture

Since robot architectures and programming began in the late 1960s,[178] there has been a body of research on the architectures for robot control systems. Practical challenges imposed by the domain-specific functional requirements of robot systems have driven the evolution of the robot control architecture, whereas safety was not a primary focus of those architectures. As safety is receiving more attention in robotics, various approaches have been proposed, such as integration with safety standards and application of formal methods to the verification of safety properties of robot systems. Among those methods, our focus is on architectural approaches.

Fig. 5.4 shows different architectural styles with which safety features can be implemented. The first style is the *monolithic* architecture (Fig. 5.4a), where safety features are "embedded" in the system along with other functional elements of the system. This style leads to tight coupling between safety features and the rest of the system, thereby achieving highly application-specific safety features. At the same time, however, it is hard to reuse those safety features for other systems or applications due to the same reason.

The second style is the *framework-based* architecture (Fig. 5.4b) where a system consists of two layers: the *framework layer* and the *application layer*. These two layers are consistent with the GCM model of the architecture. This architecture represents typical component-based systems where the framework layer provides a component-based envi-



(a) Monolithic **(b)** Framework-based **(c)** SAFECASS-based

Figure 5.4: Evolution of the architectures of robot systems with safety features

ronment on which the application layer is built, as well as framework-level services that the application layer uses. Each layer tends to maintain its own safety features because of the inherently distinctive characteristics of each layer, as discussed in Sec. 4.3.2. From the safety perspective, this structure is similar to the monolithic architecture in that safety features are still embedded in the system.

The third style is what we propose, called the *Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)-based* architecture. As depicted in Fig. 5.4c, this architecture extends the framework-based architecture by (1) applying the concept of safety design decomposition to each layer, and (2) introducing a new layer called the *SAFECASS* layer. In this architecture, safety features of each layer are decomposed into mechanisms and specifications. The mechanisms are moved to the SAFECASS layer, whereas the specifications remain at each layer as before. In this way, this architecture realizes separation between mechanisms and specifications. This SAFECASS-based architecture is conceptually inspired by the *safety kernel* (Wika, 1995[214]) and *safety executive* (Leveson *et al.*, 1983[215,216]) approaches. Like the security kernel[217] in the security computing domain, the safety kernel coordinates various monitoring mechanisms between devices (hardware) and application software, and enables centralization and encapsulation of safety mechanisms with the advantages of reusability and possible formal verification of the operations of the executive.[15]

Fig. 5.5 shows a more detailed view of the architecture of SAFECASS. The following sections describe each layer in more detail.

5.6.1.1 SAFECASS Layer

The SAFECASS layer is mainly comprised of three parts: the *core library*, the *middleware*, and the *back-end tools*.

The core library is a C++ library that provides a code-level implementation of the essential elements of the GCM, and services and utilities that facilitate use of the GCM. The essential elements of the GCM are implemented as the *base mechanisms*, and SAFECASS provides four base mechanisms: state machines, events, filters, and coordinator. In Fig. 5.5, each mechanism is represented as the *base state machine*, the *base event*, the *base filter*, and the *base coordinator*. Secs. 5.6.2 through 5.6.6 describe the detailed design of these. One important decision on the design of the base mechanisms is that they are implemented as *objects*, rather than *components*. Defining a component requires a component model, which necessitates the introduction of a particular thread execution model

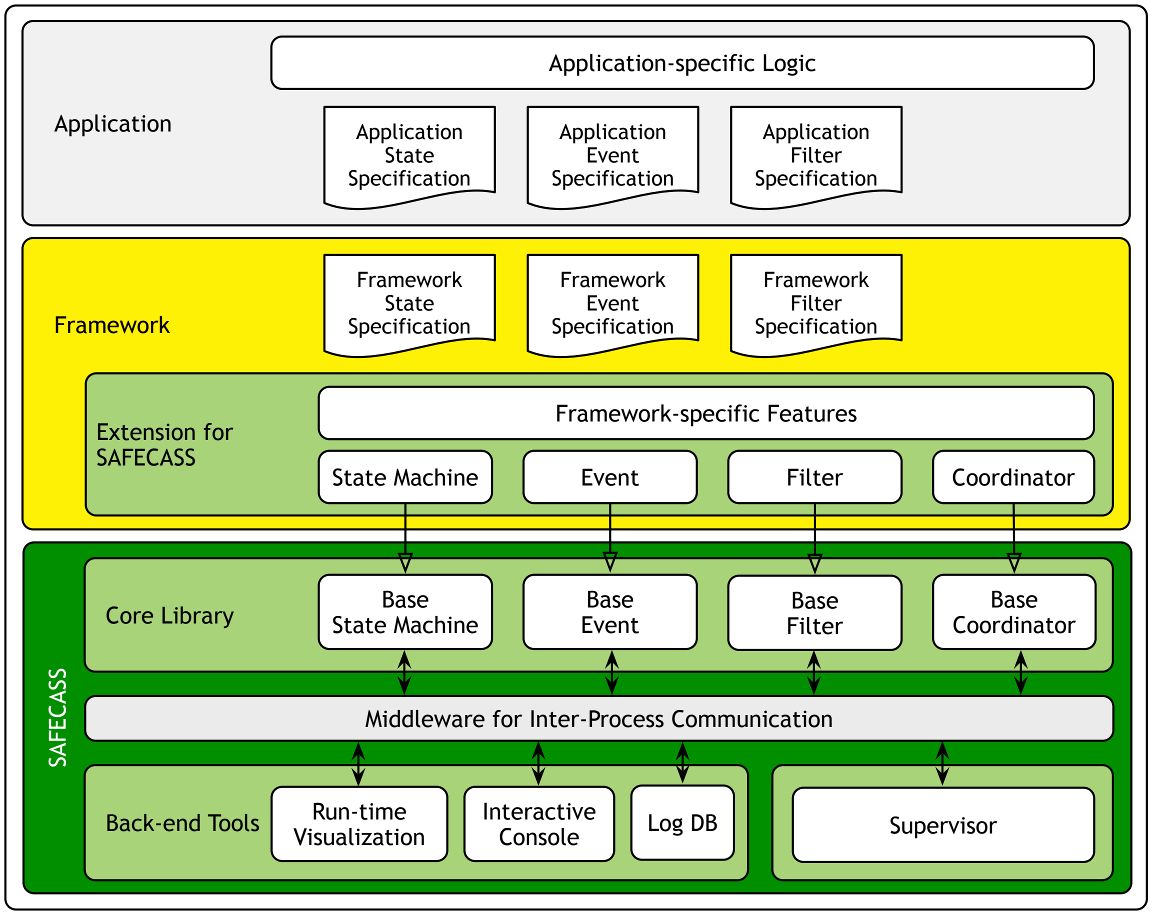

Application-specific Logic
Application
Application State Specification
Application Event Specification
Application Filter Specification
Framework
Framework State Specification
Framework Event Specification
Framework Filter Specification
Extension for SAFECASS
Framework-specific Features
State Machine
Event
Filter
Coordinator
SAFECASS
Core Library
Base State Machine
Base Event
Base Filter
Base Coordinator
Middleware for Inter-Process Communication
Back-end Tools
Run-time Visualization
Interactive Console
Log DB
Supervisor

Figure 5.5: Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)

to SAFECASS. This is not a desirable design because it unnecessarily complicates not only the design of SAFECASS, but also the integration process between SAFECASS and the framework layer. The rationale is that SAFECASS should provide only generic and fundamental functionality without enforcing any additional threading model on the system. In this way, the base mechanisms become inherently customizable and extensible to the needs of safety features of the upper layers, i.e., the framework and the application layers.

The middleware represents a standalone network infrastructure that enables inter-process communication (IPC) within SAFECASS. It provides a dedicated communication channel for SAFECASS to exchange messages. Supporting IPC is essential because modern robot systems are typically comprised of multiple processes in order to achieve challenging functional requirements, such as real-time robot control (typically 1 kHz) and run-time vision stream processing (up to around half a gigabyte video stream per second at 30+ frames per second).[49] The middleware is a standalone facility that does not rely on data exchange services that the framework layer provides. This independent communication infrastructure within SAFECASS improves data communication redundancy of the overall system. Messages are serialized and exchanged in the JavaScript Object Notation (JSON) format, which is a lightweight, text-based, language-independent data interchange format.[10] Currently, SAFECASS uses IceStorm, a publish-subscribe event distribution service from ZeroC.[218]

The back-end tools are comprised of a set of additional tools and services, such as run-time visualization, interactive console (for system introspection, fault injection, and event generation), and database support. Sec. 5.6.7 describes more details of these tools and services.

The system designer can optionally deploy the *Supervisor* to the system to form a hierarchy in the system information management. The Supervisor exchanges messages with every process of the system to access key information in each process. The Supervisor can also be used to impose particular safety policies or strategies on the system. Currently, the Supervisor is implemented as a lightweight, standalone process and thus can be dynamically launched or terminated at any time.

5.6.1.2 Framework Layer

The framework layer provides a component-based environment for the system; typically, this layer is provided by an existing component-based framework such as *cisst*, ROS, or Orocos. This layer is augmented by the *extension for Safety Architecture for Engineering Computer-Assisted Surgical Systems* that provides (1) a set of framework-specific mechanisms derived from the base mechanisms, and (2) the framework-specific features required by SAFECASS. In Fig. 5.5, the framework-specific mechanisms are shown as the *State Machine*, *Event*, *Filter*, and *Coordinator*. Depending on a framework, the base mechanisms can be directly used without specialization. The *framework-specific features* represents a set of “glue” code that provide framework-specific services required by SAFECASS, such as SAFECASS bootstrapping code and access to internal data of the framework. In addition, the framework layer defines the *specifications* of *framework-specific* safety features. SAFECASS uses these software artifacts – usually configuration files – to deploy and configure framework-specific safety features.

Currently, we use the *cisst* package[49] as a component-based framework, and implemented the *cisst* extension for SAFECASS. A brief overview of the *cisst* package is provided in Appendix A.

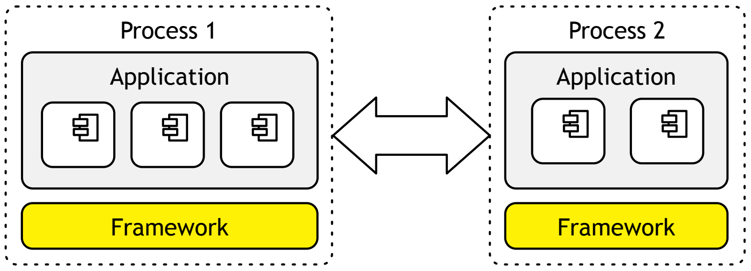
5.6.1.3 Application Layer

The application layer is a thin and lightweight layer in terms of the safety-related mechanism. This is a result of the SAFECASS-based architecture where the two underlying layers – the framework and SAFECASS layers – provide all the necessary mechanisms. Typically, this layer contains components that implement application-specific logic and the functional behavior of the system. Because of the large design space of these components, a particular structure cannot be enforced on the components of this layer. Instead, the framework allows system designers to facilitate the design of application-specific safety features. For example, the framework provides a set of APIs that enable the application components to access the information managed by the GCM (e.g., states, outstanding events). The framework also supports the state-dependent operational modes of the GCM, where the application logic can be decomposed into three separate processing modules, each corresponding to the three GCM states (N, W, E), as described in Sec. 4.3.8.

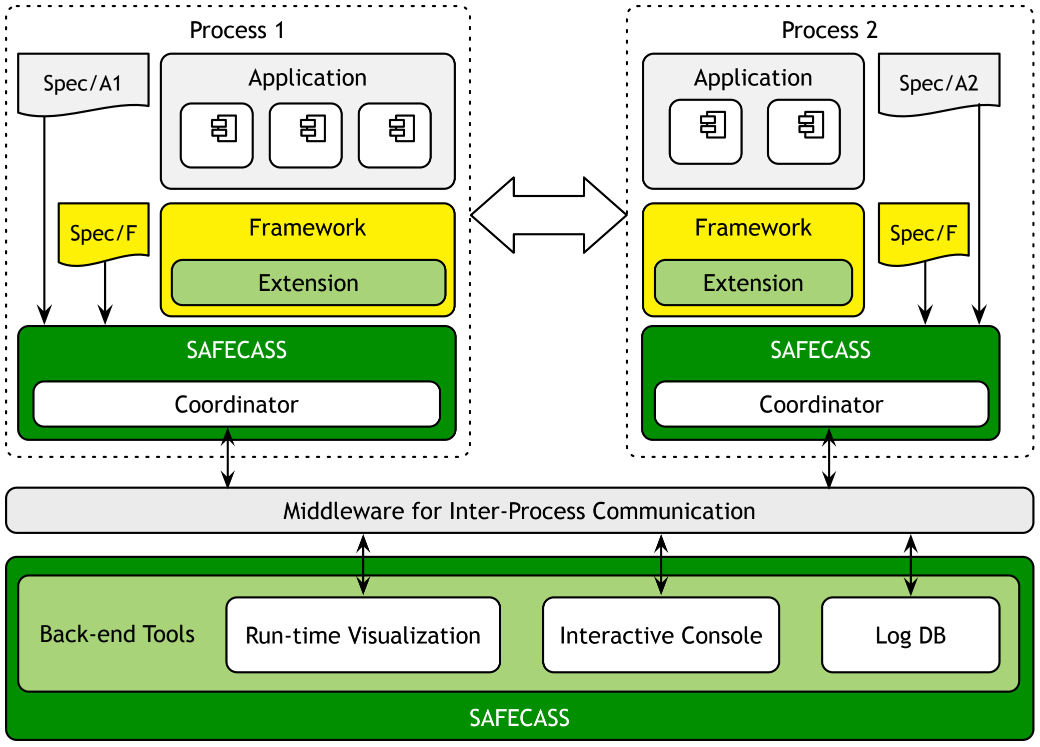
5.6.1.4 Deployment Concept

One of the main design goals of SAFECASS is to provide system designers with a *run-time* environment where safety features can be easily instantiated and deployed to the system with minimal code-level changes. This is achieved by the SAFECASS-based architecture and the decomposition of safety features. Fig. 5.6a illustrates a typical component-based distributed system built with the framework architecture. If we apply SAFECASS to this system, the system is *augmented* with SAFECASS, as in Fig. 5.6b, changing its architecture to the SAFECASS-based architecture.

Although the introduction of SAFECASS appears to bring significant changes to the original system, the part of the system that actually changes is limited to the extension for SAFECASS and the specifications of safety features of each layer. If a component-based framework already supports SAFECASS, i.e., the extension for SAFECASS is already available, the only "visible" changes to the original system include (1) adding code snippets for bootstrapping SAFECASS, and (2) defining safety specifications for the framework and application layers. This concept is illustrated in Fig. 5.7, where these changes are represented as the "extension for SAFECASS", "Spec/F", and "Spec/A1" and "Spec/A2".



(**a**) Original system



(**b**) System with SAFECASS

Figure 5.6: Deployment view of SAFECASS

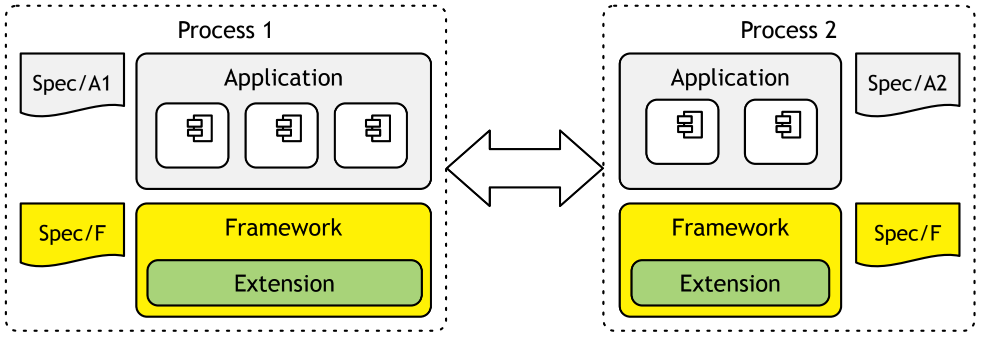


Figure 5.7: Concept of the deployment of SAFECASS: The deployment of SAFECASS to an existing system requires minimal changes to the original system. The only changes required include code snippets for bootstrapping SAFECASS (as part of the extension for SAFECASS) and safety specifications that are used to instantiate safety features.

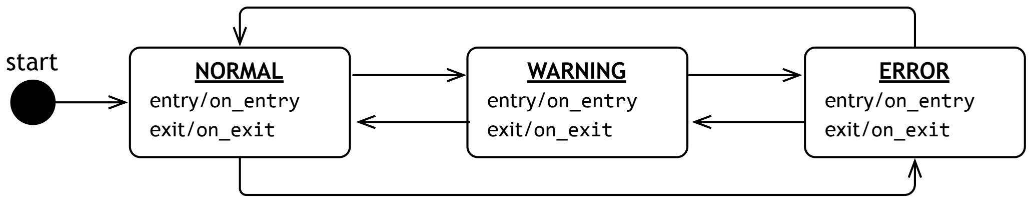
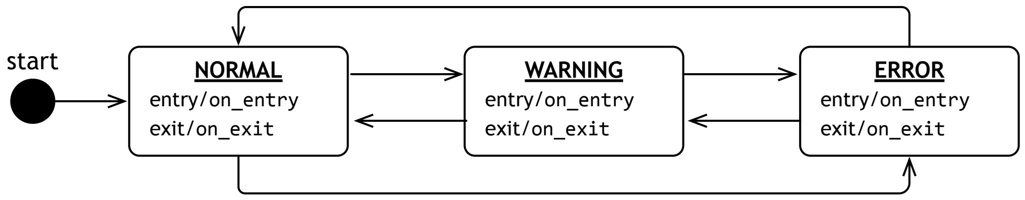


Figure 5.8: UML state diagram of the SAFECASS state machine. All possible state changes and state transitions are explicitly modeled and can be handled.

5.6.2 State Machine

The *base state machine* implements the GCM state semantics that defines the three GCM states – *Normal*, *Warning*, *Error* – and the six transitions between the states, as described in Sec. 4.3.3. According to the design principle of the GCM state machine, an implementation of the GCM state machine should enable the explicit, generic representation of the run-time status of the system. It should also allow the framework to handle any state entry/exit action or state transition in a structured manner.

Fig. 5.8 shows the UML state diagram of the SAFECASS state machine. Each state



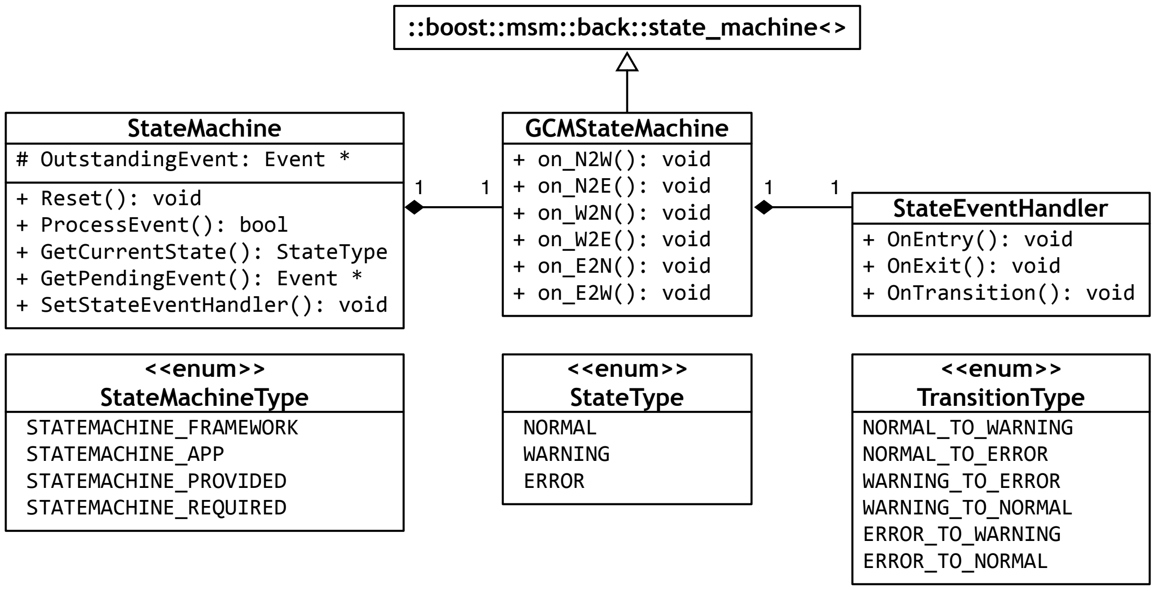


Figure 5.9: UML class diagram of the SAFECASS state machine class (`SC::StateMachine`)

defines *entry* and *exit* actions (`on_entry()` and `on_exit()`), where a system designer can specify any activity that is executed upon entering and exiting a state.

The SAFECASS state machine is implemented as the `SC::StateMachine` class of which detailed design is depicted in Fig. 5.9. We omit the namespace prefix "`SC::`" from the names of SAFECASS classes unless the omission leads to confusion. The three enumerations – `StateMachineType`, `StateType`, `TransitionType` – define the four types of state variables, the three GCM states, and six state transitions, respectively. The `StateMachine` class maintains `OutstandingEvent` that represents the outstanding event (Sec. 4.3.5.2). When a

GCM event occurs, `StateMachine::ProcessEvent()` handles the event and makes state transitions, if necessary, based on the event specification (described in the next section).

The SAFECASS state machine class maintains an instance of the `GCMStateMachine` class that provides an underlying state machine mechanism, which is implemented using the Boost Meta State Machine (MSM) library.[219] The MSM library enables a quick and easy implementation of state machines with high run-time performance using the template meta programming technique.[220] This run-time performance gain comes in exchange for the increased overhead at compile-time, which is acceptable in SAFECASS.

The design of the SAFECASS state machine allows the framework to easily override the default handlers for state entry/exit actions and state transitions. This is enabled by the separation of the event handler (i.e., the `StateEventHandler` class) from the state machine (i.e., the `GCMStateMachine` class). Upon a state transition, `GCMStateMachine` calls a state transition handler that corresponds to the transition, which is one of the following: `on_N2W()`, `on_N2E()`, `on_W2N()`, `on_W2E()`, `on_E2N()`, and `on_E2W()`. Then, a state transition handler internally calls `OnTransition()` of `StateEventHandler` with a transition type as an argument, which actually handles state transition events. State entry/exit actions are handled in the similar manner by `OnEntry()` and `OnExit()` of `StateEventHandler`.

5.6.2.1 *cisst* Implementation

cisst uses the default implementation of the base state machine of SAFECASS without specialization. Although it is possible to completely catch and handle state change and state transition events, *cisst* currently does not employ this feature.

5.6.3 Service State

As described in Sec. 4.3.4.3, the service state represents whether a provided interface can provide its service correctly, considering its service dependencies, i.e., on which state(s) the provided interface state depends. The service state in its complete form is defined in Eq. 4.5, which encodes the service dependency information into a single state. In the GCM, the service dependency is the key element that determines the impact of errors and error propagation.

Most component-based software frameworks in robotics do not explicitly capture the service dependency information at the framework level. Thus, SAFECASS relies on the system designer to provide this information in the JSON format as part of the safety specification. Additional details of this format are described in the following section.

5.6.3.1 Configuration File

Code 5.1 shows an example of a JSON specification that defines the service state dependency. The system designer defines the service state dependency for each component and the specification is loaded into SAFECASS as part of the system initialization process. The name of a component is specified with the key `component`. The key `service` defines an array of service dependency information. Each element of the array specifies the name of a provided interface and identifies the state variables on which it depends. To completely

Code 5.1: Example of JSON specification of the service state dependency

```
{
  "component" : "aComponentName",
  "service" : [
      {   // provided interface name
          "name" : "provided_interface_1",
          // dependency information
          "dependency" : {
              // s_R: depends on required interface(s)
              // s_A: depends on component state in the application view
              // s_F: depends on component state in the framework view
              "s_R" : [ "required_interface_1", "required_interface_2" ],
              "s_A" : true,
              "s_F" : true
          }
      },
      {   "name" : "provided_interface_2",
          "dependency" : {
              "s_R" : [ "required_interface_2" ],
              "s_A" : true,
              "s_F" : false
          }
      }
  ]
}
```

define service states of a component, a set of array elements that correspond to each provided interface have to be defined.

5.6.4 Event

The SAFECASS event implements the GCM events. As summarized in Table 4.4, a GCM event has five attributes: name, severity, transitions, timestamp, and description. Fig. 5.10 shows the design of the SAFECASS event class, `SC::Event`, that has the five attributes.

The `Name` is a unique string name of the event and is used as an identifier of the event. For readability, SAFECASS uses a prefix “`EVT_`” and “`/EVT_`” for the onset and completion events.

The `Severity` determines the relative priority of the event and ranges from 1 to 255,

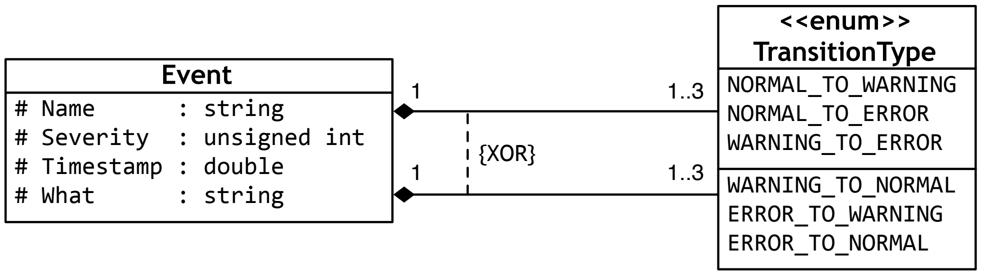


Figure 5.10: UML class diagram of the SAFECASS event class (`SC::Event`). The `XOR` represents the concept of the onset and completion events. That is, the `Transitions` attribute is allowed to contain only either one (or all) of the first three transitions or one (or all) of the others.

where the maximum severity of 255 is an arbitrarily chosen value. The range is divided into three groups: 1-200 for SAFECASS events in the application layer, 201-250 for SAFECASS events in the framework layer, and 251-255 for the broadcast events. The SAFECASS events for the framework layer have higher severity values than those for the application layer because the correct execution of the application layer relies on the correct services that the underlying framework provides. For example, a component cannot perform any computation or data exchange with other components if its processing thread crashes. The broadcast events can be used to define critical events that need to be simultaneously broadcast to all the state machines in the system. SAFECASS uses a prefix "`EVT(B)_`" and "`/EVT(B)_`" for the names of onset and completion broadcast events.

The `Transitions` defines possible state transitions associated with a SAFECASS event. When a SAFECASS event occurs, the SAFECASS state machine determines the next state based on the current state, the severity and possible transitions of the event. The six state transitions of the GCM state machine are represented as "`N2W`", "`N2E`", "`W2N`", "`W2E`", "`E2N`", and "E2W". In accordance with the GCM's distinction between the onset and completion events, transitions of the onset events can include one or all of N2W, N2E, and W2E, whereas those of the completion events can contain one or all of the others, i.e., W2N, E2N, and E2W.

The Timestamp represents a timestamp when a SAFECASS event was generated, and the What contains a detailed description of the event in human readable format. These two fields are determined at run-time.

5.6.4.1 Configuration File

The three design-time attributes – name, severity, transitions – are defined by the system designer in the JSON format. Code 5.2 shows an example of this event specification. SAFECASS reads the file on start up and populates the internal data structure accordingly based on the event specification. The other two run-time attributes – timestamp and description – are not specified in the configuration; they will be set later at run-time.

5.6.4.2 *cisst* Implementation

SAFECASS provides a complete implementation of the GCM events that do not require any framework-specific feature, and thus *cisst* directly uses the base event mechanism without any modification or specialization.

Code 5.2: Example of JSON specification of GCM events

```
{
  "component": "aComponentName"
  "event": [
    // Sensor warning
    { "name"            : "EVT_SENSOR_WARNING",
      "severity"        : 10,
      "state_transition": [ "N2W" ]
    },
    { "name"            : "/EVT_SENSOR_WARNING",
      "severity"        : 10,
      "state_transition": [ "W2N" ]
    },
    // Sensor error
    { "name"            : "EVT_SENSOR_ERROR",
      "severity"        : 15,
      "state_transition": [ "N2E", "W2E" ]
    },
    { "name"            : "/EVT_SENSOR_ERROR",
      "severity"        : 15,
      "state_transition": [ "E2N", "W2N" ]
    }
  ]
}
```

5.6.5 Filter

The GCM filter is an abstract unit of computation with the filtering algorithm where the system designer can implement arbitrary application-specific logic (Sec. 4.3.6). The two key requirements on the implementation of the GCM filter are *flexibility* and *extendibility* so that any custom algorithm can be deployed to the system in a consistent and systematic manner. SAFECASS provides an implementation of the GCM filters via the base filter.

5.6.5.1 Structural Elements

The GCM filter is comprised of inputs, filtering algorithm, outputs, and the additional elements to support the GCM event semantics. This section describes in detail how each of

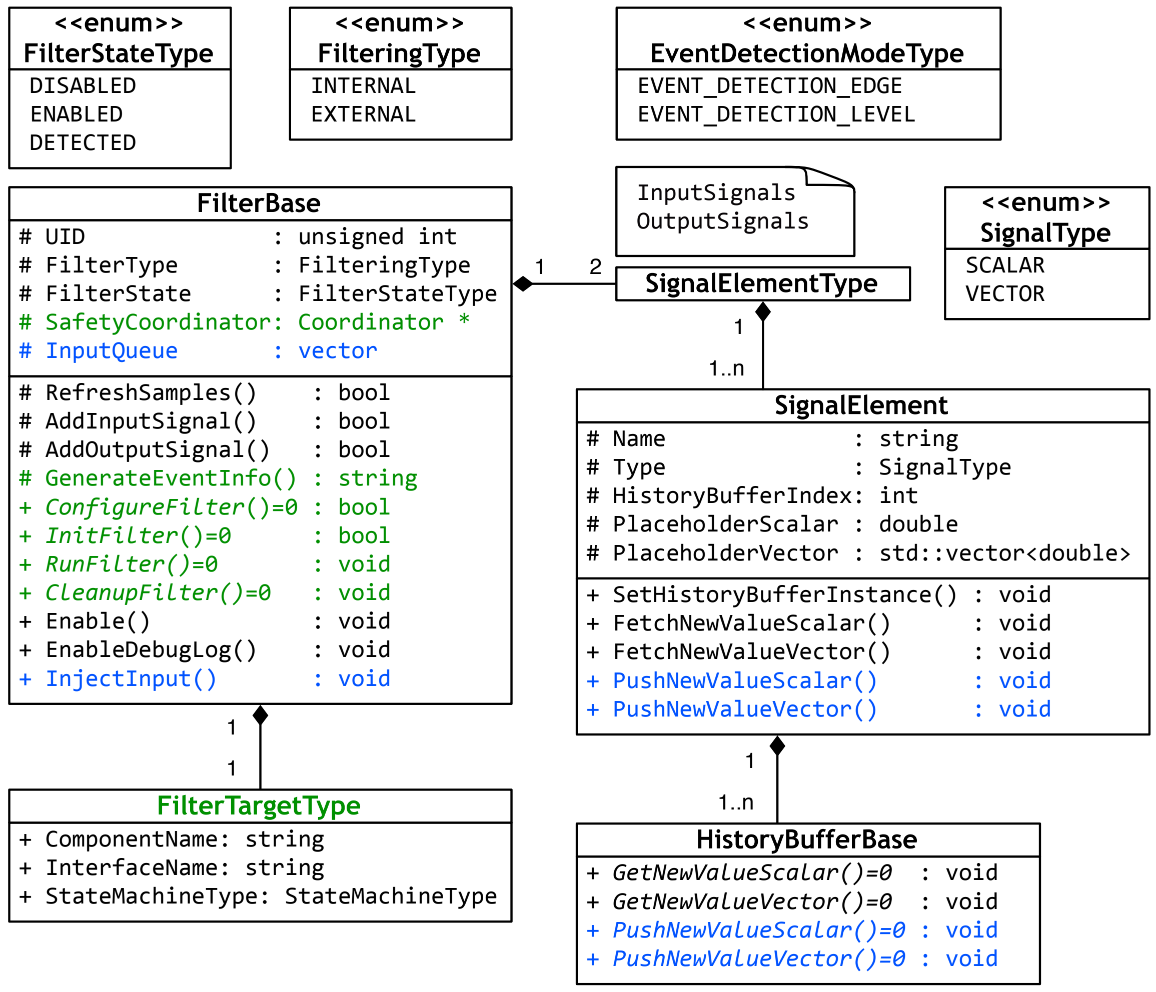

<<enum>>
FilterStateType
DISABLED
ENABLED
DETECTED
<<enum>>
FilteringType
INTERNAL
EXTERNAL
<<enum>>
EventDetectionModeType
EVENT_DETECTION_EDGE
EVENT_DETECTION_LEVEL
FilterBase
# UID : unsigned int
# FilterType : FilteringType
# FilterState : FilterStateType
# SafetyCoordinator: Coordinator *
# InputQueue : vector
# RefreshSamples() : bool
# AddInputSignal() : bool
# AddOutputSignal() : bool
# GenerateEventInfo() : string
+ ConfigureFilter()=0 : bool
+ InitFilter()=0 : bool
+ RunFilter()=0 : void
+ CleanupFilter()=0 : void
+ Enable() : void
+ EnableDebugLog() : void
+ InjectInput() : void
InputSignals
OutputSignals
1
2
SignalElementType
<<enum>>
SignalType
SCALAR
VECTOR
1
1..n
SignalElement
# Name : string
# Type : SignalType
# HistoryBufferIndex: int
# PlaceholderScalar : double
# PlaceholderVector : std::vector<double>
+ SetHistoryBufferInstance() : void
+ FetchNewValueScalar() : void
+ FetchNewValueVector() : void
+ PushNewValueScalar() : void
+ PushNewValueVector() : void
1
1
FilterTargetType
+ ComponentName: string
+ InterfaceName: string
+ StateMachineType: StateMachineType
1
1..n
HistoryBufferBase
+ GetNewValueScalar()=0 : void
+ GetNewValueVector()=0 : void
+ PushNewValueScalar()=0 : void
+ PushNewValueVector()=0 : void

Figure 5.11: Class diagram of the SAFECASS base filter class (`SC::FilterBase`). The extension from the original filter design[203] is represented in green, and the fault injection facility is shown in blue.

these elements is implemented within SAFECASS.

1. Input: Multiple inputs can be used, and each input is called an *input signal*. In Fig. 5.11, the `SignalElement` class implements the signal. Although the GCM is independent of a particular component model, it requires component framework-specific mechanisms to access data because each component framework maintains and manages information in a framework-specific manner. The key element that enables loose coupling between SAFECASS and the component-based framework is the history buffer, which is implemented as the `HistoryBufferBase` class that only contains pure virtual methods. This forces a component framework to provide a derived class that implements those pure virtual methods in a framework-specific manner. This derived class is called the *derived history buffer* class, and is the key element that enables SAFECASS filters to access internal data of the framework.

A new input signal can be added by `FilterBase::AddInputSignal()`. Internally, this creates a new instance of the signal (`SignalElement`) that has a pointer to an instance of `HistoryBufferBase`. Then, the newly created signal instance is added to the list of input signals that the base filter maintains (`FilterBase::InputSignals`). When a filter is executed, `RefreshSamples()` is called to update the cache of each input signal, while iterating the list of input signals.

2. Filtering Algorithm: The filtering algorithm is defined by `FilterBase::RunFilter()`. Because this is a pure virtual method, filters derived from `FilterBase` should provide a filter-specific implementation of this method. SAFECASS provides a guideline on the implementation of the filtering algorithm as follows:

1. Call `RefreshSamples()` to refresh the cache of all input signal(s).
2. Implement a filtering algorithm that uses (the history of) the input signal(s).
3. Define the serialization of event(s) that a filter may generate (via `GenerateEventInfo()`).

If an algorithm requires the history of the input signal(s), it can be implemented in the derived classes. In this way, SAFECASS allows the system designer to design and deploy any arbitrary algorithm with an arbitrary number of input signals in a flexible and extendible manner.

3. Output: The output of the GCM filter is managed in a similar manner to the input. An output signal is added by `FilterBase::AddOutputSignal()`, which internally creates a new instance of `SignalElement` and adds it to the list of output signals of each filter (`FilterBase::OutputSignals`). In the case of the input signal, the placeholder is used as a cache of the latest value retrieved from the history buffer. The output signals, on the other hand, use the placeholder as original output data.

4. Extension: The GCM filter is derived from the original filter[203] with additional features to support the GCM event semantics, as shown in Fig. 4.12. The extension and its implementation within SAFECASS include the following:

- Specification of a state machine associated with the GCM filter:

  » `FilterTargetType`

- Support for GCM event generation:

  » `FilterBase::GenerateEventInfo()` and `FilterBase::SafetyCoordinator`

- Life cycle management of the GCM filter:

  » `ConfigureFilter()`, `InitFilter()`, `RunFilter()`, and `CleanupFilter()` of `FilterBase`

- Fault injection facility
  - » `FilterBase::InjectInput()` and `FilterBase::InputQueue`

Fig. 5.11 represents these methods and variables in green. Among the methods for the life cycle management of filters, only `RunFilter()` is required to be defined by system designers; the other methods are internally called by SAFECASS. Of course, system designers can override and customize those methods, if necessary.

SAFECASS also provides the *fault injection facility* that allows system designers to inject (a vector of) test data into the system. Depending on how deep fault injection is performed, i.e., which element of SAFECASS the test data is injected to, two different modes for fault injection are supported: *shallow fault injection* and *deep fault injection*.

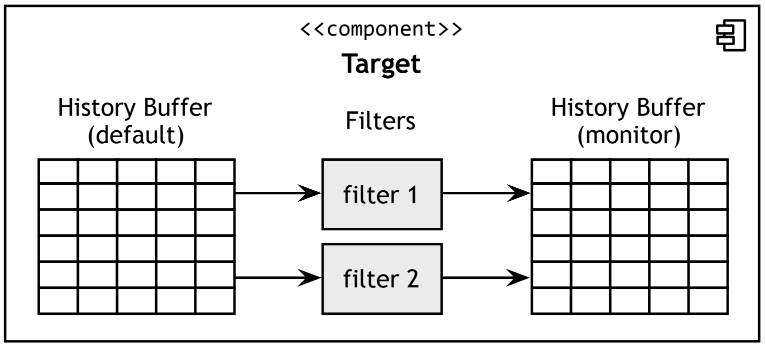
In shallow fault injection, test data is injected to the internal queue of the *filter* (`FilterBase::InputQueue`). At the next iteration of filter execution, if the internal queue is not empty, the filter dequeues one element and uses it instead of the real input signal. When the internal queue becomes empty, the filter once again uses the latest value of the real input signal.

Deep fault injection occurs in the history buffer, a level deeper than shallow fault injection. When test data is injected, the filter enqueues it to the internal queue of the history buffer, rather than the filter, by calling `PushNewValueScalar()` or `PushNewValueVector()` of the input signal, depending on the type of the test data. Because these two methods require a framework-specific implementation, deep fault injection is only available if a component framework supports the optional requirement. More specifically, the optional requirement necessitates an implementation of (1) the internal queue of the history buffer, (2) APIs to enqueue test data to the internal queue, and (3) a mechanism to "intercept" application requests for the data so that an element from the queue can be returned instead of the actual data. This latter requirement is difficult to implement in most component-based frameworks, with the exception of *cisst* (see Sec. 5.6.5.7).

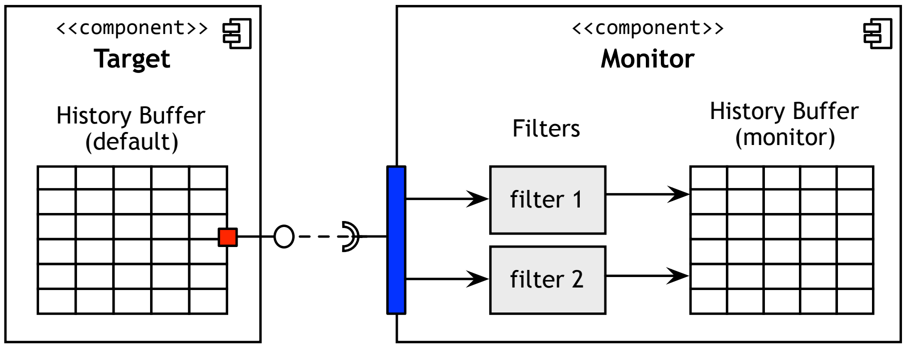
Both shallow and deep fault injection use the same API, `FilterBase::InjectInput()`, with test data to inject and a boolean flag specifying the fault injection mode. In Fig. 5.11, the fault injection facility is represented in blue.

5.6.5.2 Filtering Modes

The SAFECASS provides an option that defines two different filtering modes: *internal filtering* and *external filtering*. The main difference between the two execution modes is which thread performs the actual computation for filters. In internal filtering, filters and filter pipelines (henceforth, simply "filters") are installed on a component being monitored – the *target* component – and filters are executed by the thread of the component (Fig. 5.12a). This approach requires a "hook" into the target component to request that it execute the internal filters; typically, this can be implemented in the framework routine that invokes the component's main processing method (e.g., the `Run` method in *cisst*). The advantage of internal filtering is that all internal resources are available to the thread of the component, so it involves no other mechanism for data sampling, thereby guaranteeing that filters are executed as long as the target component runs. The disadvantage, however, is that



(a) Internal filtering



(b) External filtering

Figure 5.12: Two filtering modes in SAFECASS : Internal vs. external filtering modes

internal filtering consumes processing resources of the target component to execute filters, introducing run-time overhead. This may not be desirable in cases where components have strict restrictions on the execution time (e.g., hard real-time performance), especially when a filtering algorithm requires heavy computation.

In contrast to internal filtering, external filtering deploys filters in a separate component that monitors the target component, called the *monitor* component, and uses the thread of the monitor component to process the filters. Thus, external filtering does not use any processing resource of the target component, although this filtering requires additional run-time overhead for data retrieval from the target component. To enable external filtering, SAFECASS requires the framework to provide more substantial features. These features are implemented using the framework services and include (1) the definition, instantiation, and deployment of the monitoring component, (2) establishing connections between the monitoring component and the target component, and (3) setting up data retrieval mechanisms. As a result, the run-time characteristics of external filtering inherently follow those of the framework services.

Fig. 5.12b illustrates the two filtering modes. Essentially, the component executes filters which read input data from the history buffer, execute the filtering algorithm, and write filter output data to the history buffer. The main difference between the two modes is where the component filters are deployed (i.e., to the *target* component in internal filtering and to the *monitor* component in external filtering), and as a result, which thread executes the filtering algorithms. It should be noted, however, that the post-filtering process that generates and handles GCM events is almost the same in both cases, possibly with timing differences due to the thread execution model.

One design consideration of the external filtering approach is the component execution model of the monitor component, i.e., how the monitor component acquires and manages its execution thread, how the thread is executed, and when and how it retrieves data from the target component. If the monitor runs periodically, SAFECASS inherently becomes a *periodic system* and can take advantage of characteristics of periodic systems such as implicit flow control and the protection of the system from erroneous overload conditions due to faults.[221] However, this leads to implicit delay in filter processing and thus may result in the late detection of error events. On the other hand, if the monitor is driven by signals or events, SAFECASS becomes an *event-driven system* and filter processing delay can be minimized, but shielding an event from faults becomes difficult. That is, the system may suffer from hyperactivity due to spurious events, and thus a mechanism to suppress unwanted signals is required.[221] Also, it is necessary to separately detect underactivity of the target component; with SAFECASS, this can easily be handled by deploying framework-level filters to detect issues such as thread crash or overrun.

In summary, Table 5.1 presents a list of features that a framework should provide for the SAFECASS filter facility through the framework extension. Because the SAFECASS filter mechanism relies on these features, SAFECASS is only applicable to frameworks that provide those features. The *data storage* of the history buffer represents a framework-specific implementation of the history buffer. Any implementation of the history buffer that provides thread-safe access to data (possibly with timestamp) would suffice. The *data access* of the history buffer refers to a feature that allows SAFECASS to read data from, and to write data to, the history buffer. As described earlier, reading data is an essential requirement, whereas writing is an *optional* feature. The *filter initialization* is a feature that establishes connections between filter inputs/outputs and the history buffer on system start up, by calling the `SetHistoryBufferInstance()` method of the `SignalElement` class. The *filter execution* is a key feature that actually executes the filtering algorithm by calling the `RunFilter()` method of each filter instance at run-time. Considering the thread execution model, the system designer determines the timing of execution and which thread to use for

Table 5.1: List of features that the framework extension should provide for the filter facility of SAFECASS. Two optional features are the write-to-buffer feature and the support for the two filtering modes.

Element	Feature	Description
History Buffer	Implementation	Derived history buffer (derived from the base history buffer)
	Data Storage	Framework-specific implementation of the history buffer
	Data Access	Read data from the buffer; Write data to the buffer (*optional*)
Filter	Initialization	Connect filter input/output signals to the history buffer
	Execution	Execution of filtering algorithms
	Modes *(optional)*	Support for two filtering modes: internal/external filtering

filtering. The *filtering modes* is an *optional* feature. Depending on the data exchange model of the framework, one of the two modes may not be feasible, but at least one mode must be supported.

Again, the two optional features of the framework extension are (1) the feature to write data to the history buffer (for deep fault injection), and (2) the feature to support the two filtering modes.

5.6.5.3 Event Generation Modes

When detecting and generating GCM events via GCM filters, the event detection comes in two flavors: *edge-triggered* and *level-triggered*. In the edge-triggered mode, the GCM filter generates the onset event *only once on its first occurrence*, whereas the GCM filter in the level-triggered mode keeps generating onset events *whenever the event is detected*. In general, level-triggered events without careful design can possibly flood the system with numerous duplicate events, which may lead to problems in event handling (e.g., when/how to know if an event is handled properly and thus subsequent events should be ignored, or when the filter should stop generating events). However, duplication of events improves the robustness of the system against sporadic failures in the event delivery mechanism. In contrast, edge-triggered events may be missed if the event delivery mechanism is not reliable. But, they do not suffer from either the event flooding issue or the event handling problem, which generally leads to a simplification of the overall system design. When designing filters for the system, the system designer should take these characteristics into account.

The base filter class, `SC::FilterBase`, represents these two different modes as the `SC::EventDetectionModeType` enum. Although the base filter currently uses the edge-triggered mode by default, derived filters can override this property.

5.6.5.4 Configuration File

As in the case of GCM events, the GCM filters are defined in the JSON format. On startup, SAFECASS reads this specification either from a configuration file or from a string. Then it creates and configures instances of GCM filters based on the specification, and deploys these instances to the system as specified.

Code 5.3 presents an example of a filter specification. The filter configuration consists

Code 5.3: Example of JSON specification of GCM filters

```
{
  "component": "aComponentName"
  "filter" : [
    // filter 1: threshold filter
    {
      // -- common fields
      "class_name"     : "FilterThreshold",
      //
      "target"         : {
        // type of state machine associated with this filter
        // s_F: framework state
        // s_A: application state
        // s_P: provided interface state
        // s_R: required interface state
        "type"         : "s_F",
        "component"    : "component_name",
        "interface"    : "interface_name"
      },
      // filtering modes (INTERNAL or EXTERNAL)
      "type"           : "INTERNAL",
      // debug enable (true or false; default: false)
      "debug"          : false
      // -- threshold filter specific fields
      "argument" : {
        "input_signal" : "ForceY",
        "threshold"    : 0.8,
        "tolerance"    : 0.1,
        // output values
        "output_above" : 1,
        "output_below" : 0,
        // names of events to be generated
        "event_onset"      : "EVT_FORCE_Y_EXCESSIVE",
        "event_completion": "/EVT_FORCE_Y_EXCESSIVE"
      }
    }
  ]
}
```

of two parts: the *common* part and *filter-specific* part. The common part is required by the base filter class and contains attributes common to all filters. The `class_name` is a name of the filter class to use. SAFECASS uses this string name to dynamically create filter instances. For dynamic creation, SAFECASS provides the filter factory facility (`SC::FilterFactory`) using the Factory Pattern,[204] where a set of pre-defined filter classes are "registered" to the facility by name (of type string) at compile time, and are dynamically instantiated later based on the name. SAFECASS provides a set of off-the-shelf filters that implement basic operations, such as bypassing and thresholding, but it also allows the system designer to define and use *custom filters* (the following section describes more details about how to define the custom filters). The `target` specifies a GCM state machine with which this filter is associated. The `type` determines the filtering mode of this filter. It should be noted that the two filtering modes can be switched just by updating a single parameter. Verbose logs can be enabled or disabled per filter instance on startup via the `debug` attribute (set to false by default). It is also possible to set this attribute at run-time via the `FilterBase::EnableDebugLog()` method.

The filter-specific arguments are specified as a JSON value under the keyword `argument`. The content and format of the value are defined by an individual filter. Because the value is expressed in JSON format, the filter designers can benefit from its extendible and expressive syntax, such as a collection of name/value pairs, an ordered list of values, and various data types. In Code 5.3, for example, the thresholding filter of which class name is `FilterThreshold` requires seven attributes to be defined. These attributes are described in more detail in the next section (Sec. 5.6.5.5). Also note that the value of the `filter` keyword is defined as an array type (i.e., enclosed by JSON array delimiters `[` and `]`), so that one configuration file can contain specifications for multiple filters.

5.6.5.5 Basic Filters

The SAFECASS provides a set of off-the-shelf filters as part of the core library. They are called the *basic filters*, and three basic filters are currently available: the bypass, change

Table 5.2: List of the off-the-shelf basic filters that SAFECASS provides. The `Type` field follows the data type definition of JSON.[10]

Class Name	Filter-specific Attributes		
	Name	Type	Description
`FilterBypass`	`input_signal`	string	Name of input signal
`FilterChangeDetect`	`input_signal`	string	Name of input signal
	`baseline`	number	Baseline value
	`event_onset`	string	Name of onset event
	`event_completion`	string	Name of completion event
`FilterThreshold`	`input_signal`	string	Name of input signal
	`threshold`	number	Threshold value
	`tolerance`	number	Tolerance (margin) value
	`output_above`	number	Output value if input ≥ (`threshold` + `tolerance`)
	`output_below`	number	Output value if rising edge was detected and input < (`threshold` + `tolerance`)
	`event_onset`	string	Name of onset event
	`event_completion`	string	Name of completion event

detection, and threshold filters. Table 5.2 presents a list of these basic filters with the definition of filter-specific attributes.

The *Bypass* filter is the most basic filter where its output is identical to its input. Although this filter may not be particularly useful in practice, its minimal structure provides a reference for system designers on how to design filters.

The *ChangeDetect* filter generates GCM events if the value of the new input is (i) different from the *baseline*, and (ii) different from that of the last input. The intended use of this filter is to monitor discrete signals, such as counters and flags, to detect changes and to generate onset and completion events. The onset event occurs when a new input changes from the baseline to any other value, and the completion event occurs when a new input becomes the baseline. Conceptually, the onset and completion event correspond to the rising and falling edge (edge-triggered). The baseline is specified as the `baseline` attribute, and the names of onset and completion event are defined as the `event_onset` and `event_completion` attributes. This filter is suitable for detecting changes of digital signals, rather than analog signals that tend to fluctuate.

The *Threshold* filter checks if a new input exceeds the `threshold` value with a margin of `tolerance`. If the input exceeds the threshold beyond the margin, the filter output becomes `output_above` and the `event_onset` event is generated. If an `event_onset` event occurred earlier and a new input becomes less than `threshold` with the margin, the filter output becomes `output_below` and the `event_completion` event is generated.

Despite the short list of the basic filters, they are comprehensive, flexible, and expressive enough (1) to replace some existing application-specific safety features with filter-based implementations, and (2) to design new framework-specific safety features. The next chapter (Chap. 6) illustrates the use of these filters for two different medical robot systems.

5.6.5.6 Custom Filters

The basic filters are useful for quickly prototyping safety features because they can be easily deployed to the system if a JSON specification is provided. If the basic filters are not adequate for application-specific safety features, system designers can define new filters, called *custom filters*. The SAFECASS filter facility helps the system designer easily define custom filters by providing pre-defined macros.

Assuming the name of a new custom filter is “FilterCustom”, a new filter can be defined with the following four steps:

1. Define a new filter class: Define a custom filter class, `FilterCustom`, that is derived from the base filter class (`SC::FilterBase`).

2. Define input(s) and output(s): Add input and output signal(s) to the constructor of `FilterCustom` by calling `AddInputSignal()` and `AddOutputSignal()`.

3. Define filtering algorithm (and life cycle handlers): Implement a custom filtering algorithm in `RunFilter()`. Optionally, the other three filter life cycle handlers can be overridden by defining `ConfigureFilter()`, `InitFilter()`, and `CleanupFilter()`.

4. Register a new filter to SAFECASS: Register `FilterCustom` to SAFECASS using two helper macros that SAFECASS provides. First, add the following macro to the header file:

```
#include "filterBase.h"

class SCLIB_EXPORT FilterCustom: public SC::FilterBase
{
  // here come other declaration and definitions

  SC_DEFINE_FACTORY_CREATE(FilterCustom);
};
```

Next, add another macro to the source file to register the new class to the SAFECASS filter factory:

```
#include "filterCustom.h"

SC_IMPLEMENT_FACTORY(FilterCustom);
```

This macro internally instantiates an instance of the filter, enabling the filter factory to dynamically create additional instances of the filter. Once these steps are completed, the new filter class, “FilterCustom”, can be used in the JSON specification, as if it were one of the basic filter classes.

5.6.5.7 *cisst* Implementation

Table 5.1 summarizes a list of features that have to be provided for SAFECASS through the framework extension. Based on this table, Table 5.3 describes how the extension of *cisst* implements each feature.

In *cisst*, the `mtsHistoryBuffer` class is added to the *cisstMultiTask* library to provide an implementation of the derived history buffer. `mtsHistoryBuffer` internally maintains a pointer to an instance of the *cisst state table*, which provides a thread-safe, efficient, and lock-free data exchange mechanism between components (Kazanzides *et al.*, 2008)[222][i]. The derived history buffer in *cisst* uses the state table as its data storage back-end, and supports both read and write operations on the state table. Because *cisst* components systematically use the state table to provide data to other components, it is possible to support deep fault injection, as described in Sec. 5.6.5.1. Specifically, the SAFECASS can inject values into

[i]Recently, the correctness of the state table was verified using formal methods (Kazanzides *et al.*, 2012).[120]

Table 5.3: Summary of the extension of *cisst* that provides framework-specific features for Safety Architecture for Engineering Computer-Assisted Surgical Systems

Element	Feature	Implementation in *cisst*
History Buffer	Implementation	`mtsHistoryBuffer` in *cisstMultiTask* (new)
	Data Storage	`mtsStateTable` in *cisstMultiTask* (existing)
	Data Access	`mtsStateTable` in *cisstMultiTask* (both read and write supported)
Filter	Initialization	`mtsSafetyCoordinator` in *cisstMultiTask* (new)
	Execution	`mtsStateTable::Advance()` executes `FilterBase::RunFilter()`
	Modes *(optional)*	`mtsSafetyCoordinator` (both filtering modes supported)

the target component state table, which are then used by other components; note, however, that the target component typically uses its member data directly and therefore would not be affected by changes to its state table.

The filters are initialized by the `mtsSafetyCoordinator` class that is derived from the base coordinator, which is described in the next section. Filters are executed by the component to which they are deployed. That is, internal filters are executed by the target component, whereas external filters are executed by the monitor component. Within the component, filters are executed by the `mtsStateTable::Advance()` method, which is called at every iteration of the component processing loop, after the user-defined code in the `Run` method. In addition, *cisst* supports both the internal and external filtering modes. A filter is added by calling the `AddFilter()` method of `mtsSafetyCoordinator`; internally, this filter calls either `AddFilterInternal()` or `AddFilterExternal()`, depending on whether it is an internal or external filter, respectively.

5.6.6 Coordinator

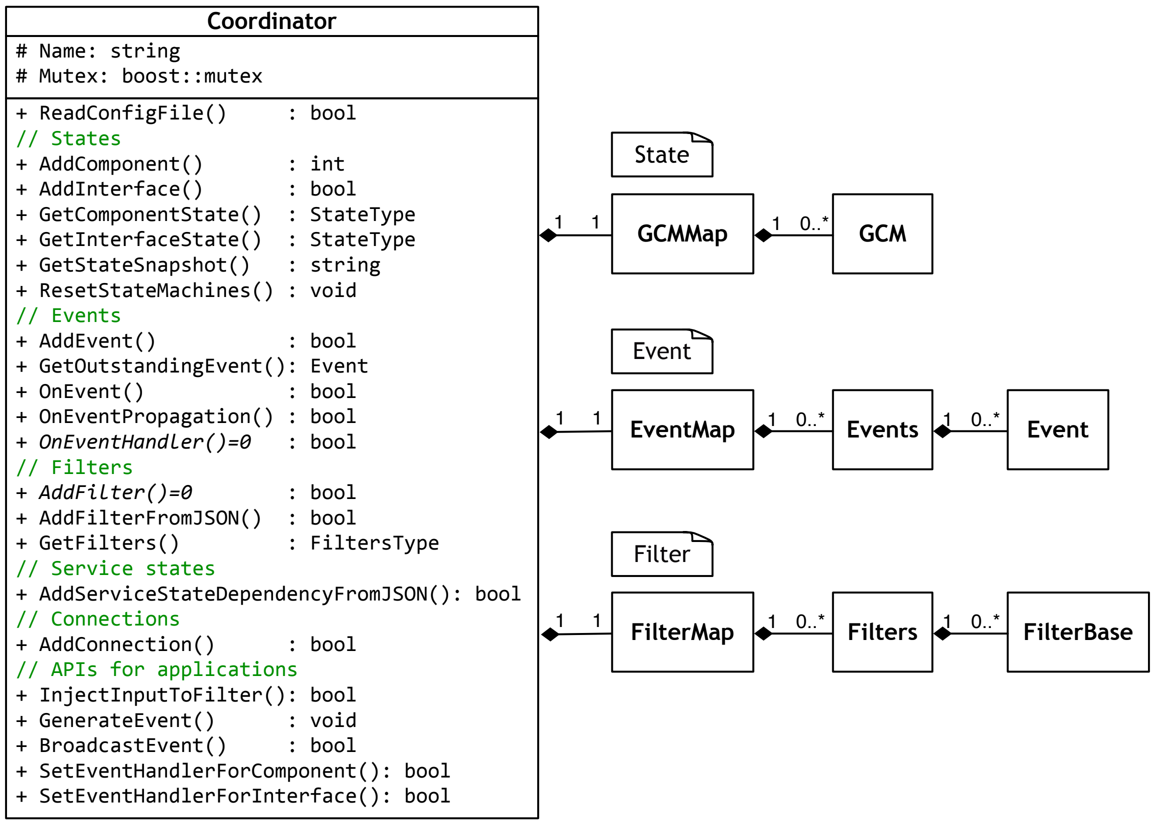
The base coordinator is the key entity that enables a run-time environment for GCM. The core features that it provides include (1) maintaining the system status in terms of the GCM states, (2) providing the framework layer with APIs that facilitate an implementation of the framework extension for SAFECASS, and (3) coordinating states and events in accordance with the GCM semantics. The `SC::Coordinator` class provides an implementation of the coordinator, as illustrated in Fig. 5.13.

The coordinator maintains three key data structures: `GCMMap`, `EventMap`, and `FilterMap`. The `GCMMap` is a set of instances of the `GCM` class, which implements the structural elements of the GCM. For each component in the system, an instance of the `GCM` is created to maintain a complete set of GCM state machine instances of the component, connection topology of the system, and service dependency information. The state of a particular component or an interface can be queried by `GetComponentState()` or `GetInterfaceState()` of `Coordinator`:

```
State::StateType GetComponentState(const std::string & componentName,
                                   const Event* & e,
                                   GCM::ComponentStateViews view = SYSTEM_VIEW) const;
State::StateType GetInterfaceState(const std::string & componentName,
                                   const std::string & interfaceName,
                                   const Event* & e,
                                   GCM::InterfaceTypes type) const;
```

It is also possible to fetch the entire set of states at once via `GetStateSnapshot()`.

The `EventMap` maintains a set of `Events`. Each instance of `Events` contains a list of



```
Coordinator
# Name: string
# Mutex: boost::mutex
+ ReadConfigFile()     : bool
// States
+ AddComponent()       : int
+ AddInterface()       : bool
+ GetComponentState()  : StateType
+ GetInterfaceState()  : StateType
+ GetStateSnapshot()   : string
+ ResetStateMachines() : void
// Events
+ AddEvent()           : bool
+ GetOutstandingEvent(): Event
+ OnEvent()            : bool
+ OnEventPropagation() : bool
+ OnEventHandler()=0   : bool
// Filters
+ AddFilter()=0        : bool
+ AddFilterFromJSON()  : bool
+ GetFilters()         : FiltersType
// Service states
+ AddServiceStateDependencyFromJSON(): bool
// Connections
+ AddConnection()      : bool
// APIs for applications
+ InjectInputToFilter(): bool
+ GenerateEvent()      : void
+ BroadcastEvent()     : bool
+ SetEventHandlerForComponent(): bool
+ SetEventHandlerForInterface(): bool
```

Figure 5.13: Class diagram of the SAFECASS base coordinator class (`SC::Coordinator`)

registered events to a component, and explicitly defines which events may occur within the component, based on the event specification, as described in Sec. 5.6.4.1. If an event occurs, `OnEvent()` is called with detailed event information encoded in the JSON format. Internally, `OnEvent()` calls `OnEventHandler()` to inform the framework of the occurrence of the event, thereby allowing the framework to handle the event in a framework-specific manner. When a service state changes and an error event is propagated across the component boundary, `OnEventPropagation()` is called to handle event propagation.

The `FilterMap` has the same structure as the `EventMap`; it is a set of `Filters` that is associated with each component and maintains a list of filter instances deployed to the component. Although the coordinator provides helper methods to add filters, the method that actually deploys a filter to the framework, i.e., `AddFilter()`, must be provided by the framework as part of the framework extension for SAFECASS.

Additionally, `Coordinator` provides a set of utility functions for applications, including `InjectInputToFilter()` for fault injection, `GenerateEvent()` for manually (i.e., without GCM filters) generating GCM events, and `BroadcastEvent()` for manually generating GCM broadcast events. The APIs of these functions are as follows:

```
bool InjectInputToFilter(FilterBase::FilterIDType fuid,
                         const std::vector<double> & inputs,
                         bool deepInjection = false);

bool InjectInputToFilter(FilterBase::FilterIDType fuid,
                         const std::vector< std::vector<double> > & inputs,
                         bool deepInjection = false);

void GenerateEvent(const std::string &      eventName,
                   State::StateMachineType type,
                   const std::string &      what,
                   const std::string &      componentName,
                   const std::string &      interfaceName = "");

bool BroadcastEvent(const std::string & eventName, const std::string & what);
```

5.6.6.1 *cisst* Implementation

In *cisst*, the *cisstMultiTask* library provides most of the framework extension, which defines the `mtsSafetyCoordinator` class derived from the base coordinator (`SC::Coordinator`). Currently, `mtsSafetyCoordinator` is implemented as part of the *cisst* local component manager, i.e., the `mtsManagerLocal` class. This class effectively corresponds to the Manager Component Client (MCC) in Fig. A.3. Thus, `mtsSafetyCoordinator` is instantiated and deployed only once per process in parallel with the *cisst* local component manager.

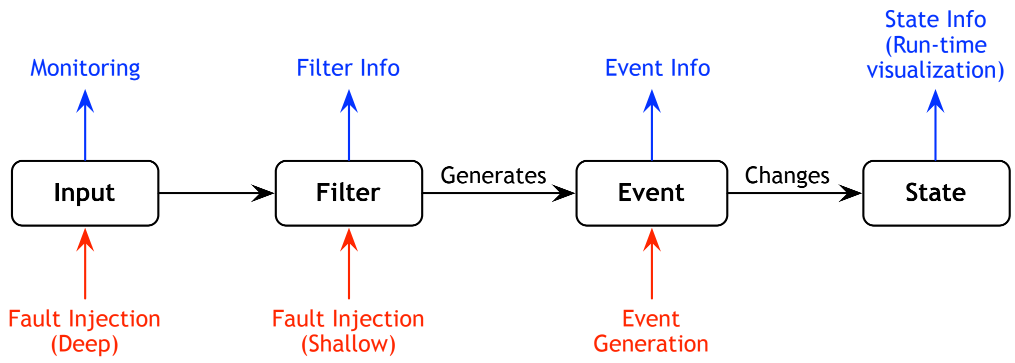


Figure 5.14: SAFECASS tool support: *console* enables access to the internal processing pipeline of the Generic Component Model. This access includes both read and write operations, each represented in blue and red, respectively.

5.6.7 Tool Support

SAFECASS provides tools to facilitate the development process. These tools are designed with no dependency on services that the framework provides. This self-contained design allows the tools to be used across different frameworks. Currently, SAFECASS supports two tools: *console* and *viewer*.

5.6.7.1 *Console*: Interactive Console

The *console* is a command line utility to interact with the system. The core feature of this tool is to allow system designers to access (read/write) the internal processing pipeline of the GCM that fundamentally defines and controls the behavior of the system. Fig. 5.14 depicts this concept, where blue and red represent read and write operations, respectively.

The features that the *console* provides center around the four elements of the internal processing pipeline: input, filter, event, and state. The *input* to the filter can be retrieved for monitoring and overridden for testing purposes (deep fault injection) through the fault injection facility, as described in Sec. 5.6.5.1.

Using *console*, it is possible to access the information about *filters*, both as a group and individually, such as a list of filters in the system, a list of input and output signals of a particular filter, and the current status (enabled/disabled) of a filter. The shallow fault injection can also be easily performed via *console*.

In addition, the *console* can show the information of any *event* registered to the system. It can also directly generate GCM events using the SAFECASS event generation mechanism. This event generation is distinct from the filter-based event generation in that events generated by filters are handled by a pre-defined state machine associated with the filter, whereas events generated by the direct mechanism can be handled by any arbitrary, user-specified state machine. The *console* can generate broadcast events as well.

The *state* information of the system can be retrieved in its entirety in the JSON format. Because states can change only with the occurrence of GCM events, the *console* does not allow any state to be manually set.

Other features that *console* provides include showing a list of connections, information about service state dependencies, and fault injection of a series of data from a file.

Based on our experience, we find that the features that the *console* provides are particularly useful for debugging and introspecting the run-time system status.

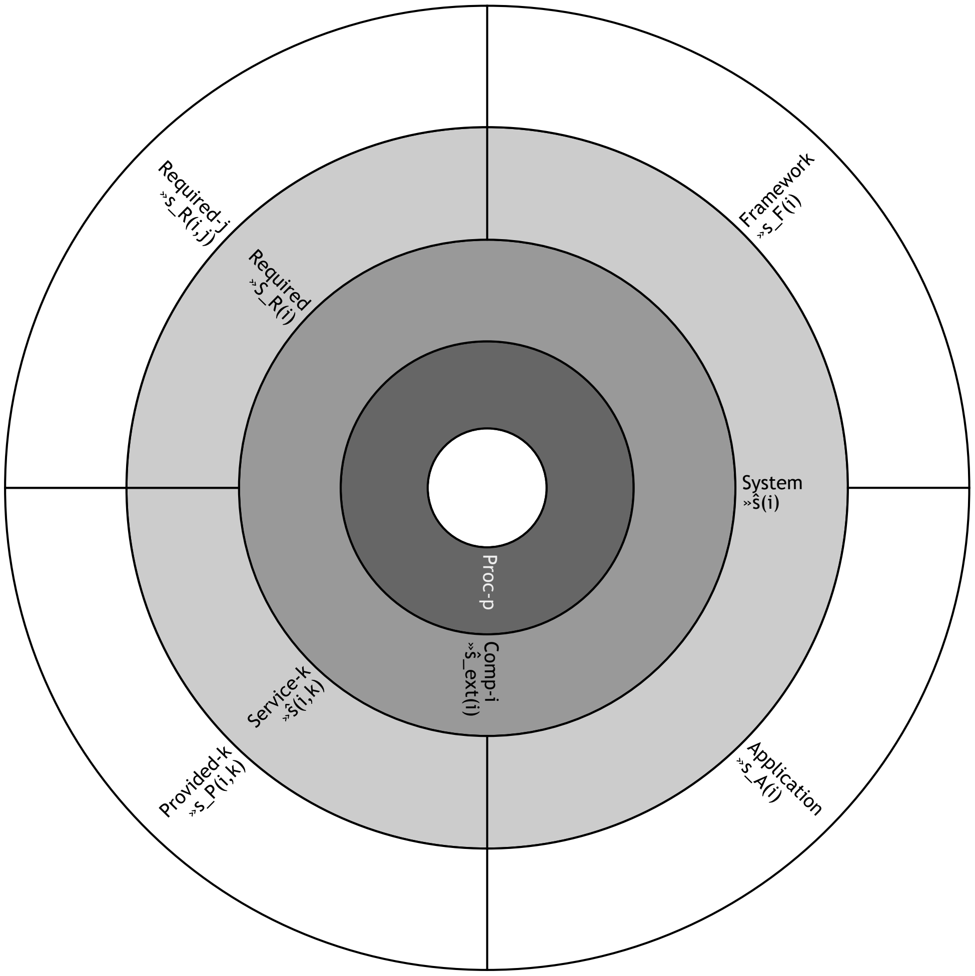


Figure 5.15: SAFECASS tool support: *viewer* visualizes the current states of the system.

5.6.7.2 *Viewer*: Run-time State Viewer

The *viewer* is a graphical user interface (GUI) tool that visualizes the run-time status of component-based systems in an effective, intuitive manner. The *viewer* uses the D3 package[223] as its visualization module and the Qt framework for its GUI.

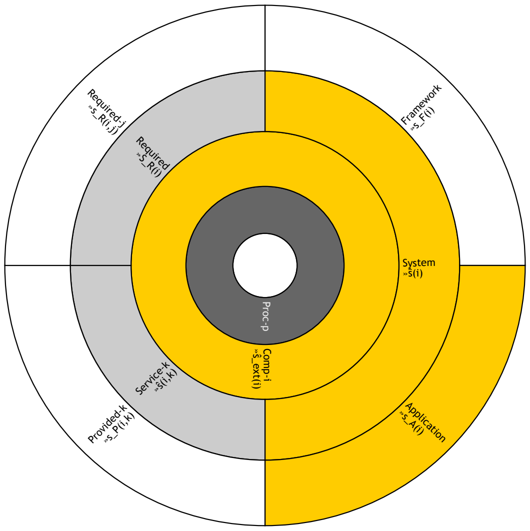
Fig. 5.15 shows how the *viewer* visualizes the states of the system in a structured manner. The innermost layer in the darkest color (`Proc-p`) is called the *process* layer, and represents a set of *processes* in the system. Although the figure shows only one process (p-th process), multiple segments will show up in case of multi-process systems. The next layer is called the *component* layer. Under the segment of the p-th process, this layer contains a set of segments where each segment, shown as `Comp-i`, represents the extended component state, $\hat{s}_{ext}(i)$ (Eq. 4.6), of the i-th component in the p-th process. The next outer layer, called the *meta* layer, consists of a group of three segments that present three different types of states:

- The `System` segment: Component states in the system view, $\hat{s}(i)$ (Eq. 4.1a)
- The `Required` segment: State product of all required interface states, $\boldsymbol{S_R}(i)$ (Eq. 4.4a)
- The `Service-k` segment: Service state of the k-th provided interface, $\hat{s}(i,k)$ (Eq. 4.5)

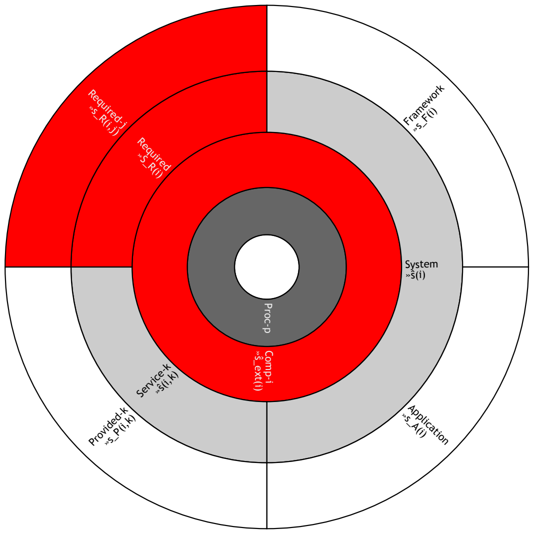
The outermost layer represents a set of *actual* states. In contrast to the other layers that present derived states or a collective form of states, this layer contains only the actual states. Within each segment of the meta layer, the following state variables are presented:

- The `System` segment: Component state in the framework view, $s_F(i)$ (Eq. 4.1b), and the same state in the application view, $s_A(i)$ (Eq. 4.1c)
- The `Required` segment: j-th required interface state, $s_R(i, j)$ (Eq. 4.3a)
- The `Service-k` segment: k-th provided interface state, $s_P(i,k)$ (Eq. 4.3b)

When a GCM event occurs, the event is handled by a designated, actual state machine that corresponds to one of the segments in the outermost layer. This event is then consecutively propagated by the event propagation mechanism towards the center, crosses the process boundary, and is handled by other relevant state machine(s). Whenever any state changes,



(a) Occurrence of *Warning* event



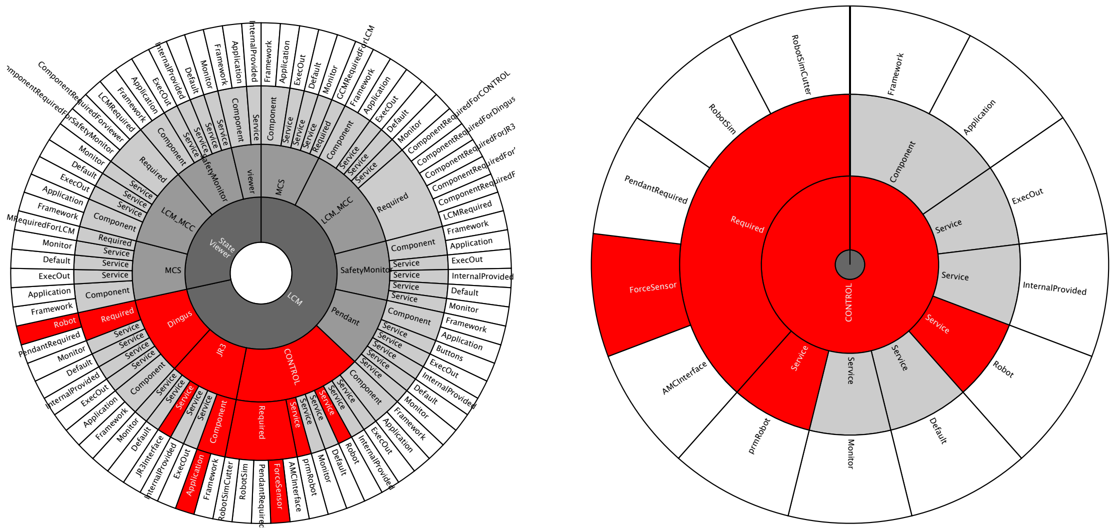
(b) Occurrence of *Error* event

Figure 5.16: The *viewer* visualizes the operational status of the overall system in an intuitive, effective, and structured manner.

the coordinator publishes the system status update message to the SAFECASS network, making the *viewer* refresh its visualization.

The color of each segment indicates the current state of the state machine corresponding to the segment. The *Normal*, *Warning*, and *Error* states are represented as white, yellow, and red. As illustrated in Fig. 5.16, this color-coded state representation visualizes the overall status of the system in an intuitive, effective, and structured manner. This tool can also visualize which parts are affected by the event. For example, Fig. 5.17a shows a snapshot of a SAFECASS-based system where an *Error* event occurred. Despite a large number of states in the system, it is possible to visually identify which part of the system was affected by the event.

The *viewer* is also interactive. If there is any outstanding event associated with a segment,



(a) Visualization of complex system with *Error* **(b)** Interactive chart showing a subset of the system

Figure 5.17: The *viewer* allows the system designer to visually inspect the impact of *Error* events with intuitive and interactive visualization.

the detailed information of the event is displayed as a pop up on the chart. Furthermore, it is possible to click any segment on the chart to "zoom in" the segment, so that only the segment clicked and its child segments are displayed. As an example, if the segment corresponding to the component "`CONTROL`" in Fig. 5.17a is clicked, only the segment and its child segments are displayed, as in Fig. 5.17b. In this case, the *viewer* hides the process layer and the component layer becomes the innermost layer. If the circle at the center is clicked, the *viewer* returns to the default view, i.e., Fig. 5.17a.

5.7 Framework Support: *cisst*

In the framework-based architecture, it is essential to guarantee that the framework provides correct, reliable services for the application layer because the correct execution of the application inherently relies on those services. Currently, the SAFECASS only supports the *cisst* component-based framework. This section describes the SAFECASS-based safety features that we introduced to *cisst*.

The framework layer has less variability than the application layer, and provides generic and application-independent services for the application layer. From a variety of different framework-level events that may cause adverse effects on the system, we select three events as exemplary cases: *thread overrun*, *thread exception*, and *command queue full* events. These events are among the major threats to the correct and timely services of the framework layer. Despite their potentially critical impact on the system, software systems are often implemented with the ideal assumption that timing errors will never occur.[224] Within large and complex component-based systems, it tends not to be easy to systematically detect those events, nor to identify the source of errors. In the following sections, we describe three safety features that we developed to address each event in a systematic and structured manner based on SAFECASS. The specifications of these safety features in *cisst* are presented in their entirety in Code B.2 in Appendix B.

5.7.1 Thread Overrun

The first safety feature is a facility that systematically manages *thread overrun* events. Robot control systems typically run under real-time operating systems, primarily because of the strict timing requirements to maintain control stability. Despite correct design, thorough analysis, and extensive testing, timing errors do occur in practice, due to, for example, interrupts occurring more frequently or executing longer than expected and variations in processing speed.[224] Among these, the most common timing error is the *missed deadline*, where a real-time thread fails to complete its execution on time.[224] In practice, such timing errors are not detected until more catastrophic failures occur, and can lead to service failures with non-obvious reasons for the cause of such failures. In the real-time embedded domain, or more broadly speaking, the high-assurance software systems domain, there exists prior work on the detection and handling of timing errors (e.g., Stewart *et al.* (1992,[209] 1997[209]), Caccamo *et al.* (2002),[225] Santos (2008)[226]).

The *cisst* framework only provided preliminary support for the detection of thread overrun. Specifically, the base component class with a thread maintained a boolean variable (`mtsTask::OverranPeriod`) that represented the thread overrun status, but it was the component designer or application developer's role to check the variable and handle the thread overrun events; the framework did not handle thread overrun events. That is, these events are ignored unless the component designer or application developer – hereafter, the *user* – explicitly handles those events. Due to additional implementation overhead on the users, the system's reliance on users to add the manual checking of the overrun status would be problematic in practice as the number of components increases.

Motivated by these necessities, we introduced SAFECASS to *cisst* and implemented a mechanism to detect and handle thread overrun events, as a new safety feature within *cisst*. With this change, the design of *cisst* is improved such that the framework is more involved in the detection and handling of thread overrun events. First, we define a pair of GCM events: `EVT_THREAD_OVERRUN` and `/EVT_THREAD_OVERRUN`. Each represents the onset and completion events of thread overrun. Then, we use the *Threshold* filter to detect thread overrun. *cisst* creates an instance of the filter for every component that is executed *periodically*, i.e., of type `mtsTaskPeriodic`, and deploys the filter instance to the component with the threshold set to the nominal period of each component. At each iteration of those components, the actual execution time of the user code is *monitored*. When a filter is executed, the onset event is generated if a thread overrun is *detected*, changing the component state from N to W. As *reaction*, the user handles thread overrun in an application-specific manner. Different timing error handling techniques can be adopted, such as use of soft real-time threads, imprecise computations, and adaptive real-time scheduling.[224] In the following iteration, the completion event is generated if a component is *recovered* from thread overrun, i.e., the execution time is less than the threshold. This completion event restores the component state back to N. Note that the GCM events associated with thread overrun declare state transitions only between N and W. Thus, the component state cannot be E due to thread overrun.

The major benefit of the SAFECASS-based safety feature is that the framework is more heavily involved in the detection and handling of thread overrun than before, thereby enabling automatic, configurable installation of the safety feature throughout the system. In addition, the user does not need to check the overrun status manually for each component.

This greatly simplifies the application code. Furthermore, the user can focus on application-specific content for reaction and recovery, rather than dealing with monitoring and detection as part of the application-specific logic.

5.7.2 Thread Exception

The enhancement of the *thread exception* handler is the second safety feature that we added to *cisst*. When a thread executes the user code, the occurrence of C++ exceptions leads to the partial and incomplete execution of the application logic. This may not affect some components with little logic, such as sensor or device "wrapper" components because sensor feedback may be available at the next iteration. However, it would be a critical issue for components with heavy, complex logic, such as robot control components, because part of the code that the thread failed to execute may render the service unavailable, or may break the consistency of internal states.

cisst provides a facility that handles thread exceptions as part of the framework services (`OnStartupException()` and `OnRunException()` of `mtsTask`). The default behavior of these handlers was simply to leave a log message about the occurrence of the exception. Although the user can override these handlers to implement an application-specific exception handler, the fact that an exception occurred was not propagated to other components. Because of the potential possibility of unavailable services due to repetitive thread exceptions, the lack of error propagation is not desirable.

To enable error propagation as part of the exception handlers, we defined two GCM events, `EVT_THREAD_EXCEPTION` and `/EVT_THREAD_EXCEPTION`. Compared to the thread overrun events that are generated by the filter, these events are directly generated by a call to the `GenerateEvent()` API that the SAFECASS Coordinator provides. The onset event is generated at the end of the existing exception handlers, whereas the completion event is generated on the completion of execution of the user code. The two events define transitions between N and E, which leads to error propagation according to the GCM semantics (Sec. 4.3.7).

5.7.3 Command Queue Full

Lastly, the detection and reaction of the command queue full error is a new safety feature added to *cisst* using the SAFECASS services. *cisst* uses the Command Pattern[204] for data exchange between components – the service *requester* and the service *provider* – where an interface provides services via *command objects*. Each command object maintains internal first-in, first-out (FIFO) queues to process requests. A command queue full error occurs when any of these internal queues becomes full because of either too slow request processing (i.e., dequeuing) or too fast command requests (i.e., enqueuing), and thus cannot enqueue any more requests. Once this happens, further requests are dropped and thus in effect ignored until at least one pending element is dequeued and processed.

The previous implementation of *cisst* supported the detection of the command queue full errors. When they occurred, the service provider component generated log messages and returned a particular error code (`mtsExecutionResult::COMMAND_ARGUMENT_QUEUE_FULL`).

The service requester component checked this return code to determine if its last request was successfully queued. There are several issues with this implementation. First, it was difficult to identify and locate the exact source of the error when it occurred. From the service provider's perspective, error logs did not identify the complete information of the owner of a command object, i.e., names of the component and the interface to which the command object belongs. These logs only showed the name of the command object associated with the full queue. Secondly, the service requester had no means to check the status of the service provider other than the return code. Thus, the system designer had to *manually* check the return code for *each and every* function call for command requests as part of the application logic. This led to the duplication of error handling code for every command request. For this reason, system designers in practice were tempted to skip error handling or ignored the return code and dealt with such issues by trial-and-error that typically involved time-consuming debugging processes. Lastly, it was not feasible to perform prognostic fault detection; the command queue full event was detected *only after* they actually occurred.

To address these issues, we applied SAFECASS down to the command object-level. We first refactored the previous design of the command object so that each instance of the command object maintains the names of the owner component and the owner interface. Then, we defined two GCM events, `EVT_COMMAND_QUEUE_FULL` and `/EVT_COMMAND_QUEUE_FULL`, to represent the onset and completion events. Next, we added code snippets to generate the onset event on the detection of the command queue full error. With the onset event, the state of the provided interface to which the command object belongs becomes E. The service state of the provided interface is successively changed to *E* by the GCM error propagation. As a result, all the connected components are notified of the error and thus have a chance to react to the error accordingly. The completion event is generated when the current state of the provided interface is *E* and the queue is not full. With this new design, it is now possible to exactly identify which command objects cause problems. Also, the service requester can check the current status of the service provider *prior to* actually committing a request. This allows the service requester to adaptively control the flow of command requests, thereby achieving more reliable communication between components. In addition, this new facility would simplify the code-level implementation for command requests, return code checking, and error handling on the requester's side.

Currently, we use code snippets embedded in the command-level to generate the GCM events *on the detection* of the command queue full error, but it is possible to replace the embedded code snippets with a filter-based scheme. With slight modification, this design can be further improved to enable *prognostic fault detection*, so that a *Warning* event can be generated and handled prior to the actual occurrence of an *Error* event. This can be easily achieved with just two steps: (1) registering the remaining size of each queue to the history buffer and (2) installing a threshold filter on each command object instance with a threshold of, for example, 80% of the queue size (this can be adjusted simply by modifying a JSON specification). This would help to consolidate code snippets distributed over the code into a central place (e.g., constructors of command objects), thereby simplifying code and thus improving code maintainability.

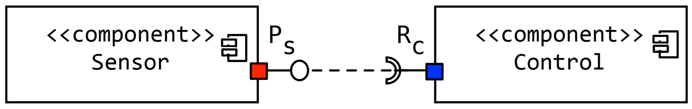


Figure 5.18: Simple system with two components

5.8 Illustrative Example

To illustrate overall operation of SAFECASS, we present experimental results based on data that we collected using a simple system with two components, as shown in Fig. 5.18[ii]. The `Sensor` component represents a typical sensor or hardware device wrapper component where its sensor feedback is periodically updated and is provided via the provided interface, P_S. The `Control` component reads sensor feedback from P_S via its required interface, R_C, and implements the control logic.

For demonstration purposes, the `Sensor` component internally maintains two independent variables – `Value1` and `Value2` – and defines two pairs of events: `(/)EVT_WARNING` (severity: 10, transition: $N \rightleftarrows W$) and `(/)EVT_ERROR` (severity: 20, transition: $N \rightleftarrows E$). Two instances of the *Threshold* filter (Sec. 5.6.5.5) are installed to generate these events. Each filter instance monitors `Value1` and `Value2` against thresholds of 5.0 and 10.0, respectively, and generates onset or completion events if the value exceeds or falls below the threshold. To simplify the demonstration, the `Control` component defines no event and only handles errors propagated from the `Sensor` component. Both components are executed at 10 Hz and we simulate warning and error conditions by changing the two variables of the

[ii]This simple system is implemented and provided as one of the examples of the Safety Architecture for Engineering Computer-Assisted Surgical Systems.

`Sensor` component.

To demonstrate the differences between the filter's two modes of event generation (Sec. 5.6.5.3), we repeat the same test scenario with filters in each mode. The test scenario defines a particular pattern of the event generation: a warning event → an error event → a warning event superseded by an error event (event prioritization). Although the actual values for the input variables are randomly generated, the pattern and timing of warning and error conditions are almost the same for both cases.

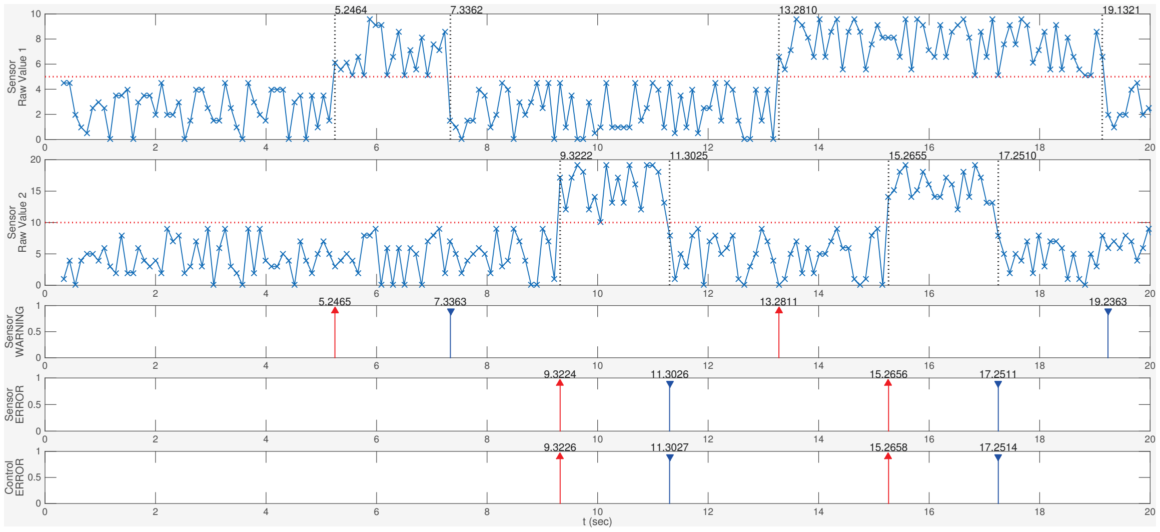
5.8.1 Edge-triggered Mode

Fig. 5.19 presents the results when the *Threshold* filters are configured to operate in the *edge-triggered* mode.

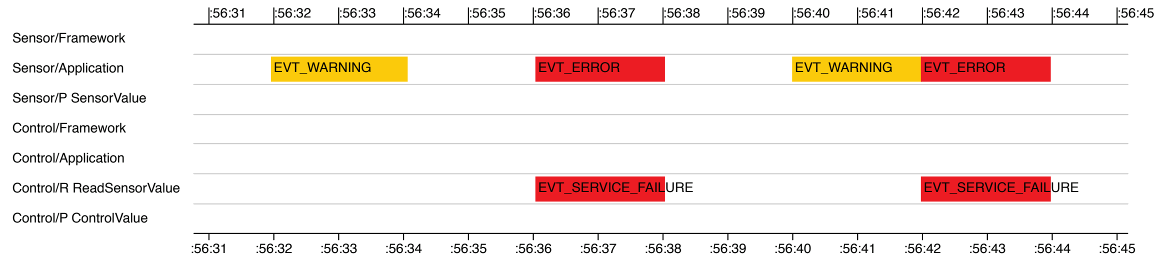
Initially, both values are below the threshold and nothing occurs. At t=5.2464 (sec), the filter of the `Sensor` component detects that `Value1` is larger than the threshold of 5.0, and immediately generates `EVT_WARNING` at t=5.2465. At t=7.3362, the filter detects that `Value1` falls below the threshold and thus generates `/EVT_WARNING` at t=7.3363.

`(/)EVT_ERROR` occurs in a similar manner to `(/)EVT_WARNING` at t=9.3224 and t=11.3026 when `Value2` exceeds and falls below the threshold at t=9.3222 and 11.3025. Because of the connection between the two components, `(/)EVT_ERROR` are propagated to the `Control` component (`EVT_SERVICE_FAILURE`), changing its state from *Normal* to *Error* on t=9.3226 and back to *Normal* on t=11.3027.

At t=13.2810, another instance of `Value1` exceeding the threshold is detected and



(a) Timing diagram: Dotted horizontal lines in red represent thresholds, dotted vertical lines in black indicate the time of event detection (in seconds), and up/down arrows in red/blue show onset and completion events.



(b) Event and state diagram: Each block in yellow/red represents *Normal* and *Error* events with the name of the outstanding event above the blocks. The horizontal axis is the time (":mm:ss" format) and the vertical axis represents a set of states in the system.

Figure 5.19: Event and state change timeline with edge-triggered filters

`EVT_WARNING` is immediately generated at t=13.2811, setting the outstanding event to `EVT_WARNING`. Now, `Value2` exceeds the threshold of 10.0 at t=15.2655 – *before* `Value1` falls below the threshold – and `EVT_ERROR` is generated, changing the state of the `Sensor` and `Control` components to *Error*. Because the severity of `EVT_ERROR` (20) is higher than that of `EVT_WARNING` (10), `EVT_WARNING` is superseded by `EVT_ERROR` and the outstanding event becomes `EVT_ERROR`. This is an example of the *event prioritization*. At t=17.2510, `Value2` falls below the threshold and `EVT_ERROR` is generated, resetting the states back to *Normal*.

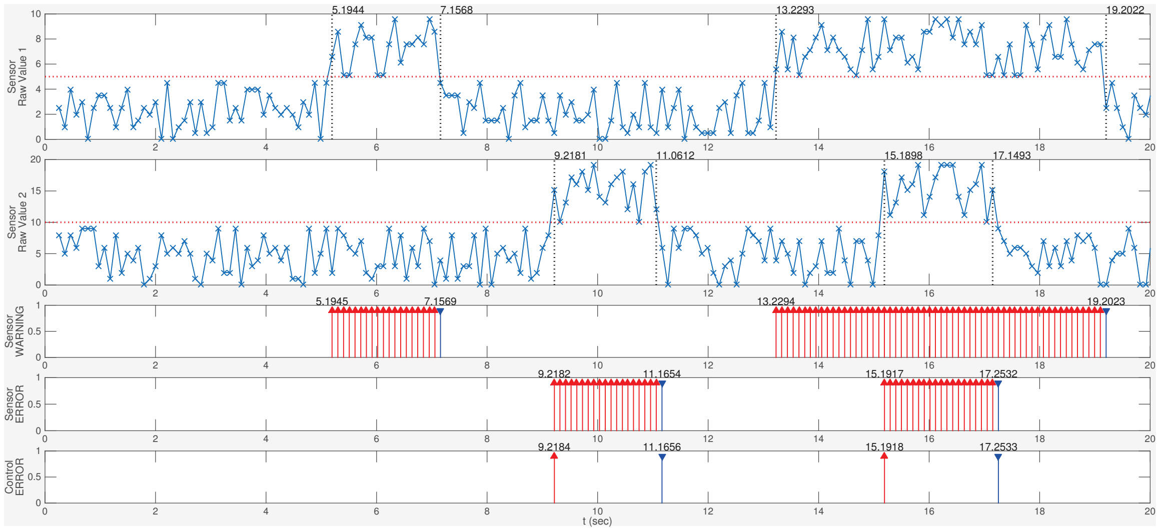
Although `/EVT_WARNING` is generated at t=19.2363, this event is ignored because the current state is *Normal*.

Fig. 5.19b presents a different view of the same experiment. It visualizes the timeline of state changes, the names of outstanding events, and the duration of each outstanding event.

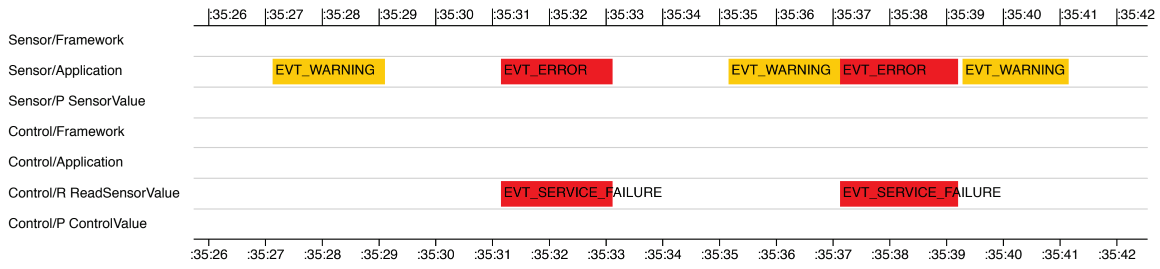
5.8.2 Level-triggered Mode

Fig. 5.20 shows the behavior of the same system when filters are set to the *level-triggered* mode. In terms of timing, the behavior is mostly similar to the previous case. The major difference is that the filter keeps generating the onset events in the level-triggered mode until the completion events are generated.

This inherent difference results in one significant difference when event prioritization occurs. In the edge-triggered mode, after `/EVT_ERROR` was generated for the second time, the state was changed to, and remained at, *Normal* due to the event prioritization. In a way, `EVT_WARNING` was indirectly resolved or suppressed by `/EVT_ERROR`. In case of the level-triggered mode, however, the state is changed to *Normal* when `/EVT_ERROR` occurs, and then is changed to *Warning* again after a sampling period (0.1 seconds in our setup), because the filter generates `EVT_WARNING` at the next iteration. This results in different states after `/EVT_ERROR` occurs: *Normal* in the edge-triggered mode and *Warning* in the level-triggered mode. This difference is easily noticeable when we compare the two event and state diagrams (Fig. 5.19b and Fig. 5.20b). Note that errors propagated from the `Sensor` component, `EVT_SERVICE_FAILURE`, are still handled in the same manner as before,



(a) Timing diagram: Dotted horizontal lines in red represent thresholds, dotted vertical lines in black indicate the time of event detection (in seconds), and up/down arrows in red/blue show onset and completion events.



(b) Event and state diagram: Each block in yellow/red represents *Normal* and *Error* events with the name of the outstanding event above the blocks. The horizontal axis is the time (":mm:ss" format) and the vertical axis represents a set of states in the system.

Figure 5.20: Event and state change timeline with level-triggered filters

regardless of the event detection mode; they are directly generated by the SAFECASS, not by filters. The system designer should consider these characteristics and potential state differences when designing the system and filters.

5.9 Discussion

In this chapter, we have presented the Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS) that provides a run-time environment for the Generic Component Model (GCM). Starting with the four design requirements that consider domain characteristics, we designed a layered architecture, called the SAFECASS-based architecture. The SAFECASS-based architecture allows SAFECASS to be independent from the component framework layer, while providing reusable services for the framework. To validate our approach, we used the *cisst* component-based framework and implemented the *cisst* extension for SAFECASS. The key design concepts that underpin the overall design and architecture are *separation* and *framework independence*.

The design of SAFECASS tried to achieve the design requirements throughout the overall design process. In accordance with the design rationale of GCM, SAFECASS is reusable for different component frameworks, and the system status is managed in an explicit and structured manner based on the GCM semantics (**REQ. 1**). We presented two approaches to reusability: the framework independence and the safety design decomposition. The framework independence enables reuse of SAFECASS across different frameworks. This is achieved by the SAFECASS-based architecture that loosely couples the SAFECASS layer with the framework layer via the generic APIs of SAFECASS. The safety design decomposition splits safety features into generic, reusable safety mechanisms and configurable safety specifications. This decomposition is instrumental in achieving flexibility and reusability

(**REQ. 2**). Particularly, the use of JSON makes safety specifications expressive and flexible, while keeping safety mechanisms generic and customizable. Composing safety features as a combination of safety mechanisms and safety specifications allows us to reuse the mechanisms across systems and applications. The separation between safety mechanisms and safety specifications also facilitates the testing process by making subsystems, modules, and even safety specifications available for individual testing and verification processes (**REQ. 3**). Furthermore, because safety specifications are extracted and managed separately in the JSON format, it is possible to trace any change to safety specifications by tracking changes through, for example, the version control system (e.g., svn, git). The safety mechanisms are also implemented in the code and are typically maintained by the version control system. Thus, the design of safety features as a whole, i.e., the combination of the safety mechanism and the safety specification, becomes traceable. There is another aspect of traceability, which is tracing from requirements to specification, implementation, and to testing. Although the current design of SAFECASS does not directly address this aspect, the separation between mechanisms and specifications could be a useful element to make traceable connections between requirements, specifications, implementation, and testing (**REQ. 4**).

One important design consideration that fundamentally affected the architecture of SAFECASS is the degree of SAFECASS's dependence on the services that the framework provides. At one end of the spectrum, where the current design of SAFECASS lies, SAFECASS attempts to minimize its reliance on the framework services, such as interprocess communication (IPC) and event management (e.g., event generation/dispatch/

handling), by implementing equivalent but independent services as part of SAFECASS. Although this design requires significant implementation overhead on the SAFECASS side, SAFECASS can rely on its own services – once the implementation is in place – no matter which component framework is used. Also, use of a separate IPC infrastructure in SAFECASS improves data communication redundancy. Furthermore, such a self-contained design allows system designers, who have a decent understanding of the thread execution model of the framework, to focus on the framework side when integrating SAFECASS with existing systems. At the other end of the spectrum, SAFECASS could maximally use the framework services. Compared to the aforementioned case, this design requires less features to be implemented on the SAFECASS side and leads to tighter and more seamless integration of SAFECASS within the framework. However, this design loses the benefits of framework independence and increased data communication redundancy. Also, this design complicates the overall design if different component frameworks are simultaneously used in the same system. Furthermore, because SAFECASS has to rely on a thread execution model of the framework, the system designer has to have a thorough understanding of the thread execution model of each framework to resolve any potential conflicts among different thread execution models. Considering these two design options, we chose the first option, which has minimal dependency on the framework services.

One assumption that the current design of SAFECASS relies on is that the underlying network middleware for IPC is *ideal*; the middleware guarantees timely, correct, and reliable data delivery across networks. Although this is not a practical assumption, those issues have been extensively studied in other disciplines, such as dependable computing and distributed computing domains, and thus are beyond the scope of this work. Of course, it is possible to adopt techniques and design methods from those domains and to apply them to the design of the SAFECASS IPC infrastructure.

Another assumption that SAFECASS makes is that it is sufficient to make a system *fail-safe*, rather than *fault tolerant*, in accordance with the safety design rationale described by Kazanzides (2009).[33] In terms of fault tolerance, the current design of SAFECASS has a *single point of failure*. Currently, instances of `SC::Coordinator` in each process are the only objects that maintain key information of the system, such as states and outstanding events. Because SAFECASS does not offer any mechanism to improve data resiliency or fault tolerance, those key data are prone to critical errors (e.g., process crashes) and are irrecoverable. Although fault tolerance is generally desirable, its benefits come with increased design complexity and implementation overhead. Like the IPC infrastructure, fault tolerance is an area that has been extensively investigated in other domains, as reviewed in Sec. 2.1.1.3, and there exist various, established techniques (e.g., Fig. 2.3). Our goal of this chapter is to prove that it is possible to use the GCM at run-time by actually building a run-time environment for the GCM, considering the domain characteristics of medical robotics.

5.10 Conclusions

This chapter presented the design and architecture of Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS), which provides a run-time environment for the Generic Component Model (GCM). Starting from the design requirements that consider the domain characteristics, we described our approaches, and proposed our safety-oriented layered architecture called the SAFECASS-based architecture. We also presented the detailed design and implementation of SAFECASS in terms of its essential elements, i.e., states, events, filters, and the coordinator. Throughout this chapter, we showed that it is possible to build a run-time environment for the GCM with the four design requirements that consider domain characteristics: (1) conformity to the GCM (particularly, component model independence and explicit state management), (2) flexibility and reusability, (3) testability, and (4) traceability.

Recently, safety is getting more attention not only within medical robotics, but also in the general robotics domain. Although there exists prior work outside robotics that proposed framework-based approaches to safety, it may not be effective to directly apply them to medical robot systems, mainly due to differences in the domain characteristics. In robotics, there is a wide variety of component-based software frameworks that facilitate the development of robot systems. However, a safety-oriented system architecture has not yet received much attention. Within medical robotics, numerous prior work explored the safety issues of medical robot systems considering domain characteristics. But, we found no prior work that (1) presents a framework-based approach to safety, (2) provides a run-time software environment, based on a safety-oriented architecture, that aims to facilitate research and development of safety systems for medical robots, and (3) adopts component-based software engineering as the programming model.

One possible direction for future work includes applying SAFECASS to other prominent robot software frameworks, such as OROCOS or ROS. This will help us to further refine the design of SAFECASS and facilitate safety research by allowing researchers to use different frameworks with the common foundational GCM semantics. In the longer term, SAFECASS aims to establish a run-time software environment for safety research and development in medical robotics, and possibly in robotics as well. Within such a software environment, SAFECASS will facilitate the development of safe medical robot systems in accordance with the GCM semantics. A collection of reusable mechanisms and configurable specifications for application-specific requirements will enable the accumulation of safety knowledge and experiences across various applications and systems. In this way, system designers will be able to share their expertise in a structured manner, and benefit from those shared experiences in the form of SAFECASS artifacts. In the longer term, SAFECASS is expected to facilitate the testing and certification of medical robot systems.

5.11 Contributions

The contributions presented in this chapter are as follows:

1. Architecture: SAFECASS-based Architecture

We proposed a safety-oriented, layered architecture for component-based robot systems, called the Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)-based architecture. This architecture adopts component-based software engineering (CBSE) as the programming model. The component framework-independence of the architecture improves reusability and maintainability of the system, and allows the heterogeneous use of component-based frameworks within the same system. In addition, the decomposition of safety features makes the design of safety features more explicit, reusable, testable, and traceable within this architecture. To the best of our knowledge, no prior work in medical robotics presented a safety-oriented architecture that (1) adopts CBSE as the programming model, and (2) considers the domain characteristics in a structured, systematic manner.

2. Implementation: Run-time environment for the Generic Component Model

We designed and implemented the SAFECASS that provides a run-time software environment for the Generic Component Model (GCM) using the *cisst* component-based framework. The SAFECASS inherits the characteristics and benefits of the SAFECASS-based architecture, as summarized above. The SAFECASS allows system designers to design and build safety systems of medical robots in accordance with the GCM, thereby facilitating research and development of safety systems for medical robots.

3. Framework extensions for *cisst* to support SAFECASS

We developed the *cisst* framework extension that allows SAFECASS to be used within *cisst*. As part of the extension, we updated the design of two existing safety features (thread overrun and thread exception) and added a new feature (command queue full) that can be used to improve the robustness of data exchange between components.

Chapter 6

Case Studies: ROBODOC® and Robo-ELF

6.1 Introduction

The previous chapter presented the SAFECASS architecture and the design of the Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS) that provides a run-time environment for the Generic Component Model (GCM). Considering the domain characteristics, we defined the four design requirements of the SAFECASS, and presented the two key concepts of our approaches: the framework independence and the decomposition of safety features. We also showed how the SAFECASS can be used with component-based software frameworks, using the *cisst* component-based framework as an example.

CHAPTER 6. CASE STUDY: ROBODOC® AND ROBO-ELF

Our contention of the previous chapter is that the design and implementation of the SAFECASS meet its four design requirements, thereby facilitating the research and development of safety features of medical robot systems. This chapter attempts to prove this hypothesis by (1) applying the SAFECASS and the state-based semantics of the GCM to existing surgical robot systems and (2) empirically comparing the original design and the SAFECASS-based design. For this validation, we use two medical robot systems: the research ROBODOC® System for orthopaedic surgery, and the Robotic Endo-Laryngeal Flexible (Robo-ELF) Scope System for minimally invasive laryngeal surgery. We have full control over the entire source code of these systems, which use *cisst* as their underlying component-based framework.

The following sections (Secs. 6.2 and 6.3) describe more details of these two robot systems. For each system, we first present a brief introduction to the system, including its design and architecture, and summarize a list of its safety features. Then, we illustrate how the proposed methods can be applied to the system to improve the design of safety features, while maximally preserving the original design and implementation of the safety features. Sec. 6.4 reviews these case studies and discusses how the SAFECASS meets its four design requirements, based on our experience. Finally, Sec. 6.5 concludes this chapter with a remark on future work, and Sec. 6.6 presents a list of contributions described in this chapter.

6.2 Case 1: ROBODOC® System for Orthopaedic Surgery

As part of a collaborative research project with the manufacturer, we have one research ROBODOC system installed at JHU and have full access to the source code of the original product software. The commercial ROBODOC system (THINK Surgical, Inc., Fremont, CA, USA) has a solid set of safety features that have obtained FDA approval and EU CE marking, and has been in clinical use since 1992. Additionally, the medical robotics literature provides a relatively large number of academic publications on the system, which cover various aspects of the system, including hardware and software architectures, safety designs, and system design rationale. The authenticity and open accessibility of the design of safety features provide us with a solid foundation for our empirical, comparative evaluation.

We first present a brief history and introduction of the ROBODOC system and summarize a list of its safety features based on the academic publications (Sec. 6.2.1). Next in Sec. 6.2.2, we describe how we change the monolithic architecture of the commercial system to the framework-based architecture for the research ROBODOC system. During this change, we use the code that was actually used for the *commercial* ROBODOC system. Although the commercial system and the research system have different architectures, we maximally preserve the original design and implementation of the safety features. Then, we apply the SAFECASS to this research system, changing its architecture from the framework-based to the SAFECASS-based architecture (Sec. 6.2.3). As part of this change, we consider the framework layer and the application layer separately and perform code/design refactoring of safety features in the application layer. Throughout the SAFECASS application process, the benefits and design considerations of the SAFECASS-based approaches are highlighted, from the system designer's perspective, with discussion of limitations and applicability.

6.2.1 Background

This section briefly describes the history of the ROBODOC system and presents a set of safety features in the literature. The list of the literature that we reviewed includes Taylor *et al.* (1990,[11] 1991,[4] 1994,[130] 1996[227]), Kazanzides *et al.* (1992,[5,107] 1993,[136] 1995,[137] 1996,[228] 1999[138]), Mittelstadt *et al.* (1993,[140] 1996[229]), and Cain *et al.* (1993[139]).

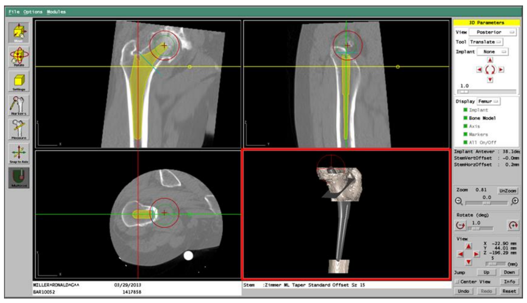
6.2.1.1 Overview of the ROBODOC System

The ROBODOC® system has been developed to increase the accuracy and efficacy of surgical procedures such as cementless Total Hip Replacement (THR) surgery and Total Knee Replacement (TKR) surgery by enabling surgeons to precisely specify the desired prosthesis placement in a preoperative CT scan of the patient and then use the robotic system to accurately machine the bone to achieve that plan.

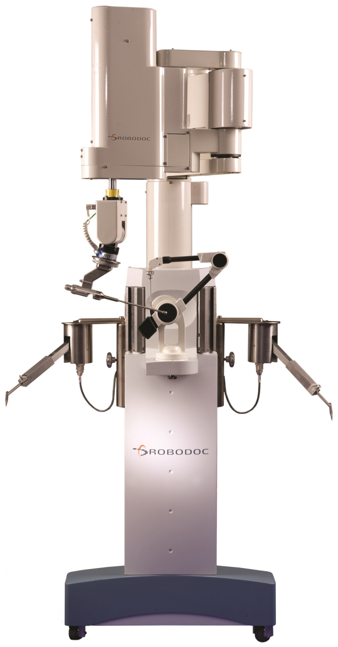
It began as a joint project between the University of California, Davis and IBM Research. The canine system (*alpha prototype*)[4,5,107,130,227] was developed for canine THR in a well-controlled, supervised research environment, i.e., under the active supervision of the developing engineers. It was first clinically used in 1990 for canine patients at a veterinary hospital in Sacramento, California and performed 26 canine surgeries.[229] This alpha prototype went through fundamental changes and improvements[229] to evolve into a *beta (investigational) medical device* that was operated by surgical teams–without engineers–for human clinical trials at multiple clinical sites. The beta version of ROBODOC was the subject of an FDA-authorized, multi-center clinical trial in the United States, and performed over 200 sugeries at a hospital in Frankfurt, Germany.[228] It was then further developed as a *commercial-level product* to be used in Europe and this required an architectural change from a centralized architecture to a distributed architecture.[228]

Fig. 6.1 shows the ROBODOC System. The two main components of the ROBODOC System are the ORTHODOC™ Preoperative Planning System (ORTHODOC)[230] and the ROBODOC Surgical Assistant (ROBODOC). ORTHODOC allows the surgeon to develop a preoperative plan that ROBODOC can execute. The two inputs to ORTHODOC are a Computed Tomography (CT) scan of the patient's femur and a set of implant models based on data from the implant manufacturers. Using implant models and a three-dimensional model of the femur constructed from the CT data, the surgeon visually determines an appropriate implant model with its precise location. Once the plan is finalized, the preoperative plan is recorded and transferred to ROBODOC via a transfer medium (tape or CD).

ROBODOC reads the preoperative plan from the transfer medium and machines a cavity for the implant in the femur according to the plan. To precisely execute the plan, it is required to register the patient's femur in the preoperative plan with the intraoperative physical reality



(a) ORTHODOC: Preoperative Planning



(b) ROBODOC: Surgical Robot

Figure 6.1: ROBODOC System: The two main components are the ORTHODOC and the ROBODOC (Courtesy: THINK Surgical Inc.).

(i.e., the robot's workspace coordinates). Calibration of the robot kinematic parameters and the cutting tool's dimensional parameters also play a vital role in achieving high dimensional accuracy. Based on clinical input, the specification of the robot's placement accuracy (deviation from the preoperatively planned position) is less than 1.0 mm, and that of the dimensional accuracy (deviation of the machined shape from its ideal dimensions) is less than ±0.4 mm on a cross-section (or ±0.2 mm on each side).[138]

6.2.1.2 Safety Features of ROBODOC

The principal safety requirements of ROBODOC were defined by a surgeon, who was a user of the system, as follows:[4, 11]

- *The robot should never "run away"*: No single-mode hardware failure or system error should cause the application software to lose control of the robot motions.
- *The robot should never exert excessive force on the patient*: Any cutting force substantially more than needed means something may be wrong and the robot better stop its current motion.
- *The robot's cutter should stay within a pre-specified positional envelope relative to the volume being cut*: A systematic positional shift in the placement or shape of the hole should be prevented.
- *The surgeon must be in charge at all times:* The system must provide the surgeon with timely information about its current status and the surgeon must be able to stop motions at any time.

A set of safety features distributed throughout the system were designed to achieve these principle safety requirements. The following presents a *partial* list of the safety features of ROBODOC based only on the *published, academic* literature. At the end of each safety feature is the class of each entry in terms of the category of domain-specific safety features presented in Table 2.1.

1. **System integrity monitoring (with redundant sensors)**: The real-time control loop periodically monitors system integrity, such as tolerance checking of primary and redundant encoders at each joint, with the capability to turn off the robot arm.

   » Class 3 (redundant sensing and/or computation)
2. **Use of state variables for error handling**: State variables are defined to represent

application-specific procedural flow and are used for error or exception handling.

» Class 8 (software engineering techniques)

3. **Dedicated processor for the safety system**: The safety system runs on a separate processor to isolate it from errors in other subsystems and has a direct hardware interface to power off the system.

   » Class 3 (redundant sensing and/or computation)

4. **Force/torque checks with two thresholds**: Force and torque feedback from the force sensor are monitored against two thresholds, PAUSE (1.5 kg-f) and STOP (3.0 kg-f). PAUSE halts all robot motion and turns off the cutter, whereas STOP removes power from both the robot and the cutter.

   » Class 5 (environment sensing)

5. **Speed limiter (low speed)**: The low-level software implements the speed limiter.[136] It operates on the output of the trajectory generation functions and limits the robot's speed and torque. The rationale of this safety feature is that the surgical staff can stop the robot before any hazard occurs if the robot is slow and weak.

   » Class 6 (software constraints)

6. **Safety volume (dynamic constraints)**: During cutting, the software verifies that the cutter tip is within a pre-planned safety volume, with a 3 mm margin for error. The safety volume is derived from the prosthesis geometry, but is independent from the file containing the cutting paths.

   » Class 6 (software constraints)

7. **Notification of exceptions to application**: Any reflex action initiated by a safety system leads to an exception to the application which interrupts the normal procedural flow and invokes a handler function for error recovery.

   » Class 8 (software engineering techniques)

8. **Error detection**: Data integrity checks (e.g., if there is any corruption of critical data), data rationality checks (e.g., if case-specific data are reasonable), detection of procedural errors (e.g., if surgeons make procedural errors that can compromise both safety and system performance).

   » Class 6 (software constraints)

9. **Startup diagnostics**: During start up, all safety- and performance-related components are verified to make sure that they are functioning within specified tolerances.

   » Class 7 (diagnostic tests)

10. **User oversight with emergency pause/stop**: An external custom pendant with five buttons, including emergency pause and stop, helps surgeons to interact with the system and a graphical display in the operating room provides the current status of the surgical procedure.

    » Class 4 (human-computer interface)

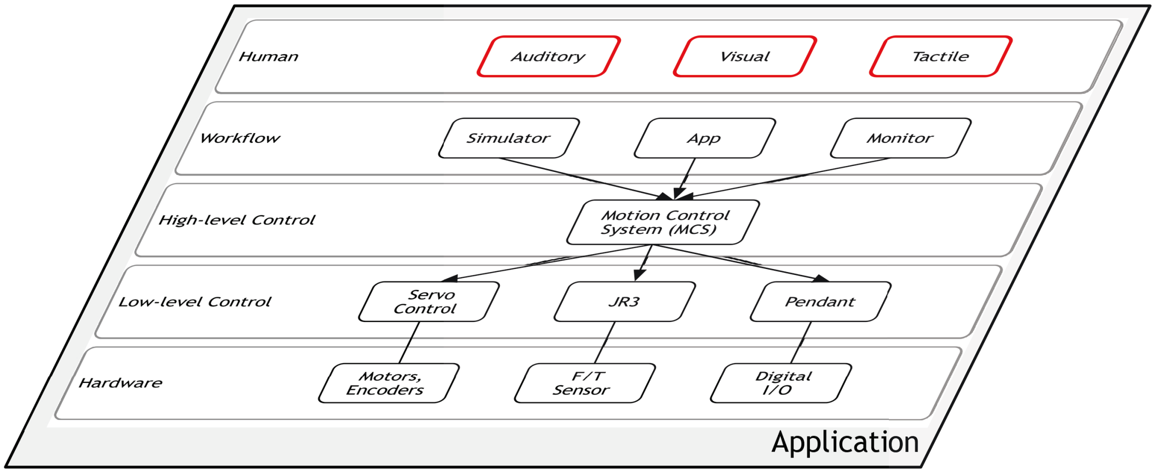


Figure 6.2: Monolithic architecture of the commercial ROBODOC system where only the application layer exists (simplified view; not all the components of the system are shown).

6.2.2 Research ROBODOC System

The architecture of the ROBODOC system has evolved [i] from a centralized architecture to a distributed architecture to address the commercial requirements, including compliance with European directives (CE marking) and improved usability and serviceability.[228] In terms of the architectural style for safety, as shown in Fig. 5.4, this system has a *monolithic* architecture. Fig. 6.2 depicts the system with its key elements, along with the five layers of the System View and the application layer.

As part of a collaborative research project with the manufacturer of the robot (THINK Surgical, Inc.), we have one research robot system installed on site with full control over the entire software. This research system has the same kinematic design and hardware architecture as the commercial system, but uses different joint (low-level) controller boards

[i]Refer to Kazanzides *et al.* (1992,[5] 1996[228]) for more detailed description on the evolution of the architecture of the ROBODOC system.

and a different operating system (QNX or Linux) on the control PC. We ported the commercial system software to the research system, which involved (1) converting the system to a component-based design using the *cisst* framework, (2) taking advantage of the portability of *cisst* to run the software on QNX or Linux, instead of DOS (used by the commercial system), and (3) developing a component to interface to the new low-level joint controllers.

The following section briefly describes how we integrated the code base of the commercial system into the component-based environment, despite inherent differences in the programming model and the architectural style.

6.2.2.1 From Monolithic to Framework-Based

An architectural change from the *monolithic* to the *framework-based* architecture begins with the introduction of a component-based framework to the system. For the research ROBODOC system, we use the *cisst* component-based framework.[49] The existing core modules are wrapped as *cisst* components or reusable C++ libraries. The common services, such as data exchange, thread execution, and event management (i.e., event generation, event distribution, event handling), are replaced by the framework services that *cisst* provides. Although this change makes the system operate in accordance with the data exchange and thread execution models of *cisst*, the design and implementation of safety features are maximally preserved throughout the change. This process is conceptually depicted in Fig. 6.3, where the framework layer is introduced below the application layer.

The software of the research ROBODOC system includes (1) the interface to the low-

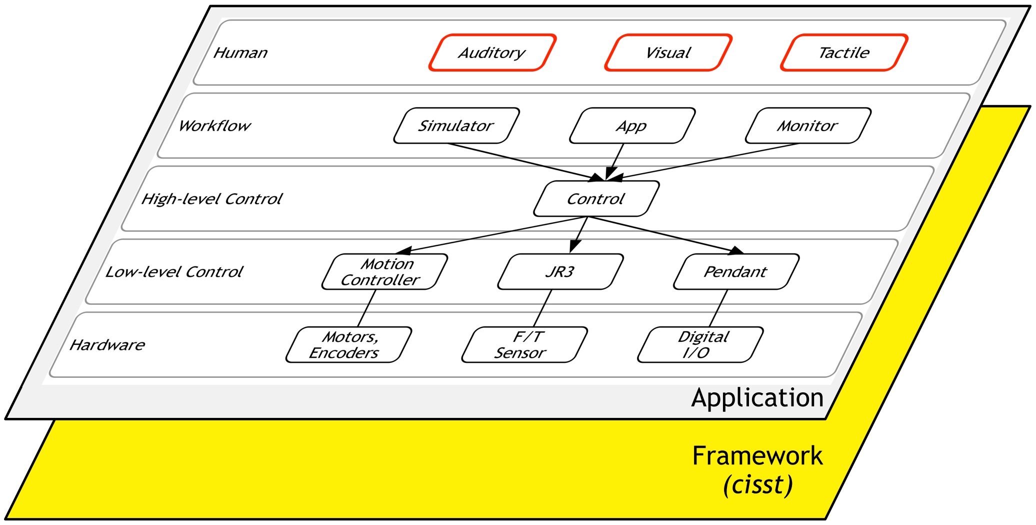


Figure 6.3: Framework-based architecture of the research ROBODOC system. The underlying framework layer (*cisst*) provides the application layer with the component-based environment and the common services, such as data exchange, event management, and thread execution.

level controller on commercial controller boards that support motor control at up to 500 Hz, (2) the high-level controller (e.g., Cartesian motion control, joint control, force control) that runs on a PC as a real-time loop, and (3) the application component that defines the procedural workflow and is executed in non-real-time on the same or a different PC, possibly with a graphical user interface (GUI).

This system is fully functional and has been used for investigating research problems. For example, we have used this system to develop new methods for the calibration of robot kinematic parameters using nonlinear optimization methods.[231] We also applied a similar approach to the calibration of bone cutting tool parameters.[232]

6.2.3 Application of SAFECASS

The research ROBODOC system is now fully functional within the component-based environment. The next step is to apply SAFECASS to this system in order to empirically evaluate the effectiveness and applicability of our SAFECASS-based approach. This process begins with extending the current architecture of the system from the *framework-based* architecture to the *SAFECASS-based* architecture, as described in Sec. 6.2.3.1. Sec. 6.2.3.2 presents how we apply SAFECASS to the system to improve the design and implementation of safety features for the application layer. Then, Sec. 6.2.3.3 illustrates how the system behaves when warnings or errors occur, using visualization of states and events based on experiment results that we collected.

6.2.3.1 From Framework-Based to SAFECASS-Based

As discussed in Sec. 5.6.1, the safety features of the research ROBODOC are still embedded in the system and tightly coupled with the other functional parts of the system, as in the commercial ROBODOC. To apply SAFECASS to the research ROBODOC system, we first introduce the SAFECASS layer to the system, as depicted in Fig. 6.4. Once SAFECASS is in place, we decompose safety features of each layer (i.e., the framework layer and the application layer) into reusable mechanisms and configurable specifications, and deploy them to the system. This necessitates code-level refactoring. On the right side of Fig. 6.4, the specifications from each layer are represented as the "SAFECASS Artifacts for

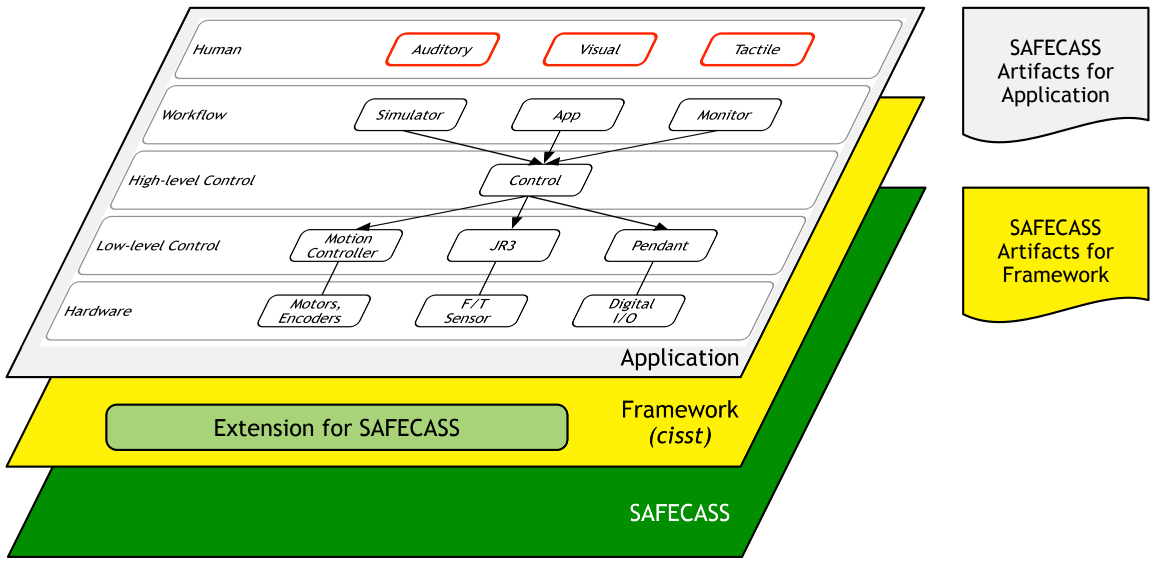


Figure 6.4: SAFECASS-based architecture of the research ROBODOC system where the SAFECASS layer is introduced to the system. This introduction requires the framework extension for the SAFECASS. The framework layer and the application layer maintain SAFECASS artifacts that define the specifications of safety features of each layer.

Application" and "SAFECASS Artifacts for Framework". Note that the SAFECASS layer is completely hidden from the application layer, which is an inherent characteristic of the layered architecture. In the following sections, we describe further details of this change by layer.

6.2.3.2 To Application: *Cutting*

In the previous chapter (Sec. 5.7), we presented three safety features that we introduced to *cisst* using SAFECASS, and described how SAFECASS can be applied to a component-based framework to deploy these features in a simple, flexible, and systematic manner. They handle *framework-specific* and *application-independent* events, and thus are applicable to any application built on top of *cisst*.

We now turn to the application layer. This layer handles *application-specific* events that have a wide range of variations in terms of the design and characteristics. Although it is difficult for SAFECASS to provide generic solutions that fit for diverse application structures, SAFECASS facilitates the design and deployment of safety features by providing APIs and a structured programming scheme. We describe this approach by applying SAFECASS to the research ROBODOC system (hereafter, ROBODOC) as an illustrative example.

Among many applications of ROBODOC, we select the *Cutting* application. Cutting was developed by the company to more easily test the machining of the cavity for an artificial implant in total hip replacement (THR) surgery. This application is a representative example of ROBODOC in that it provides significant functionality for THR surgery, including safety features and the ability to perform the entire cutting procedure based on the pre-planned cut file; primarily, it lacks the registration capabilities of the full THR application. We apply SAFECASS to Cutting and perform code/design refactoring on a subset of safety features of Cutting. In terms of the architecture of ROBODOC where applications are separated from the base technology,[5] many of these safety features are part of the base technology and thus are common to the other ROBODOC applications as well. Among various components of Cutting, our focus is on the three key components of the system that operate at different layers in terms of the System View, as shown in Fig. 6.5: (1) the `JR3` (force sensor) component of the low-level control layer, (2) the `CONTROL` component of the high-level control layer, and (3) the `Cutting` component of the workflow layer.

We first summarize the prerequisites for SAFECASS-based code/design refactoring, and

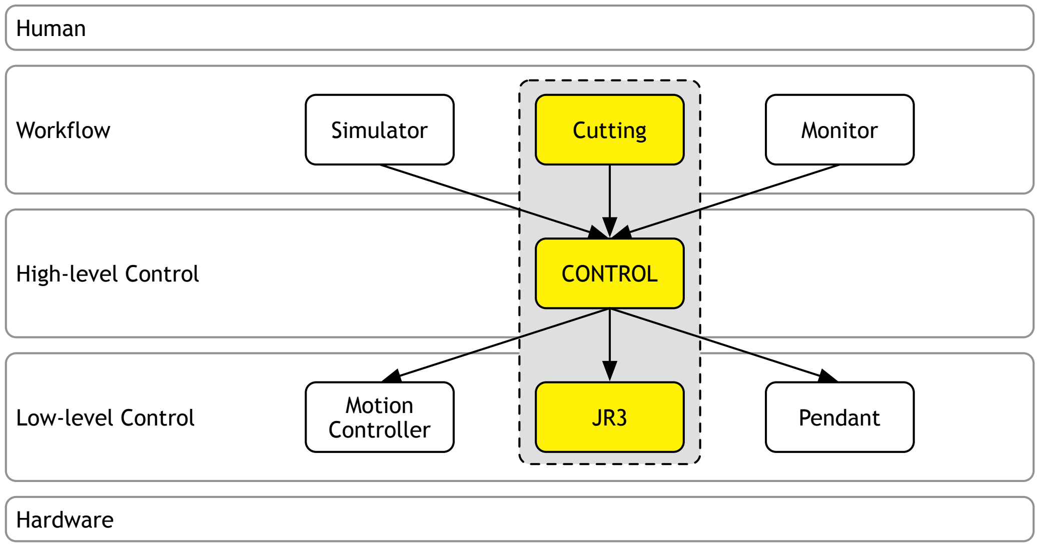


Figure 6.5: Three key components of the research ROBODOC: `Cutting`, `CONTROL`, and `JR3`. Each of these components that operate in three different layers of the system represents three typical types of components in robotic systems. The SAFECASS is applied to these three components of inherently different characteristics.

then describe more details of how SAFECASS can be applied to each component, thereby illustrating the benefits of SAFECASS in terms of flexibility, reusability, testability, and traceability.

6.2.3.2.1 Prerequisites

The three prerequisites for SAFECASS are: (1) the availability of the history buffer, (2) key data registered to the history buffer, and (3) support for deep fault injection (*optional*). As described in Sec. 5.6.5.7, *cisst* uses the state table as the history buffer. Every *cisst* component with a processing thread (i.e., components of type `mtsTask`) maintains internal state tables. The system designer registers the key variables of components to the monitoring state table by calls to `AddData()` of the `mtsStateTable` class:

```
StateTableMonitor.AddData(foo, "foo");
StateTableMonitor.AddData(bar, "bar");
```

This enables SAFECASS to access data registered to the state table. In most cases, it also enables deep fault injection because other components will call a `Read` command that accesses data from the state table in a thread-safe manner (this is the usual implementation in *cisst*).

6.2.3.2.2 *JR3*: Low-level Control

The JR3 component is a force sensor wrapper component that provides 6 degree-of-freedom force feedback (3 forces and 3 moments) for the system. Force feedback is used for various purposes, such as providing safety features (by preventing excessive force)[ii], enabling tactile search capabilities (e.g., pin/post finding during calibration and registration), and an improved human-machine interface (i.e., force control).[107]

This component periodically monitors the sensor status to detect any warning or error. If any non-normal status is detected, it sets its warning or error flags accordingly. If nothing is detected, it reads the latest force/torque feedback from the sensor and updates its local cache (state table) that would be provided for other components upon request. This is a typical implementation of a device wrapper component with straightforward error checking and error reporting as safety features. Code 6.1 shows the simplified code of the JR3 component implemented as a *cisst* component. The variables and functions that begin with a capital letter represent class member variables and functions.

[ii]As described in Sec. 2.3.1.2, the force sensor-based safety is one of the most widely used safety features in medical robotics.

Code 6.1: Simplified code of JR3 component of research ROBODOC

```
#include "JR3.h"

JR3::JR3(const std::string & name): mtsTaskPeriodic(name, 0.005) // 5 msec period
{
  // Reset variables
  // Register variables to the state table
  // Create provided interface
}

void JR3::Run(void)
{
  // Process commands and events from other components
  ProcessQueuedCommands();
  ProcessQueuedEvents();

  // Read sensor status values from sensor device
  Device->GetError(Error);
  Device->GetWarning(Warning);
  Device->GetErrorCount(ErrorCount);

  if (no_error) {
    Device->GetForceTorque(FT);
  } else {
    // Handle errors
  }
}
```

Although this typical implementation performs error checking, this design suffers from inherent limitations in terms of *testability* and *error propagation*. To dynamically test this implementation, this code requires a separate testing facility or code-level changes. For example, it is possible to simulate sensor errors by manually adding test code and test data. However, such manual testing would require an event-based mechanism (i.e., event generation, event delivery, event handling) to trigger the execution of the test code at a desired timing, and another event to disable the test code. Furthermore, it has to be recompiled whenever the test code – possibly test data as well – changes and the robot system may have to be restarted, making it a time consuming process. In addition, this test code is "embedded" in this component and thus is not reusable for other components. More importantly, this design does not support error propagation in a systematic manner.

Although it is possible to define error events for error propagation, the disadvantage with this implementation is that it would force other components connecting to this component to define event handlers for those error events. This would be an implementation burden on the other components. Furthermore, it may not be possible to enforce this requirement on other components if, for example, the JR3 component designer does not have code-level access to the other components, or design changes on other components are not allowed. To address these limitations, we apply SAFECASS to this component and perform design and code refactoring, thereby improving testability and error propagation.

The application of SAFECASS to the component is performed in two steps: (1) defining a safety specification, and (2) code structure refactoring.

1. Definition of safety specification: The starting point is to define a JSON specification file that defines GCM events, GCM filters, and GCM service state dependency information. The first step is to analyze "*what can go wrong*", i.e., failure modes of the sensor device, based on the documentation of the sensor. This can be skipped if the safety analysis has already been performed. According to the official JR3 documentation,[233] there exist three types of abnormal events: *warning*, *error*, and *error count*. Examples of these events are strain gauges near saturation (warning), strain gauges saturated (error), and communication error (error count). Thus, we define three pairs of GCM events, each pair corresponding to each abnormal event: `(/)EVT_JR3_WARNING`, `(/)EVT_JR3_ERROR`, and `(/)EVT_JR3_ERROR_COUNT`. In addition, we define another pair of GCM events, `(/)EVT_JR3_DEVICE_ACCESS_ERROR`, to represent errors that may occur if an access to the device fails due to, for example, the device driver not yet loaded into the kernel or an insufficient privilege.

The next step is to determine "*how to detect*" these events. Because JR3 represents the abnormal status as non-zero error codes, it is sufficient to monitor if there exists any non-zero sensor status value. This can be easily achieved by using the *ChangeDetect* filter, one of the basic filters of SAFECASS. Thus, we deploy three instances of this filter (of type "`FilterChangeDetect`"), each associated with the first three GCM events. In the case of `EVT_JR3_DEVICE_ACCESS_ERROR`, it is directly generated by code snippets embedded in the code, and thus no filter is defined for this event.

The implication of defining these events and filters is that we react to abnormal events by generating GCM events that correspond to the abnormal events. The complete JSON specification of safety features for the JR3 component is presented in Code B.3 in Appendix B.3.

2. Refactoring of code structure: The use of the safety configuration file defined in the previous step necessitates code structure refactoring that aims to exploit the state-dependent operational modes of the GCM (in Sec. 4.3.8) and the services that SAFECASS provides.

The *cisst* extension for SAFECASS supports the state-dependent execution semantics that allows component designers to define three different behaviors of the component via the `RunNormal()`, `RunWarning()`, and `RunError()` methods of the base component. Within *cisst*, component designers have flexibility in determining whether to override the default behavior. If a user component overrides none of these methods, the method for the default behavior, `Run()`, is executed all the time. If any of those methods is overridden, the overridden method(s) will be executed accordingly depending on the extended component state, $\hat{s}_{ext}(i)$ (i: a unique ID of a user component under consideration), that comprehensively captures all the states within the component.

The following code snippets show the original and new code structure side-by-side where the new SAFECASS-based JR3 overrides all three methods.

Before:

```
void JR3::Run(void)
{
  ProcessQueuedCommands();
  ProcessQueuedEvents();

  Device->GetError(Error);
  Device->GetWarning(Warning);
  Device->GetErrorCount(ErrorCount);

  if (no_error) {
    Device->GetForceTorque(FT);
  } else {
    // Handle error case
  }
}
```

After:

```
void JR3::UpdateStatus(void)
{
  if (!Device->GetError(Error)) {
    Coordinator->GenerateEvent("
      ➥ EVT_JR3_DEVICE_ACCESS_ERROR");
    Error = 0;
  }
  if (!Device->GetWarning(Warning)) {...}
  if (!Device->GetErrorCount(ErrorCount)) {...}
}

void JR3::RunNormal(void)
{
  ProcessQueuedCommands();
  ProcessQueuedEvents();

  if (!Device->GetForceTorque(FT)) {
    Coordinator->GenerateEvent("
      ➥ EVT_JR3_DEVICE_ACCESS_ERROR");
    FT.SetZero();
  }

  UpdateStatus();
}

#define ON_EVENT(evt_name)\
  if (e == Coordinator->GetEvent(evt_name))

void JR3::RunWarning(const SC::Event * e)
{
  ON_EVENT("EVT_JR3_WARNING") {...}
  ON_EVENT("EVT_JR3_ERROR_COUNT") {...}

  RunNormal();
}

void JR3::RunError(const SC::Event * e)
{
  ON_EVENT("EVT_JR3_DEVICE_ACCESS_ERROR") {...}
  ON_EVENT("EVT_JR3_ERROR") {...}

  ProcessQueuedCommands();
  ProcessQueuedEvents();

  UpdateStatus();
}
```

`RunNormal()` essentially implements the same functionality that the previous `Run()` method defines. However, the key difference is that `RunNormal()` can now rely on the fact that there is *no outstanding warning or error event* at the time of execution. This is possible because $\hat{s}_{ext}(i)$ has already been determined to be *Normal*. Such reliance on $\hat{s}_{ext}(i)$ simplifies the implementation by separating error checking and error handling code from the core functional code, which is in this case to read force sensor feedback (`GetForceTorque()`). However, it is still necessary to check errors that may occur with `GetForceTorque()` because such errors may change the operational state by generating `EVT_JR3_DEVICE_ACCESS_ERROR`. If this error occurs, force/torque feedback is reset as a response. At the end of the execution, the latest sensor state is retrieved from the sensor by calling `UpdateStatus()`. This call may change the operational state at the following iteration, if $\hat{s}_{ext}(i)$ becomes *Warning* or *Error*.

`RunWarning()` is executed if $\hat{s}_{ext}(i)$ is *Warning*. The component can access the information about the outstanding event, which SAFECASS passes as an argument, i.e., a const pointer to a GCM event. By checking the name and the detailed information of the outstanding event, it is possible to determine what caused a state transition to *Warning* and why. This provides the component with the ability to react differently to warning events depending on the cause of the event. Since the safety specification of the JR3 component defines two warning events, `EVT_JR3_WARNING` and `EVT_JR3_ERROR_COUNT`, two event handlers are implemented to generate more detailed logs about the warning event based on the JR3 documentation. At the end of the iteration, `RunNormal()` is called again because the JR3 component is still able to provide its service for other components.

`RunError()` is called when $\hat{s}_{ext}(i)$ is *Error*. As in `RunWarning()`, the two GCM events are handled separately by event handlers dedicated for each event, where the system designer can define event-specific *reaction* and *recovery* behaviors. One possible strategy for recovery is to reset the variables that represent the current sensor states, followed by a call to `UpdateStatus()`. Then, at the next iteration, $\hat{s}_{ext}(i)$ would remain *Error* if the problem persists; otherwise, it would become *Normal* if the cause of the error has been resolved.

Although the code length has increased after code refactoring, the additions are mostly new code snippets to enhance error detection and handling and lead to design benefits. First, the three separate `Run()` methods explicitly define the three different behaviors of the JR3 component in each state, allowing the component designers to focus on each state. In addition, part of the safety mechanisms performed by filters do not show up in the new design, thereby simplifying the code. Instead, the JSON safety specification explicitly defines a safety mechanism (i.e., a type of filter) with a set of parameters that determine the filter's specific behaviors. Once the correctness of the mechanism (i.e., an implementation of the filter) is thoroughly verified and validated, the component designer can use the filter with a reliance on the mechanism.

One reason why the JR3 component was able to benefit from the filters is because the code that may change the component state can be well isolated and consolidated into a single method, `UpdateStatus()`. However, if the component state may change in many different places of the code within a component, the design of the filters acts as a limitation due to its inability to support fine-grained timing control of the state change. This occurs when the processing logic of a component is complex, or when a component maintains application-specific state machines. The following two sections exemplify these cases and further discuss how SAFECASS handles them.

6.2.3.2.3 *CONTROL*: High-level Control

The `Control` component is the essential component that provides core functionalities for the system, such as robot control (e.g., trajectory generation, force control) and safety/integrity checks. It also provides various services for other components that enable system status monitoring, simulation, and interactive console. This component is designed to be reusable for different applications of ROBODOC in accordance with its original architecture, where it was called the Motion Control System (MCS).[5]

A large portion of the code of the component is dedicated to extensive and thorough consistency checks on the system status. At a high level, these safety features *monitor* the key variables of the component at each iteration. If any abnormality or inconsistency within the system is *detected*, the component *reacts* by executing a reflex reaction, such as stopping robot motion or powering off the motors, and then generating a system event. Depending on the cause, the system *recovers* from the situation by a pre-planned recovery plan or via human intervention (e.g., asking the human to resolve the problem). Code 6.2 shows the code structure of the `Control` component. Although this code is highly simplified for presentation purposes and only includes one safety check (for excessive force), it shows the essential flow of the component. A similar structure is also found in the literature (e.g., the AESOP robot control, Hayashibe *et al.*, 2006[234]).

Code 6.2: Simplified code of `CONTROL` component of research ROBODOC

```
#include "ControlTask.h"

ControlTask::ControlTask(const std::string & name)
  : mtsTaskPeriodic(name, 0.01) // 10 msec period
{
  // Reset variables
  // Register variables to the state table
  // Create provided interfaces
  // Create required interfaces
}

void ControlTask::Run(void)
{
  try {
    // Read feedback from joint controllers
    // Read feedback from force sensor

    // Process commands and events from other components
    ProcessQueuedCommands();
    ProcessQueuedEvents();

    // Check excessive force
    if (excessive_force) {
      // Generate E_FORCE_FREEZE, E_FORCE_EPO, or E_FORCE_HW_EPO
    }

    // Perform motion control (STARTUP, SHUTDOWN, HOME, FCOMPLY, ..)

    // Send next goal position to low-level controller
    if (no_error) {
      Servo.LoadPVT(pvt);
    }
  } catch (const McsError & e) {
    GenerateMcsEvent(e);
  }
}
```

The design of this component is much more complicated than that of the JR3 component, mainly due to its diverse logic, algorithms, and structures for robot control and various services for other components. In the current design, safety features are tightly coupled with other code that implements functional behaviors. As in the case of the JR3 component, it is difficult to perform systematic testing of safety features using this code and to identify the design of safety features in terms of the Mechanism View (Sec. 3.2.1). To address these limitations, we apply SAFECASS to this component through the two-step approach that we described in the previous section.

1. Definition of safety specification: The design and implementation of safety features in the current design are the results of safety analysis that have obtained FDA approval and EU CE Marking. Thus, they provide us a solid, authoritative baseline for evaluation of design refactoring, and also allow us to define a safety specification for this component by simply mapping a subset of the ROBODOC events to the GCM events. Among numerous ROBODOC events, we selected the following five events that tend to frequently occur in practice, based on our experience. The five events are:

- `E_FORCE_FREEZE`: Emergency pause (E-pause) when the magnitude of an instantaneous force vector exceeds the lower threshold.
- `E_FORCE_EPO`: Emergency power off (E-stop) when the magnitude of an instantaneous force vector exceeds the higher threshold.
- `E_FORCE_HW_EPO`: E-stop if any force sensor error is detected.
- `E_MANIPWR`: Motors on the manipulator are powered off.
- `E_PEND_PAUSE`: E-pause due to the human operator pressing the pendant pause button.

The names of the GCM events are set as those of the ROBODOC events with their previous prefix "`E`" replaced by "`EVT_CONTROL_MCS`". For example, the GCM event that corresponds to `E_FORCE_FREEZE` is `EVT_CONTROL_MCS_FORCE_FREEZE`. In addition, we define another GCM event, `EVT_CONTROL_MCS`, to collectively represent the rest of the ROBODOC error events outside our selection.

Compared to the JR3 component that has a simple logic with a straightforward safety feature, the `Control` component implements complex algorithms for robot motion control and interacts with numerous other components. Furthermore, a set of full-fledged safety features are already embedded in the algorithms. Although it is still possible to deploy filters to replace existing safety features, such refactoring may require changes in the logic or in the design of existing safety features. In such cases, an alternative option is to skip filtering and to directly generate events. For this purpose, SAFECASS provides APIs that allow the system designer to generate GCM events without GCM filters: `GenerateEvent()` and `BroadcastEvent()` of the `SC::Coordinator` class (see Sec. 5.6.6 for more details). These inline APIs allow us to use the SAFECASS services in accordance with the state-based semantics of the GCM, while at the same time keeping the timing of monitoring and detection mechanisms exactly the same as before. We find these APIs particularly useful for cases where safety features are distributed over lengthy code with tight coupling between functional code and safety features.

Code B.4 in Appendix B.4 presents the complete JSON specification of safety features for the `Control` component.

2. Refactoring of code structure: To enable safety features defined in the first step, the `Control` component has to be updated such that its functional part remains effectively the same as before, whereas its safety features are adapted for the state-based semantics of the GCM. As in the JR3 component, we first define the three `Run()` methods for the state-dependent execution and implement event handlers for each GCM event:

Before:

```
void ControlTask::Run(void)
{
  try {
```

After:

```
void ControlTask::RunNormal(void)
{
  try {
```

```
 4      // Read joint controllers
 5      // Read force sensor
 6
 7      ProcessQueuedCommands();
 8      ProcessQueuedEvents();
 9
10      ....
11
12    } catch (const McsError & e) {
13      GenerateMcsEvent(e);
14    }
15  }
```

```
 4      // Read joint controllers
 5      // Read force sensor without error check
 6
 7      ProcessQueuedCommands();
 8      ProcessQueuedEvents();
 9
10      ....
11
12    } catch (const McsError & e) {
13      GenerateMcsEvent(e);
14    }
15  }
16
17  void ControlTask::RunWarning(const SC::Event * e)
18  {
19    // framework warning
20    ON_EVENT("EVT_THREAD_OVERRUN") {...}
21
22    RunNormal();
23  }
24
25  void ControlTask::RunError(const SC::Event * e)
26  {
27    // framework error
28    ON_EVENT("EVT_THREAD_EXCEPTION") {..}
29
30    // error propagated from other components
31    ON_EVENT("EVT_SERVICE_FAILURE") {..}
32
33    // errors generated within this component
34    ON_EVENT("EVT_CONTROL_MCS_FORCE_FREEZE") {..}
35    ON_EVENT("EVT_CONTROL_MCS_FORCE_EPO") {..}
36    ON_EVENT("EVT_CONTROL_MCS_FORCE_HW_EPO") {..}
37    ON_EVENT("EVT_CONTROL_MCS_MANIPWR") {..}
38    ON_EVENT("EVT_CONTROL_MCS_PEND_PAUSE") {..}
39
40    ProcessQueuedCommands();
41    ProcessQueuedEvents();
42  }
```

In the SAFECASS-based design, the new `Control` component handles not only the GCM events defined by the safety specification (lines 34-38), but also other GCM events from the framework (lines 20 and 28). With the event handlers for each GCM event, this structure *explicitly* shows (1) the fact that the component designer considered the two events from the framework layer, and (2) how to handle each individual event (recall that GCM events are not handled, i.e., practically ignored, unless specific event handlers are provided). Furthermore, the new design also handles error from other components (line 31). The service failures of other components are propagated as `EVT_SERVICE_FAILURE`. If this event is propagated, the component is able to identify which component and interface failed to provide the correct services for it using the SAFECASS APIs such as `GetComponentState()` and `GetInterfaceState()`.

Another design element to note is that `RunNormal()` is called at the end of `RunWarning()` so that the component is able to continue providing its services without disruption, possibly at a degraded mode if $\hat{s}_{ext}(i)$ is *Warning*. Although not shown above, it is also possible for the component designer to call `RunNormal()` from `RunError()`. As a result, `RunNormal()` can be called for instances where $\hat{s}_{ext}(i)$ in not *Normal*. This necessitates APIs that enable the *inline* check of the current state of particular elements in the component. In *cisst*, the following APIs – wrappers of the SAFECASS APIs – are provided for this purpose as part of the extension for SAFECASS (in the `mtsTask` class):

```
SC::State::StateType GetComponentState(void) const;
SC::State::StateType GetProvidedInterfaceState(const std::string & interfaceName) const;
SC::State::StateType GetProvidedInterfaceState(const std::string & interfaceName,
                                               const SC::Event* & e) const;
SC::State::StateType GetRequiredInterfaceState(const std::string & interfaceName) const;
SC::State::StateType GetRequiredInterfaceState(const std::string & interfaceName,
                                               const SC::Event* & e) const;
```

One benefit of the new design of the `Control` component is the ability to *explicitly* and *completely* handle all state transitions of $\hat{s}_{ext}(i)$. This can be easily achieved by overriding the default state transition handlers of the *cisst* component class (`mtsTask`) as follows:

```
void ControlTask::OnNormal2Warning(const SC::Event * e) {..}
void ControlTask::OnNormal2Error(const SC::Event * e)
{
  ON_EVENT("EVT_SERVICE_FAILURE") {
    const SC::Event * e = 0;
    if (SC::State::ERROR == GetRequiredInterfaceState("ForceSensor", e)) {
      // Set force hardware flag if any error detected
      statev.force.error = 1;
    }
  }
}

void ControlTask::OnWarning2Normal(const SC::Event * e) {..}
void ControlTask::OnWarning2Error(const SC::Event * e) {..}
void ControlTask::OnError2Warning(const SC::Event * e) {..}
void ControlTask::OnError2Normal(const SC::Event * e) {..}
```

6.2.3.2.4 *Cutting*: Workflow

The Cutting component, a component in the workflow layer, implements a test workflow for the THR application that provides most of the key functionality to perform THR surgery with application-specific safety features. It also provides a graphical user interface (GUI) where the surgeon can interact with the system. The component is structured using a state-based approach where the procedural flow is controlled by the state variables, called SVAR,[5] as shown below:

```
TOP_LEVEL top;

BEGIN_SVAR (task_A);

  BEGIN_SVAR (subtask_A_1);
    // code
  END_SVAR (subtask_A_1);

  BEGIN_SVAR (subtask_A_2);
    // code
  END_SVAR (subtask_A_2);

END_SVAR (task_A);

BEGIN_SVAR (task_B);
  // code
END_SVAR (task_B);
```

Briefly, each state variable is an instance of the SVAR class that internally maintains its dependencies (on the other SVAR states) and current states (i.e., reset, started, finished, skipped). The implementation includes two macros, BEGIN_SVAR and END_SVAR, and the code between these macros is executed only if a state variable or its dependencies are not finished yet. The SVAR variables are used to hierarchically arrange the procedural flow, where the top of the hierarchy is defined as a “top-level” entry point (TOP_LEVEL), which can be branched to at any time, from anywhere within the application. It is reported[5] that this structure (1) facilitates error (exception) handling, (2) is well suited for structuring a surgical workflow that has a well-defined ordering, and (3) improves the organization and readability of the code.

Code 6.3 shows the simplified code structure of the `Cutting` component. This component defines only one required interface to use services provided by the `CONTROL` component in the lower layer, such as system startup/shutdown, robot homing, execution of motion primitives, and system status checks.

Code 6.3: Simplified code of `Cutting` component of research ROBODOC

```
#include "cuttingTask.h"

CuttingTask::CuttingTask(const std::string & name): mtsTaskContinuous(name)
{
  // Create required interface
}

void CuttingTask::Run(void)
{
  bool finished = false;

  TOP_LEVEL maintop;

  while (!finished) {
    try {
      BEGIN_SVAR (leg_setup);
      // initialize safety volume
      END_SVAR (leg_setup);

      BEGIN_SVAR (check_workspace);
      if (CheckWorkspace() == false) {
        // error handling
        quit_top();
      }
      END_SVAR (check_workspace);

      BEGIN_SVAR (done_cutting);
      // perform cutting
      END_SVAR (done_cutting);

      finished = true;
    } catch () {
      quit_top();
    }
  }
}

bool CuttingTask::CheckWorkspace()
{
  bool within_workspace = false;

  if (..) {
    within_workspace = true;
  } else {
    within_workspace = false;
  }

  return within_workspace;
}
```

Compared to the JR3 and CONTROL components, this component has distinctive characteristics. Its SVAR-based code structure maintains its own application-specific states to control the procedural workflow, and provides structured error handling and error recovery mechanisms. These mechanisms override the thread execution model of the framework and execute the thread based on the state variables. Because of these characteristics, the goal of design refactoring is not to modify its current design. Rather, the goal is to "augment" the component with the state-based semantics of the GCM, while maximally preserving its original design that has empirically proven and validated benefits.

As in the other components, the design refactoring process consists of two steps:

1. Definition of safety specification: Among various application-specific error events, we select the *outside workspace error* as an example. This error occurs if any part of the pre-planned bone cutting path is outside the robot's workspace. An error recovery plan dedicated to this particular error is also defined (i.e., ask the human operator to move the cutter to a different position). The JSON specification names this event `EVT_Cutting_OUTSIDE_WORKSPACE` in accordance with the GCM event naming convention.

The execution of the `Cutting` component is not periodic; it is controlled by application-specific state variables. Because the GCM filters should be executed periodically, this characteristic makes the filter-based event generation mechanism not suitable for this component. Thus, we generate this event using the SAFECASS API (`GenerateEvent()`), rather than via filters. For this reason, the specification of the `Cutting` component does not define any GCM filter.

Code B.5 in Appendix B.5 shows the complete JSON specification of safety features for the `Cutting` component.

2. Refactoring of code structure: Since we preserve both the SVAR-based structure of the `Cutting` component and the original procedural workflow, the goal of code refactoring is to represent ROBODOC events in terms of the GCM events. Specifically, this refers to (1) determining the timing to generate the onset and completion events considering the procedural workflow, and (2) adding code snippets accordingly to generate those events.

In the case of `EVT_Cutting_OUTSIDE_WORKSPACE`, the onset event is generated inside the `CheckWorkspace()` method when any part of the cut path is determined to be outside the robot's workspace. The pre-defined reaction to this error is to call `quit_top()` that branches to `TOP_LEVEL` (other detailed reactions are omitted for demonstration purposes). Thus, we generate the completion event right before the call to `quit_top()`. The code snippets that generate the onset and completion events are shown in lines 41 and 20 on the right side of the code below:

Before: *After*:

```
 1 void CuttingTask::Run(void)
 2 {
 3   bool finished = false;
 4
 5   TOP_LEVEL maintop;
 6
 7   while (!finished) {
 8     try {
 9
10
11
12
13
14
15
16       ...
17
18       BEGIN_SVAR (check_workspace);
19       if (CheckWorkspace() == false) {
20         // error handling
21         quit_top();
22       }
23       END_SVAR (check_workspace);
24
25
26
27
28       ...
29
30       finished = true;
31     } catch () {
32       quit_top();
33     }
34   }
35 }
36
37 bool CuttingTask::CheckWorkspace()
38 {
39   bool within_workspace = false;
40
41   if (..) {
42     within_workspace = true;
43   } else {
44     within_workspace = false;
45   }
46
47   return within_workspace;
48 }
```

```
 1 void CuttingTask::Run(void)
 2 {
 3   bool finished = false;
 4
 5   TOP_LEVEL maintop;
 6
 7   while (!finished) {
 8     try {
 9       // error from other components
10       const SC::Event * e = 0;
11       if (SC::State::ERROR ==
             GetRequiredInterfaceState(e)) {
12         ON_EVENT("EVT_SERVICE_FAILURE") {..}
13       }
14
15       ...
16
17       BEGIN_SVAR (check_workspace);
18       if (CheckWorkspace() == false) {
19        // error handling
20        Coordinator->GenerateEvent(
              "/EVT_Cutting_OUTSIDE_WORKSPACE")
              ➥ ;
21        quit_top();
22       }
23       END_SVAR (check_workspace);
24
25       ...
26
27       finished = true;
28     } catch () {
29       quit_top();
30     }
31   }
32 }
33
34 bool CuttingTask::CheckWorkspace()
35 {
36   bool within_workspace = false;
37
38   if (..) {
39     within_workspace = true;
40   } else {
41     Coordinator->GenerateEvent(
           "EVT_Cutting_OUTSIDE_WORKSPACE");
42     within_workspace = false;
43   }
44
45   return within_workspace;
46 }
```

Another GCM event to consider is EVT_SERVICE_FAILURE that occurs when a service failure event is propagated from the Control component. This error propagation event is explicitly handled by a code snippet wrapped with the inline check of the current state of the required interface (lines 10-13 in the right side of the code).

We demonstrated how SAFECASS can be applied to the design of the Cutting component in the workflow layer while preserving the existing logic and "flow" of the component.

Even though the component is structured to define the procedural workflow and application-specific states, the flexibility of SAFECASS allows us to augment the component with the state-based semantics of the GCM, so that we capture the onset and completion events of an existing application-specific event with just a single function call. We also illustrated that SAFECASS enables the handling of error propagation from other components in an explicit and structured manner with almost no additional implementation overhead. Although we considered only one application event for demonstration purposes, the same approach and refactoring strategy can be applied to other types of application-specific events in a similar manner.

6.2.3.3 Experiment Results

In this section, we illustrate how the SAFECASS-enabled Cutting application behaves when user-defined warning or error events occur. Because the SAFECASS provides access to the key elements of the system, we can easily inject faults or errors to the system and systematically verify if the system response is correct.

To demonstrate a series of sequential error propagation throughout the system, we choose the JR3 component that runs in the lowest hierarchy of the system, as depicted in Fig. 6.5. This component monitors the sensor status information that the sensor firmware maintains, and generates events if any abnormal condition is detected. The sensor firmware provides two two-byte integers that represent the current sensor status. Each of the 16 bits of these integers is a bit field indicating the warning or error state. In C code, they are defined as follows (excerpted from the official document[233]):

```
struct  warning_bits
{
  unsigned  fx_near_sat : 1;
  unsigned  fy_near_sat : 1;
  unsigned  fz_near_sat : 1;
  unsigned  mx_near_sat : 1;
  unsigned  my_near_sat : 1;
  unsigned  mz_near_sat : 1;
  unsigned  reserved : 10;
};

struct  error_bits
{
  unsigned  fx_sat : 1;
  unsigned  fy_sat : 1;
  unsigned  fz_sat : 1;
  unsigned  mx_sat : 1;
  unsigned  my_sat : 1;
  unsigned  mz_sat : 1;
  unsigned  reserved : 4;
  unsigned  memory_error : 1;
  unsigned  sensor_change : 1;
  unsigned  system_busy : 1;
  unsigned  cal_crc_bad : 1;
  unsigned  watch_dog2 : 1;
  unsigned  watch_dog : 1;
};
```

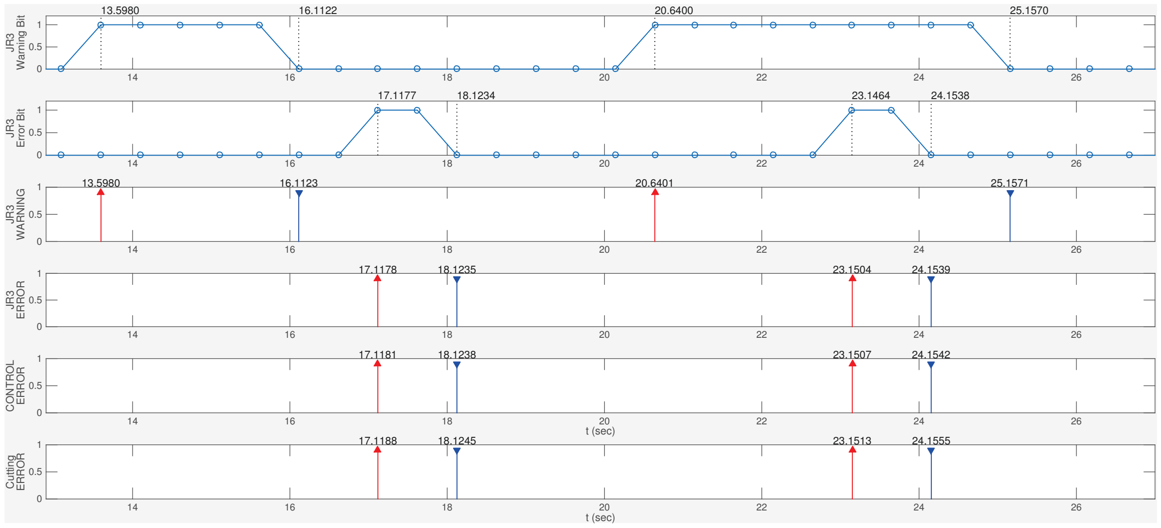
These values allow us to easily simulate arbitrary warning and error conditions simply by setting or resetting bits of the values. In this way, we can introduce to the JR3 component various warning and error events, check if events are generated in a timely manner, and verify if state changes are correct.

For presentation purposes, we increased the period of the JR3 component from 5 msec to 0.5 sec; otherwise, it would be difficult to visualize state changes. While the Cutting application was running, we performed the deep fault injection (Sec. 5.6.5) using the *console* utility (Sec. 5.6.7.1) as follows:

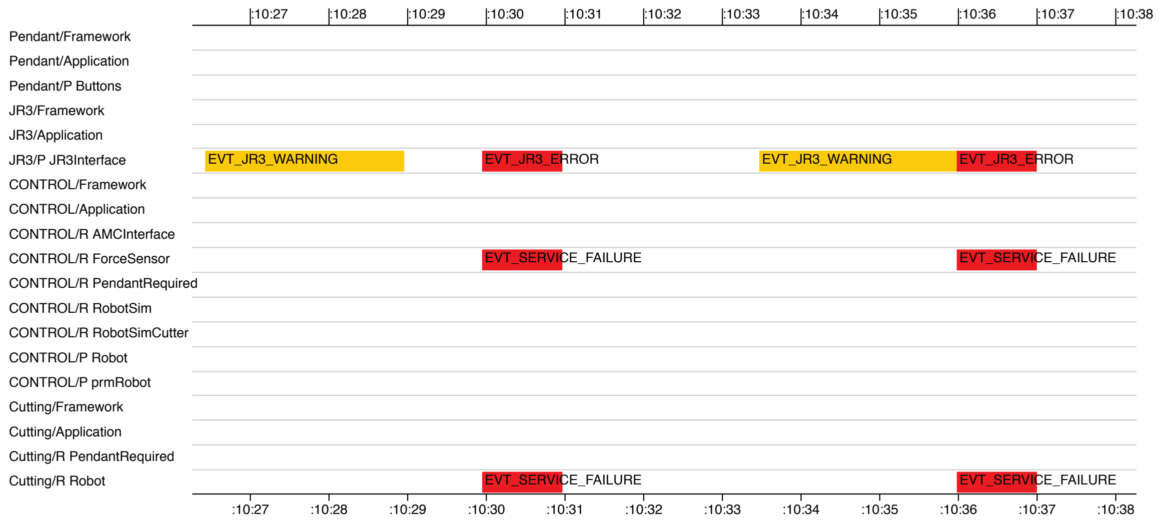
```
> filter list
Component: "CONTROL"
  [5] s_F "CONTROL" FilterThreshold "ExecTimeTotal" EVT_THREAD_OVERRUN,
      /EVT_THREAD_OVERRUN
Component: "JR3"
  [6] s_F "JR3" FilterThreshold "ExecTimeTotal" EVT_THREAD_OVERRUN, /EVT_THREAD_OVERRUN
```

```
6    [7] s_A "JR3" FilterChangeDetect "Warning" EVT_JR3_WARNING, /EVT_JR3_WARNING
7    [8] s_A "JR3" FilterChangeDetect "ErrorCount" EVT_JR3_ERROR_COUNT,
         /EVT_JR3_ERROR_COUNT
8    [9] s_A "JR3" FilterChangeDetect "Error" EVT_JR3_ERROR, /EVT_JR3_ERROR
9  Component: "SafetyMonitor"
10   [4] s_F "SafetyMonitor" FilterThreshold "ExecTimeTotal" EVT_THREAD_OVERRUN,
         /EVT_THREAD_OVERRUN
11
12 > filter dinject
13 usage: filter dinject [safety_coordinator_name] [filter_uid] [input(s)]
14
15 > filter dinject LCM 7 1 1 1 1
16 > filter dinject LCM 9 1 1
17 > filter dinject LCM 7 1 1 1 1 1 1 1 1 1
18 > filter dinject LCM 9 1 1
```

Lines 1-10 show a list of filters deployed to each component in the system. Each line displays brief information of each filter, such as the filter id (between [and]), a type of the state machine associated with the filter (`s_F`: the state machine represents the framework state, `s_A`: the state machine represents the application state), the name of filter class, and so on. In this experiment, the filter id 7 and 9 were used to generate warning and error events. By entering Line 15 followed by Line 16 with a bit of manual delay (around 0.5 second), we performed deep fault injection to simulate the individual warning and error conditions, respectively. Next, we entered Line 17, *immediately* followed by Line 18, to introduce an error event prior to the completion of the existing warning event (event prioritization). Although we used “1” as a warning and error code for simplicity, any other value that can be represented using the `warning_bits` and `error_bits` structures could be also used instead. Fig. 6.6 illustrates the results (edge-triggered filters).



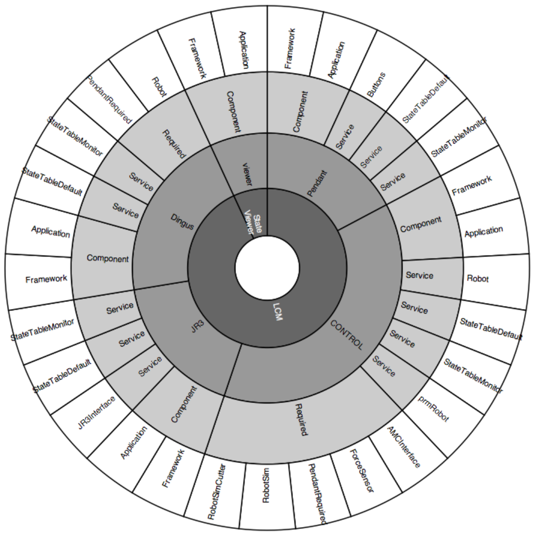
(a) Timing diagram: Red and blue arrows represent onset and completion events (horizontal axis: time in seconds).



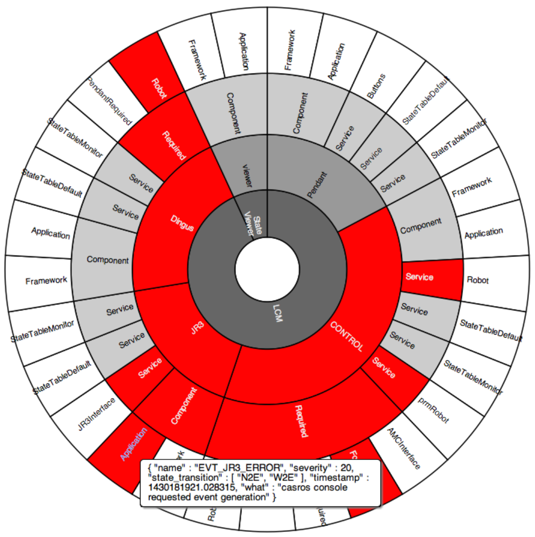
(b) Event and state diagram: Each block in yellow/red represents *Normal* and *Error* events with the name of the outstanding event. The horizontal axis is the time (":mm:ss" format) and the vertical axis represents a set of states in the system.

Figure 6.6: Timeline of events and state changes of the Cutting application

Overall, the pattern and timing of event occurrences and state changes are consistent with the results that we presented in the previous chapter (Sec. 5.8). As the error propagation



(a) Initial states



```
{ "name" : "EVT_JR3_ERROR", "severity" : 20,
"state_transition" : [ "N2E", "W2E" ], "timestamp" :
1430181921.028315, "what" : "casros console
requested event generation" }
```

(b) Occurrences of *Error*

Figure 6.7: Visualization of state changes of the Cutting application (using the *viewer* application)

mechanism of the GCM is designed, warning events were not propagated, whereas error events were successively propagated from the JR3 component to the `CONTROL` component and to the `Cutting` component. During this error propagation, error events in the component were transformed into `EVT_SERVICE_FAILURE` and were propagated across the component boundary. Also, the outstanding events and event prioritization worked as expected.

Fig. 6.7 shows the snapshots of the *viewer* (Sec. 5.6.7.2), which visualizes the state changes of the Cutting application. Initially, all the states were *Normal*, as shown in Fig. 6.7a. When `EVT_JR3_ERROR` occurred, the error was propagated to other components, changing the color of segments to red. As depicted in Fig. 6.7b, the *viewer* also showed the more detailed information about the outstanding event. In this particular case, the pop up displayed a set of attributes of `EVT_JR3_ERROR`.

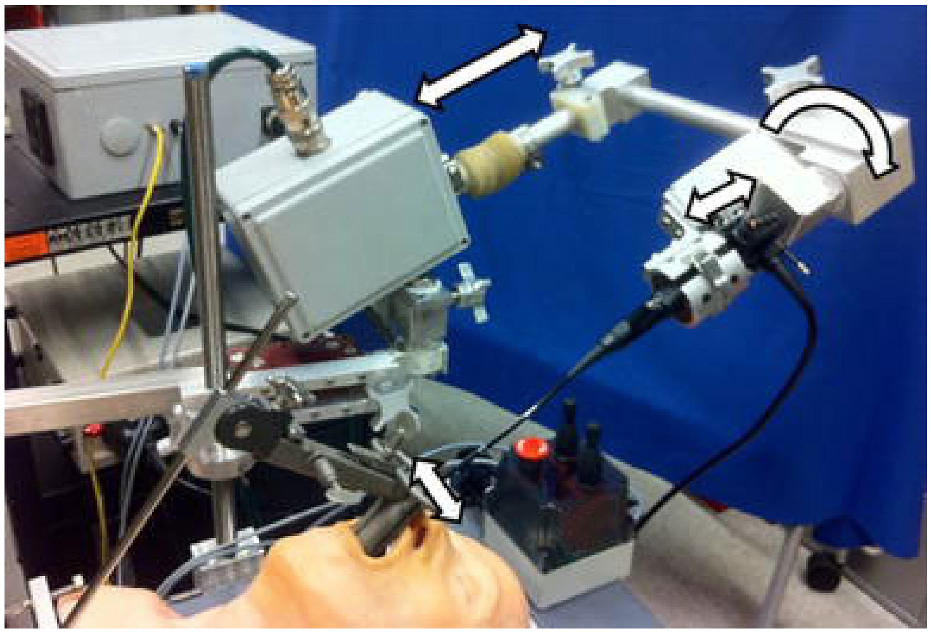


Figure 6.8: The Robotic Endo-Laryngeal Flexible (Robo-ELF) Scope System (Olds *et al.*, 2012[236])

6.3 Case 2: Robotic Endo-Laryngeal Flexible (Robo-ELF) Scope System

The Robotic EndoLaryngeal (Robo-ELF) System is a simple clinically usable robot that manipulates flexible endoscopes in laryngeal surgery (Olds *et al.*, 2011[235] and 2012[236]). This system was developed to provide the surgeon with the ability to control a flexible endoscope with only one hand and the ability to release the controls and perform bimanual surgery if necessary. As shown in Fig. 6.8, the system is comprised of four components: (1) a robot with three active and two passive joints, (2) a five degree of freedom passive positioning arm, (3) a malleable scope shaft support, and (4) a joystick controller.

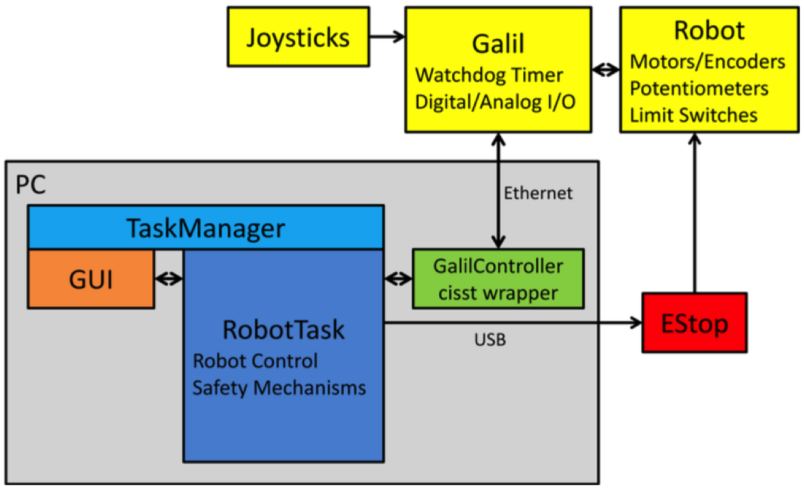


Figure 6.9: Robo-ELF: Overall system design[237]

The design of the overall system is depicted in Fig. 6.9.[237] The software system of Robo-ELF, represented as the gray box ("PC"), is built on *cisst* (Jung *et al.*, 2014[49]) and runs on Linux (Ubuntu). The features that the software provides include high-level robot motion control, safety checks, Qt-based graphical user interface (GUI), and interfaces to the low-level commercial motion controller (the Galil Motion Controller) and to an external device for emergency stop.

Fig. 6.10 shows the System View (Sec. 3.2.2) of Robo-ELF. The same color scheme used in Fig. 6.9 is also used in this figure for consistency. The software system has two components: the `RobotGUI` in the Workflow layer and the `RobotTask` in the High-level Control layer. We focus on the `RobotTask` component, which implements the safety features of the system.

The `RobotTask` component is periodically executed (at 20 Hz). At each iteration, it performs a set of safety checks and consistency checks to detect any abnormal condition

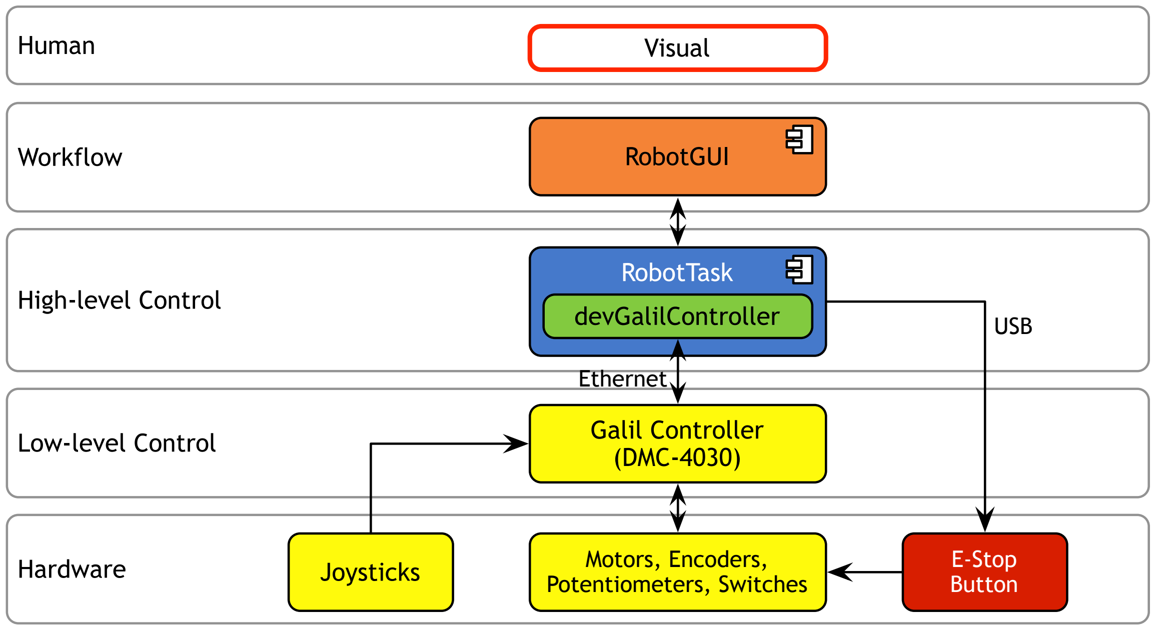


Figure 6.10: Robo-ELF: System View

in the system. If any error is detected, exceptions are thrown and caught by the pre-defined exception handlers. Otherwise, it reads inputs from the input device (joysticks) and controls the robot motions based on the inputs. The overall workflow of system safety checks implemented in the `RobotTask` component is illustrated in Fig. 6.11, where the four essential run-time safety mechanisms, i.e., the four components of the Mechanism View (Sec. 3.2.1), are tagged in the green boxes.

6.3.1 Safety Features of Robo-ELF

The system hazards of Robo-ELF have been analyzed through failure modes and effects analysis (FMEA),[238] and five safety features are implemented in the `RobotTask` component. Each safety feature is associated with a custom C++ exception class of which an instance is

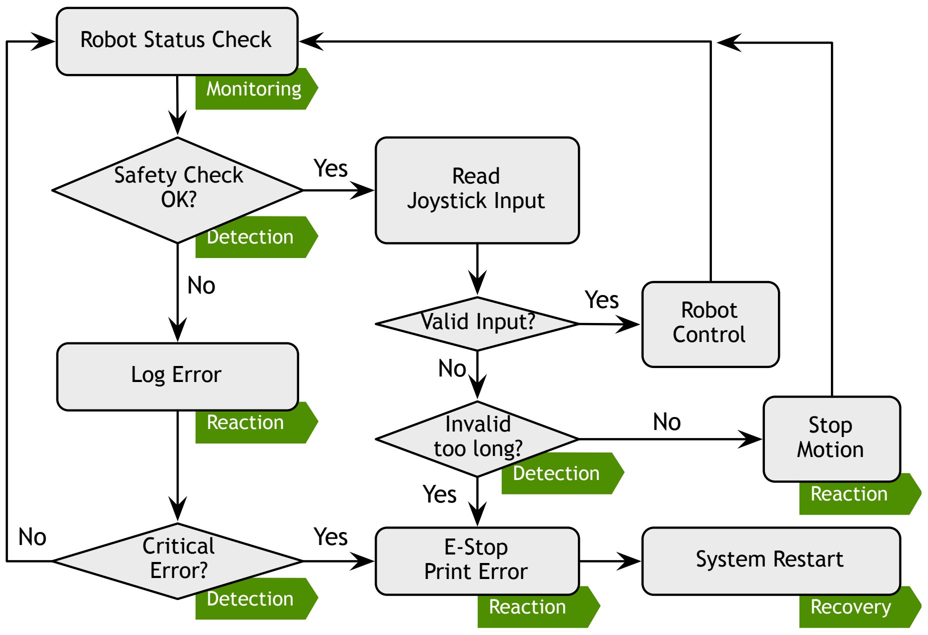


Figure 6.11: Robo-ELF: System safety check workflow (adapted from Olds *et al.*[237]). Tagged in the green boxes are the four components of the Mechanism View (i.e., monitoring, detection, reaction, and recovery).

thrown if safety checks detect any anomaly or inconsistency within the system. The five safety features and the custom exception classes associated with each safety feature are as follows:

- **Emergency Stop (E-Stop)**: A software-activated emergency stop switch that can power off the robot motors at any time.

  » Class `EStopException`

- **Encoder/Potentiometer Check**: A feature to detect mismatch between the encoder value and the potentiometer value with a threshold.

  » Class `EncoderException`

- **Galil Over-Voltage Check**: A feature to stop robot motions if too much voltage is applied to the motor.

» Class `MotionException`

- **Input Joystick Check**: A feature to verify if the joystick input reading is valid and its switches are in the valid state.

  » Class `InvalidInputException`

- **Watchdog Timer**: A watchdog to monitor the connection between the PC and the low-level motion controller. Two time limits are used: (1) 75 msec that does not stop robot motions, and (2) 125 msec that triggers E-Stop, stopping the robot.

  » Class `WatchdogException`

6.3.2 Application of SAFECASS

As in the previous case using the ROBODOC, we perform the design refactoring with two steps:

1. Definition of safety specification: We first define four pairs of GCM events with the prefix of "`EVT_RobotTask_`", each corresponding to the first four exception classes : `(/)EStop`, `(/)Encoder`, `(/)Motion`, and `(/)InvalidInput`. For the last exception class, i.e., `WatchdogException`, we define two pairs of GCM events because the two different time limits lead to two different behaviors of the system: `(/)Watchdog_Warning` and `(/)Watchdog_Error`. The former defines transitions between Normal and Warning, whereas the latter defines transitions to and from Error. Although it is possible to introduce GCM custom filters to further simplify the design of safety features, no GCM filter is defined for the `RobotTask` component to maximally preserve the current design for demonstration purposes. The `RobotTask` component has one provided interface, "ProvidesThroatRobot", that provides robot status for the GUI component. Because the service state of this provided interface becomes Error when application errors occur, we specify this information to enable error propagation.

The following code partially presents the specification (see Sec. B.1 for the complete specification):

```
{
  "component": "RobotTask",
  "event": [
    // WatchdogException (warning)
    { "name"             : "EVT_RobotTask_Watchdog_Warning",
      "severity"         : 10,
      "state_transition": [ "N2W" ]
    },
    {   "name"             : "/EVT_RobotTask_Watchdog_Warning",
      "severity"         : 10,
      "state_transition": [ "W2N" ]
    },
    // WatchdogException (error)
    {   "name"             : "EVT_RobotTask_Watchdog_Error",
      "severity"         : 10,
      "state_transition": [ "N2E", "W2E" ]
    },
    {   "name"             : "/EVT_RobotTask_Watchdog_Error",
      "severity"         : 10,
      "state_transition": [ "E2N", "W2N" ]
    },
    // Here come additional definitions of other events:
    // (/)EVT_RobotTask_EStop
    // (/)EVT_RobotTask_Encoder
    // (/)EVT_RobotTask_Motion
    // (/)EVT_RobotTask_InvalidInput
  ],
  "service" : [
    { "name" : "ProvidesThroatRobot",
      "dependency" : {
        "s_R" : [ ],
        "s_A" : true,
        "s_F" : true
      }
    }
  ]
}
```

2. Refactoring of code structure: The following code presents a side-by-side comparison of the original design of the RobotTask component (slightly simplified and modified for presentation purposes) and its new design after applying the SAFECASS to the component.

Before:

```
class robotTask: public mtsTaskPeriodic
{
  // Exception for EStop
  class EStopException {
  public:
    EStopException(const string & msg) {}



    virtual void raise() { throw *this; }
  };

  // Exception for EncoderException
  class EncoderException {
  public:
    EncoderException(const string & msg) {}



    virtual void raise() { throw *this; }
  };

  // Exceptions for other events
  class MotionException {...};
  class WatchdogException {...};
  class InvalidInputException {...};
};

void robotTask::Run(void)
{
  ProcessQueuedCommands();
  ProcessQueuedEvents();

  try {
    RunSafetyChecks();

    if (safe_condition) {
      // read user inputs
      // perform robot control
      ManualControl();
    }
  }
  catch (EStopException & e) {
    EStopExceptionHandler();


  }
  catch (EncoderException & e) {
    EncoderExceptionHandler();


  }
  catch (MotionException & e) {...}
  catch (WatchdogException & e) {...}
  catch (InvalidInputException & e) {...}
}
```

After:

```
class robotTask: public mtsTaskPeriodic
{
  // Exception for EStop
  class EStopException {
  public:
    EStopException(const string & msg) {
      Coordinator->GenerateEvent(
        "EVT_RobotTask_EStop");
    }
    virtual void raise() { throw *this; }
  };

  // Exception for EncoderException
  class EncoderException {
  public:
    EncoderException(const string & msg) {
      Coordinator->GenerateEvent(
        "EVT_RobotTask_Encoder");
    }
    virtual void raise() { throw *this; }
  };

  // Exceptions for other events
  class MotionException {...};
  class WatchdogException {...};
  class InvalidInputException {...};
};

void robotTask::RunNormal(void)
{
  ProcessQueuedCommands();
  ProcessQueuedEvents();

  try {
    RunSafetyChecks();

    if (safe_condition) {
      // read user inputs
      // perform robot control
      ManualControl();
    }
  }
  catch (EStopException & e) {
    EStopExceptionHandler();
    Coordinator->GenerateEvent(
      "/EVT_RobotTask_EStop");
  }
  catch (EncoderException & e) {
    EncoderExceptionHandler();
    Coordinator->GenerateEvent(
      "/EVT_RobotTask_Encoder");
  }
  catch (MotionException & e) {...}
  catch (WatchdogException & e) {...}
  catch (InvalidInputException & e) {...}
}

#define ON_EVENT(_name)\
  if (e == Coordinator->GetEvent(_name))

void robotTask::RunWarning(SC::Event * e)
{
```

```
63    // framework warning event
64    ON_EVENT("EVT_THREAD_OVERRUN") {...}
65
66    // application warning event
67    ON_EVENT(
68      "EVT_RobotTask_Watchdog_Warning") {
69      Coordinator->GenerateEvent(
70        "/EVT_RobotTask_Watchdog_Warning");
71    }
72
73    RunNormal();
74  }
75
76  void robotTask::RunError(SC::Event * e)
77  {
78    // application error events
79    ON_EVENT("EVT_RobotTask_EStop") {
80      EStopExceptionHandler();
81      Coordinator->GenerateEvent(
82        "/EVT_RobotTask_EStop");
83    }
84
85    ON_EVENT("EVT_RobotTask_Encoder") {
86      EncoderExceptionHandler();
87      Coordinator->GenerateEvent(
88        "/EVT_RobotTask_Encoder");
89    }
90
91    ON_EVENT(
92      "EVT_RobotTask_Watchdog_Error") {..}
93    ON_EVENT(
94      "EVT_RobotTask_Motion") {..}
95    ON_EVENT(
96      "EVT_RobotTask_InvalidInput") {..}
97
98    ProcessQueuedCommands();
99    ProcessQueuedEvents();
100 }
```

We focus on the design changes specific to Robo-ELF because the previous section about ROBODOC has already described the process and rationale of design refactoring.

We first modify the constructors of the five custom exception classes so that the onset event of each exception is generated whenever any exception is thrown (Lines 7 and 17). By generating GCM events in the constructors, we minimize timing delays between error detection and event generation. The generation of onset events allows the SAFECASS to be notified of the occurrences of events and to update the system status accordingly in a timely manner (see Sec. 5.8 for more about event processing delay and state change timing based on experiments).

Next, we define the three different methods to enable the state-dependent operational modes (Sec. 4.3.8). `RunNormal()` is identical to the original `Run()` method except for the additional changes in the `catch` clauses to generate completion events after finishing error handling (Lines 45 and 50).

`RunWarning()`, defined in Line 61, handles the watchdog warning event, which is the only application-specific warning event of the `RobotTask` component. The original code handles this warning event by generating logs, followed by resetting the watchdog timer to allow recovery. Thus, `RunWarning()` simply generates the completion event and calls `RunNormal()` to continue its execution.

In addition, we define `RunError()` as in Line 76 to enable *event-based testing*. Practically, `RunError()` due to application-specific events is never called because `RunNormal()` detects and handles all application-specific error events, including the generation of completion events, and thus the component state remains only Normal or Warning, but not Error. Still, we duplicate the identical exception handlers within the `ON_EVENT` macro in `RunError()` for testing purposes.

Using the SAFECASS tools such as the *console* utility (Sec. 5.6.7.1), we can easily generate events and verify that the behavior of each exception handler is correct. The advantage of this event-based testing is that it provides us a way to test safety features without having to simulate error conditions by directly manipulating variables. Of note, there are cases where simulating and reproducing the same error scenarios are not easily achievable without a substantial amount of test code, because of the inability to use exact data with exact timing.

6.4 Discussion

In this chapter, we presented how the design of the safety features of the ROBODOC and Robo-ELF systems can be improved in terms of testability and traceability, by introducing the SAFECASS to these systems. The introduction of SAFECASS necessitated an architectural change to the SAFECASS-based architecture. After this architectural change, we demonstrated how to exploit the SAFECASS-based architecture and the services that the SAFECASS provides to implement or improve safety features. In each system, we selected a set of components of inherently different characteristics from different layers in terms of the System View, and described how the design of each component can be improved by SAFECASS.

First, we revisit the four design requirements of the SAFECASS, as defined in Sec. 5.4, and discuss how the current design and implementation of the SAFECASS meets each requirement, based on our experience with these case studies.

1. Conformity to Generic Component Model: Although *cisst* is the only framework that we used for our case studies, we confirmed that the current design and implementation of the SAFECASS can represent the application-specific operational status of the ROBODOC and Robo-ELF systems using only the minimal structural elements of the GCM (i.e., components and interfaces) without relying on particular aspects of the component model in use (i.e., the *cisst* component model). Together with the framework-level features that are associated with component model-specific structural elements, as described in Sec. 5.7, we learned that the state-based semantics of the GCM implemented by the SAFECASS enables the explicit and structured state management of the overall system by providing tools and APIs in terms of states, events, and filters.

2. Flexibility and reusability: Throughout the case studies, we confirmed that it is possible to design and implement safety features with flexibility using the SAFECASS-based architecture and the SAFECASS. We consider its flexibility in terms of the safety specification, the event generation, and the state-dependent execution, as follows:

- *Safety specification*: The ability to easily change JSON safety specifications greatly facilitated the development and debugging processes. Any parameter in the specifications of events, filters, and service state dependencies can be easily modified with no code compilation required.
- *Event generation*: The SAFECASS provides two options to generate events, by filters or by a direct call to the `SC::Coordinator::GenerateEvent()` method. Because filters are defined by JSON specifications, the *filter-based event generation* is inherently flexible and configurable. One limitation is that the timing of event generation is dependent on the framework extension. In case of *cisst*, all filters are executed and events are generated *after* the execution (i.e., one iteration) of the user code is completed. Thus, this filter-based event generation is suitable for *periodic* com-

ponents where the code for the operational status update is well isolated (e.g., the `JR3` component). Alternatively, the *event generation via APIs* is better suited for the other cases, i.e., if a component is *aperiodic*, if the operational status can change in many different places of the code (e.g., components for robot control), or if the code to update the operational status is intertwined with application-specific logic or algorithms. It should be noted that these two options for event generation are not mutually exclusive; they can be used simultaneously within the same component.

- *State-dependent execution*: Using the state query APIs, such as `GetComponentState()` and `GetInterfaceState()`, it is possible to control the flow of execution depending on any state of interest in the system. One example is the extended component state, $\hat{s}_{ext}(i)$, that considers the entire set of states of a component, such that all the states are consolidated into one state. In case of the research ROBODOC, the `JR3` and `CONTROL` components used this feature to implement the state-dependent operational modes (Sec. 4.3.8).

3. Testability: One property that we found to be particularly improved is testability. The fault injection facility of SAFECASS greatly facilitated the development process, especially for the debugging and run-time verification. Specifically, using the interactive console utility (Sec. 5.6.7.1), we were able to (1) generate any registered GCM event to directly change a state of any state machine in the system at any time, (2) inject test data into GCM filters to verify the correctness of filters (shallow fault injection), and (3) simulate the occurrences of error events by injecting a sequence of test data into the history buffer (deep fault injection).

Also, the run-time state viewer utility (Sec. 5.6.7.2) helped to visually and interactively inspect the system state. For example, the default view (Fig. 5.17a) presented the entire set of state machines in the system, and thus was useful for checking if any state machine was in the non-normal state. The pop up window that the viewer displayed when the mouse was moved over a segment (i.e., a state machine) showed the detailed information of the outstanding event of the segment, if any. This feature helped us to quickly follow the "trail" of error propagation and identify the root cause of the error.

4. Traceability: Another benefit of SAFECASS that we experienced is that safety properties become more visible, structured, and thus traceable. For example, the safety specifications explicitly declared the entire set of GCM events that can possibly occur in the system. Each individual event had an unique name (enforced by SAFECASS), which allowed us to effectively track down the part of the code that detects and handles the event (e.g., `ON_EVENT` macro with an event name). Those specifications also completely captured the entire set of parameters for GCM filters in the JSON format. We have maintained these configuration files in the version control system (git) and it is possible to track any change in any of these configuration files over time. Although we did not have access to the original FMECA table of the commercial ROBODOC system, we were able to systematically trace our safety specifications and mechanisms back to the original safety features in the commercial ROBODOC system.

Through the case studies presented in this chapter, we empirically evaluated the effectiveness and applicability of the SAFECASS by applying it to two existing systems with safety features, rather than implementing new safety features. The main reason is to be able to compare the new SAFECASS-based design of safety features with the original design. Because the safety features of the two systems provide us with a solid, authentic baseline for comparison, we were able to highlight the benefits of SAFECASS in the new design in terms of its four design requirements.

Another benefit of the use of SAFECASS is that a systematic and structured error propagation and handling mechanism becomes available to the component with virtually no additional implementation overhead, once the component-based framework supports the SAFECASS. Because SAFECASS internally handles error propagation and collectively represents the service state with either *Normal* or *Error*, handling of error propagation only necessitates handling of the `EVT_SERVICE_FAILURE` event with the detailed information of the outstanding event. This helps to improve the organization of the structure of error handling within the component.

6.5 Conclusions

In this chapter, we demonstrated in detail how the Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS) and the SAFECASS-based architecture can be applied to existing systems to improve the design of safety features. For this study, we used the *cisst* component-based framework and two surgical robot systems: a commercial surgical robot system for orthopaedic surgery, called the ROBODOC system, and a research robot system for endo-laryngeal surgery, called the Robo-ELF Scope. In case of the ROBODOC, we built a research version of the system by adapting the original code of the commercial system to the component-based environment, and applied SAFECASS to the research system. We performed design refactoring on three application-specific components with inherently different characteristics: a sensor wrapper component, a real-time robot control component, and a component with surgical workflow and graphical user interface. Similarly, we also described how the SAFECASS can be applied to the control and GUI components of the Robo-ELF. By empirically evaluating the effectiveness and applicability of SAFECASS-based approaches, these case studies have shown that the current design and implementation of SAFECASS achieve its four design requirements: (1) conformity to the Generic Component Model, (2) flexibility and reusability, (3) testability, and (4) traceability.

Although we considered one particular component framework in robotics (*cisst*), two instances of surgical robot systems (the ROBODOC and Robo-ELF systems), and two surgical application areas (orthopaedic surgery and minimally invasive endolaryngeal surgery), our belief is that the SAFECASS-based approaches are generic and flexible enough to be applicable to other component frameworks, different types of medical robot systems, and various surgical applications. In our present and future work, we aim to extend the coverage of our case studies in these three respects. This would provide us with opportunities to further refine and enhance the design and implementation of the SAFECASS, thereby experimentally validating our belief with more use cases.

6.6 Contributions

The contributions in this chapter are as follows:

1. Validation: Design and implementation of the SAFECASS

– Empirical validation of the design and implementation of the SAFECASS

Through the case studies described in this chapter, we have shown that the current design and implementation of the SAFECASS meet its four design requirements and provide a run-time environment for the Generic Component Model (GCM) in accordance with the state-based semantics of the GCM.

2. Validation: SAFECASS using the ROBODOC and Robo-ELF systems

– Empirical validation of the effectiveness and applicability of the SAFECASS-based approaches using safety features of two surgical robot systems

We empirically evaluated the effectiveness and applicability of the state-based approaches to safety by refactoring application-specific safety features of the ROBODOC and Robo-ELF systems for orthopaedic surgery and endolaryngeal surgery, respectively. Through these case studies, we confirmed that the state-based approaches are expressive and systematic enough to implement safety features that are equivalent to the original design, and improve the design of safety features in terms of testability, reusability, traceability, and flexibility.

Chapter 7

Conclusions

"We do not have the luxury of learning from experience, but must attempt to anticipate and prevent accidents before they occur."

– Safeware, N. Leveson, 1995[15]

In this dissertation, we presented our methods that reformulate safety as a visible, reusable, and systematically verifiable property of component-based robot systems. Starting from a literature review on safety in various domains, we progressively developed our methods through the concept (the Safety Design View), the model (the Generic Component Model), and the architecture (the Safety Architecture for Engineering Computer-Assisted Surgical Systems), and validated the methods using two existing medical robot systems: the ROBODOC® System for orthopaedic surgery and the Robo-ELF Scope System for minimally invasive endolaryngeal surgery.

The Safety Design View (SDV) is a conceptual framework that defines the design space of safety features of medical robot systems. This design space comprises two axes: the Mechanism View and the System View. The Mechanism View defines essential components of safety features as monitoring, detection, reaction, and recovery. The System View identifies a canonical architecture of medical robot systems that captures the system designer's decisions on the deployment of safety features. By combining these two views, the SDV (1) enables the description of safety features in a consistent and structured manner, (2) allows to collect best practices on the design of safety features, and (3) facilitates sharing of knowledge and experience on safety.

The Generic Component Model (GCM) consists of the minimal structural elements (components and interfaces) and the state-based semantics that represent the operational status of component-based robot systems in an explicit, systematic, and structured manner. The GCM is generic enough to be specialized for other component models, yet it is expressive enough to completely describe the system status without relying on a particular component model. The essential elements of the state-based semantics are the state, event, and filter. The use of the three abstract states (Normal, Warning, Error) enables the generic and consistent representation of the operational status of the system. State changes are initiated by events, and events are generated by filters. The error propagation model of the GCM systematically defines how the state of a component is affected due to errors from the other components.

The Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS) is a run-time environment for the GCM. The SAFECASS provides an implementation of both the structural elements and the state-based semantics of the GCM. The two fundamental design principles of the SAFECASS-based architecture are (1) the framework independence and (2) the decomposition of safety features into reusable mechanisms and configurable specifications. Using this architecture, we built a software framework to show that it is possible to build a run-time environment for the GCM. By providing application- and framework-independent tools, the SAFECASS aims to facilitate safety research in medical robotics and the development of safe medical robot systems in accordance with the GCM.

We presented case studies that illustrate how the proposed methods can be applied to existing systems to improve the design of safety features. For these case studies, we used the *cisst* component-based framework, a subset of the safety features of the commercial ROBODOC system, and the entire set of safety features of the Robo-ELF system. We performed design refactoring on different types of components with distinctive characteristics, which are commonly found in robot systems. They include: a sensor wrapper component, a real-time robot control component, and a surgical workflow component with graphical user interface. By empirically evaluating the effectiveness and applicability of our methods, we showed that the current design and implementation of SAFECASS achieve its design requirements.

The methods presented in this dissertation address challenges in safety research by defining a new perspective on safety of medical robots and by providing a software environment that facilitates the design and development process of component-based medical and surgical robot systems. These methods would improve reuse of prior experience and knowledge on safety and reduce engineering effort to build safe medical robot systems. Among the three research areas in medical robotics[3] – modeling and analysis, interface technology, and systems science – our methods presented in this dissertation have made contributions to the advancement of the least developed research area in medical robotics, i.e., systems science.

To further improve the current design and implementation, future work can be explored in a few directions. One obvious direction is to *expand a list of use cases*. Although the developed methods are carefully designed to be application-, framework-, and component model-independent, we have used one component-based framework (*cisst*) and two application systems (the ROBODOC for orthopaedic surgery and the Robo-ELF system for laryngeal surgery). It would help us to further refine the design and implementation of the SAFECASS if the SAFECASS is applied to other robot software frameworks, such as OROCOS or ROS, and to other medical and surgical robot systems including the da Vinci Research Kit (Kazanzides *et al.*, 2014[239]). In particular, use of the SAFECASS across different robot software frameworks would be worth investigating in that it will enable reuse and sharing of safety knowledge and experience among each framework's user communities.

Another possible direction is to develop automated run-time safety analysis tools based on the GCM and the SAFECASS. Currently, the SAFECASS provides APIs and tools that enable the introspection and manipulation of the system status in terms of the states and events (e.g., fault injection, event generation). As an extension of these features, it could be possible to automatically generate various test scenarios and to verify if the system behavior is correct. For example, given a list of events, we can generate a sequence of events at random, certain, or pre-defined times and test if the system handles those events properly. In case of multi-process systems, we can simulate disconnection events and check whether such events are gracefully handled or not.

Other areas of exploration include the application of formal methods to the state-based semantics of the GCM and the implementation of the SAFECASS. Formal methods have emerged outside robotics, and are gaining increasing attention within the robotics community, especially for safety- or mission-critical applications. By thoroughly and extensively verifying the correctness of the GCM and the SAFECASS, system designers would be able to reuse the services that the SAFECASS provides without having to verify its correctness. Likewise, the integration of international safety standards with the GCM and the SAFECASS would be another opportunity to help us to reduce engineering effort in building safe medical robot systems.

Appendix A

Overview of the *cisst* Package

This section provides a brief overview of the *cisst* package, to provide sufficient background for Chapters 5 and 6. More detailed materials are available on the *cisst* web-site (`github.com/jhu-cisst`).

***Disclaimer**: This section has been published as part of Jung* et al.*, (2014).*[49]

Component-based software engineering (CBSE) has been widely adopted within the robotics community as an effective programming model to deal with challenges in building complex robotics systems.[45,46] Similar problems are also found in the medical robotics domain. At Johns Hopkins University, we have been developing an open-source component-based framework, called the *cisst* package[i], to facilitate the development of various medical and surgical robot systems. Although the *cisst* package was originally developed for

[i]The *cisst* is an acronym for Computer-Integrated Surgical Systems and Technology, and was named after the CISST Engineering Research Center established by the National Science Foundation.

computer-assisted intervention (CAI) systems, we also use it for other robotics applications, such as space robotics.[240]

Table A.1: Fact sheet of the *cisst* package

Language	C++
License	Open source
Programming Model	Component-based Software Engineering
Supported OS (Both 32 and 64 bits)	Windows, Linux, Mac OS X Real-time Linux (RTAI, Xenomai), QNX
Application Domain	Robotics, Medical and Surgical Robotics
Language Binding	Python
Web Sites	*cisst*: http://cisst.org/cisst *SAW*: http://cisst.org/saw

A.1 The *cisst* Component-based Framework

The *cisst* package[241] is a collection of open-source, cross-platform libraries.[242] The foundational libraries include the linear algebra and spatial transformation library (*cisstVector*), the component-based framework to define, deploy, and manage components (*cisstMultiTask*), the multi-channel video acquisition, processing, and display library (*cisstStereoVision*), the standard data type library to facilitate data exchange in component-based systems (*cisstParameterTypes*), and the robot kinematics, dynamics, and control modules (*cisstRobot*).

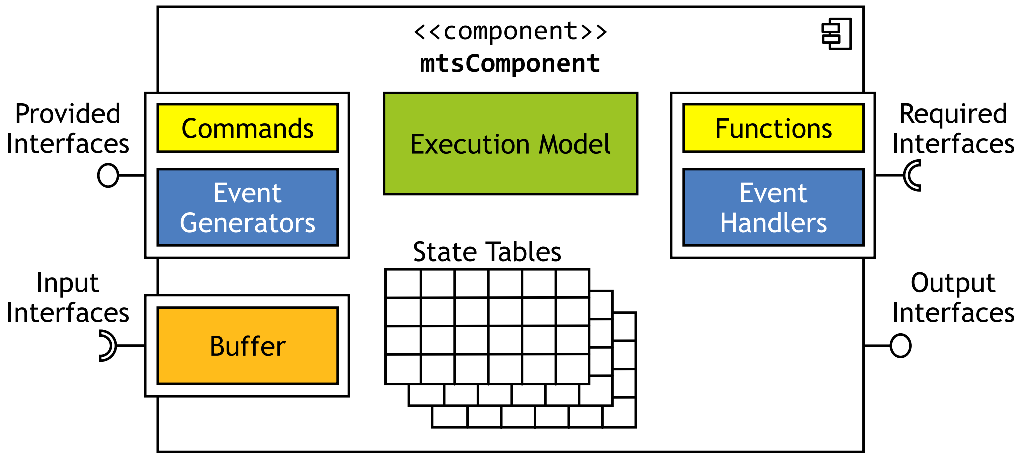


Figure A.1: Block diagram of the structure of the *cisst* component. A *cisst* component contains a list of interfaces for data exchange and time-indexed circular buffers (i.e., state tables[222]) for data archival and lock-free data retrieval.

The *cisst* libraries form the basis of the Surgical Assistant Workstation (SAW) package.[243] SAW is a collection of reusable components based on *cisst* with standardized interfaces that enable rapid prototyping of CAI systems, especially those that benefit from enhanced 3D visualization and user interaction. SAW provides diverse off-the-shelf application components that range from hardware interface components (e.g., to robots, tracking systems, haptic devices, force sensors) and software components (e.g., controller, simulator, ROS bridge). One recent addition to SAW is an open-source telerobotics research platform that is based on retired clinical da Vinci® Surgical Systems.[239] These systems use several SAW components, coupled with open-source electronics, to create a research platform that has already been replicated at more than 10 institutions (see research.intusurg.com/dvrk).

This section briefly introduces the *cisst* component-based environment that the *cisstMultiTask* library provides in three aspects: *component model*, *computation*, and *communication*.

A.2 Component: *cisst* Component Model

The *cisstMultiTask* library defines the *cisst* component model, which is implemented by the `mtsComponent` base class. Fig. A.1 and A.2 show an overview of the structural elements of a *cisst* component and its UML class diagram, respectively.

Several different types of components are derived from this base class, including `mtsTaskPeriodic`, `mtsTaskFromSignal`, `mtsTaskContinuous`, and `mtsTaskFromCallback`. All of these derived components contain a `Run` method, whereas the base class does not. Another type of derived class, the `svlFilterBase`, is defined as a base class for components of the *cisstStereoVision* library. A component contains a list of *provided interfaces*, *required interfaces*, *output interfaces*, and *input interfaces*, and the latter two interface types are relevant to the *cisstStereoVision* library.

cisst uses the Command Pattern,[204] where a service is represented as an object. 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 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 command objects provide. It may also have *event handlers* to respond to events generated by the connected component. As with the command objects, four corresponding function object classes are defined. When two interfaces are *connected* to each other, all

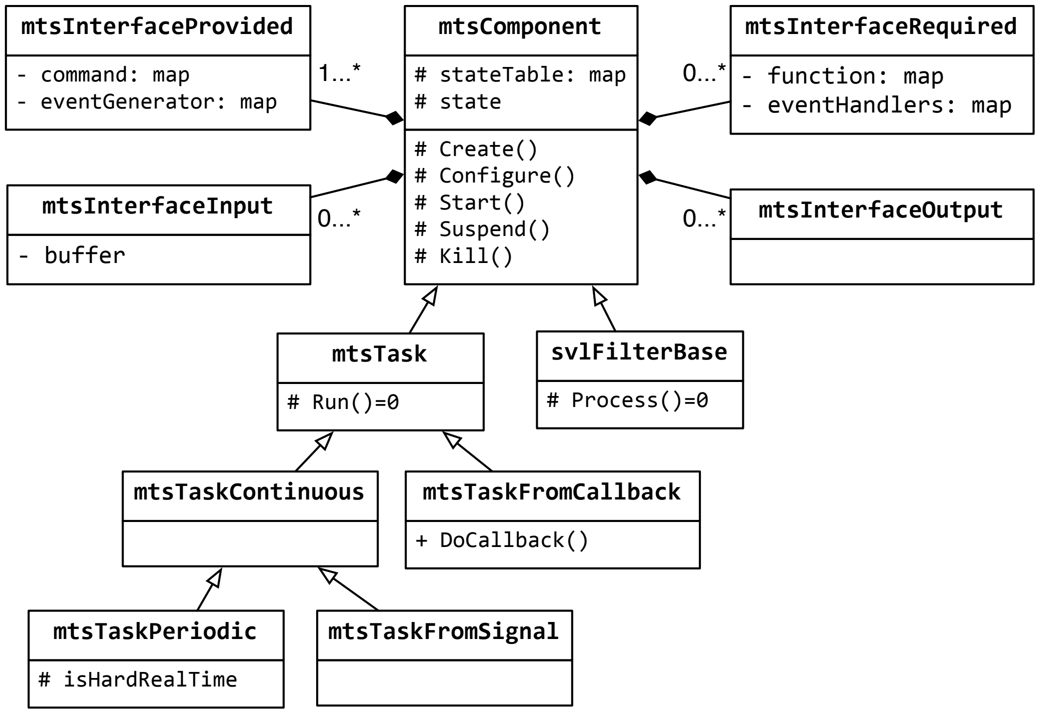


Figure A.2: UML class diagram of the *cisst* components

function 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.

A component can have multiple instances of a state table, which is a time-indexed circular buffer.[222] This table keeps the history of data registered to it, which can be used for data collection, online signal processing or for fault detection and diagnosis.[203]

The connection between components in the same process is established by the *Manager Component Client (MCC)* shown in Fig. A.3, which is unique within the process and manages all components in the same process. Each component is internally connected to the MCC when it is registered to the system. MCC provides a set of *services* via these *internal connections*: (1) *Dynamic component composition*: Users can create, configure, deploy, start, stop, resume, and connect components at run-time, (2) *Distributed lightweight logging facility*: Logs can be generated in any process in the system and be collected across a network[ii], and (3) *System information retrieval*: Users can easily access the system information, such as run-time state of components or a list of names of structural elements.

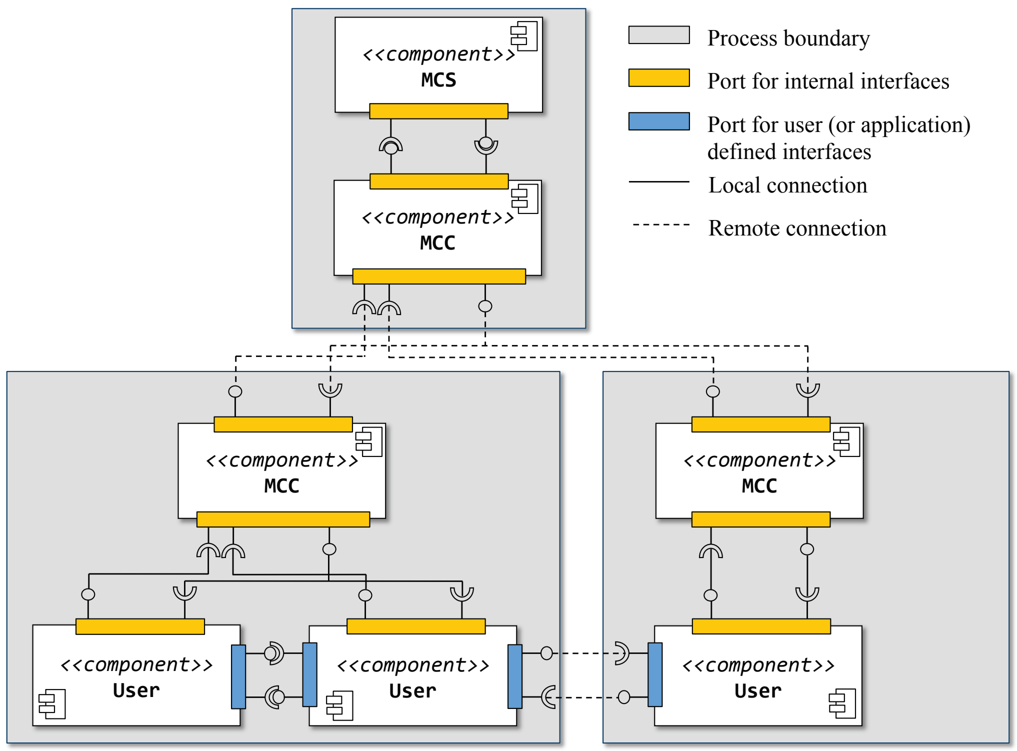


Figure A.3: Managing *cisst* components distributed over a network. The Manager Component Client component, unique within a process, connects to every component in the same process. The Manager Component Server component is unique in the entire system and connects to all the Manager Component Client components in the system. Connections among these manager components enable a set of services for user components, such as dynamic component composition and system information retrieval via the Command Pattern.

The architecture of *cisstMultiTask* has been extended to support various system config-

[ii]We found this feature to be useful, especially in real-time robot control systems where massive logging leads to heavy disk I/O and thus affects real-time performance of components.

urations, from multi-threaded scenarios to multi-process and multi-host systems.[244] This extension primarily relies on the component-based data exchange mechanism (i.e., the encapsulation of services) via a network layer. The main idea is to extend the *local* connection between components to the remote (i.e., *logical* local) connection by the introduction of “proxy” components (i.e., the Proxy Pattern[204, 245]). The proxy components mediate data exchange between the original components across the network. The network layer uses the Internet Communication Engine (ICE)[218] as middleware, but is not dependent on it because the abstraction of the network layer allows any other middleware, even a native socket, to be used. This extension introduced another type of manager, the *Manager Component Server (MCS)* as shown in Fig. A.3. The MCS manages all MCC(s) in a system and maintains system-wide information such as a list of processes, components, interfaces, and connections. With the extension, users can use two different configurations, the *standalone configuration* that supports only multi-threaded systems and the *networked configuration* that supports not only multi-threaded systems but also multi-process (and multi-host) systems. Furthermore, depending on the characteristics of components, both configurations can be deployed together to support the optimal performance of multi-threaded components in the same process as well as data exchange between different processes across a network. The conversion from the standalone configuration to the networked configuration requires minimal user code-level changes and can be done seamlessly by the MCC. Thus, users can employ the same programming model for different configurations in a flexible and consistent manner.

All the structural elements of *cisst* components are represented by a string. Together with signature and data type information, this string is used to determine if a connection can be established.

A.3 Computation: Thread Execution Model

A *cisst* component can have its own internal processing thread or use an external thread or thread pool, and can be executed with a set of different computation schemes depending on four criteria: (1) the characteristics of data that it processes ("Usage"), (2) the existence of its own internal processing loop ("Processing Loop"), (3) the thread source ("Thread Source"), and (4) the run-time behavior of computation ("Computation Scheme"). Table A.2 summarizes the *cisst* thread execution model based on these criteria.

Table A.2: Thread execution model of *cisst*

Usage	Processing Loop	Thread Source	Computation Scheme
API Wrapper	No	N/A	N/A
Image/video	Yes	External (thread pool)	Stream
Data/control	Yes	Internal	Continuous
			Periodic
			Event/signal
		External (other component)	Callback
			Stream

A component can be defined without its own processing loop. In this case, the component is typically used for wrapping external libraries (*API Wrapper*) and thus *cisst* does not manange any computation or threading.

If a component has a user-defined processing loop to execute within its context, two types of components can be defined depending on the characteristics of data that it primarily processes: a component for imaging and video data (*image/video*) and a component for general data or control (*data/control*).

The *stream* is designed to efficiently process real-time image or video data by sequentially executing multiple processing components, with minimal processing latency. We call these types of components *filters*. A typical stream requires CPU-intensive computations because it processes a large data set at a relatively low frequency (usually around 30-60 Hz). For example, the size of HD stereo vision data in 24-bit RGB format is about 11.86 MB per frame and this amounts to 355.96 MB per second at 30 Hz. In the *Stream* scheme, filters do not have their own internal processing thread but use a set of threads from thread pool provided by a *Stream Manager*. The *Stream Manager* maintains the thread pool and allocates all the threads to one filter at a time to process large data in parallel, thereby achieving minimal latency.

Components for data/control are suitable for processing robot control data or other types of data in general, where payload size is relatively small and thus data is processed with less computation than filters but at much higher frequencies (on the order of kHz). These components can have their own internal thread or run in another thread space. A component *with* its internal processing thread may execute its processing loop with three different schemes: *continuous* (no delay between execution of the processing loop; run the loop as soon as the current loop finishes), *periodic* (constant interval between execution), and *event/signal-based* (execution only when an event or signal arrives). A component *without* its own processing thread assumes no thread execution model and relies on an external thread or component that determines its computation scheme. One use case of this type of component is to implement *callbacks* from external libraries or devices. Another use case is called *stream* (the last row of the table). This is similar to the image stream described above, except that the thread source can be any other component and the stream is therefore limited to a single thread, rather than the thread pool provided by the *Stream Manager*. This is useful if multiple components are required to run synchronously because they can be grouped together such that the processing thread of the first component sequentially provides the processing thread for a second component and can then be chained to all the components in the group. With this computation scheme, the order of execution is specified.

A.4 Communication: Data Exchange Model

Use of the Command Pattern for data exchange between components enables loose coupling between two connected components, and this forms the basis of a lock-free, thread-safe, and efficient data exchange mechanism.[222] Data exchange uses strongly-typed payloads and the *cisstParameterTypes* library provides a set of standardized payload types that are frequently used in robotics.

This data exchange mechanism also allows the component-based framework to support both multi-*threaded* systems (which provides the best real-time performance) and multi-*process* distributed systems.[244] Because the programming model remains the same for both types of systems, it enables consistent and flexible deployment of components, thereby facilitating the system design process.

Recently, the correctness of this data exchange mechanism has been analyzed and proven using formal methods.[120, 246]

Appendix B

Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS) Artifacts

B.1 RobotTask

Code B.1 is a SAFECASS specification of safety features for the `RobotTask` component of the Robo-ELF system. Refer to Sec. 6.3.2 for more details.

Code B.1: SAFECASS artifact for `RobotTask` component of Robo-ELF

```
{
    "component": "RobotTask",
    "event": [
        // EStopException
        {   "name"            : "EVT_RobotTask_EStop",
            "severity"        : 10,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/EVT_RobotTask_EStop",
            "severity"        : 10,
                "state_transition": [ "E2N", "W2N" ]
        },
        // WatchdogException (warning)
        {   "name"             : "EVT_RobotTask_Watchdog_Short",
            "severity"         : 10,
            "state_transition": [ "N2W" ]
        },
        {   "name"             : "/EVT_RobotTask_Watchdog_Short",
            "severity"         : 10,
            "state_transition": [ "W2N" ]
        },
        // WatchdogException (error)
        {   "name"             : "EVT_RobotTask_Watchdog_Long",
            "severity"         : 10,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"             : "/EVT_RobotTask_Watchdog_Long",
            "severity"         : 10,
            "state_transition": [ "E2N", "W2N" ]
        },
        // EncoderException
        {   "name"             : "EVT_RobotTask_Encoder",
            "severity"         : 10,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"             : "/EVT_RobotTask_Encoder",
            "severity"         : 10,
            "state_transition": [ "E2N", "W2N" ]
        },
        // MotionException
        {   "name"             : "EVT_RobotTask_Motion",
            "severity"         : 10,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"             : "/EVT_RobotTask_Motion",
            "severity"         : 10,
            "state_transition": [ "E2N", "W2N" ]
        },
        // InvalidInputException
        {   "name"             : "EVT_RobotTask_InvalidInput",
            "severity"         : 10,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"             : "/EVT_RobotTask_InvalidInput",
            "severity"         : 10,
            "state_transition": [ "E2N", "W2N" ]
        }
    ],
    "service" : [
        {   // provided interface name
            "name" : "ProvidesThroatRobot",
            // dependency information
            "dependency" : {
                "s_R" : [ ],
                "s_A" : true,
                "s_F" : true
            }
        }
    ]
}
```

B.2 *cisst* Framework

Code B.2 presents a Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS) artifact that specifies *cisst*-specific safety features within the *cisst* framework. According to this specification, *cisst* defines three pairs of the GCM events and installs one filter to every component that owns a thread (of type `mtsTask`). Refer to Sec. 5.7 for more details.

Code B.2: SAFECASS artifact for the *cisst* component-based framework

```
{
    "component": "", // set by SAFECASS when deployed
    "event": [
        // Thread overrun
        {   "name"              : "EVT_THREAD_OVERRUN",
            "severity"          : 201,
            "state_transition": [ "N2W" ]
        },
        {   "name"              : "/EVT_THREAD_OVERRUN",
            "severity"          : 201,
            "state_transition": [ "W2N" ]
        },
        // Thread exception thrown
        {   "name"              : "EVT_THREAD_EXCEPTION",
            "severity"          : 201,
            "state_transition": [ "N2E" ]
        },
        {   "name"              : "/EVT_THREAD_EXCEPTION",
            "severity"          : 201,
            "state_transition": [ "E2N" ]
        },
        // Command queue full (for provided interface)
        {   "name"              : "EVT_COMMAND_QUEUE_FULL",
            "severity"          : 201,
            "state_transition": [ "N2E" ]
        },
        {   "name"              : "/EVT_COMMAND_QUEUE_FULL",
            "severity"          : 201,
            "state_transition": [ "E2N" ]
        }
    ],
    "filter": [
        // Filter to detect periodic thread overrun
        {   "class_name"        : "FilterThreshold",
            "target"            : {
                "type"          : "s_F",
                "component"     : ""  // "component" defined above is used instead
            },
            "type"              :  "ACTIVE",
            "argument" : {
                "input_signal"     : "",  // set by SAFECASS when deployed
                "threshold"        : 0.0, // set by SAFECASS when deployed
                "tolerance"        : 0.0, // tolerance margin
                "output_above"     : 1,
                "output_below"     : 0,
                "event_onset"      : "EVT_THREAD_OVERRUN",
                "event_completion" : "/EVT_THREAD_OVERRUN"
            },
            "last_filter"      :  true
        }
    ]
}
```

B.3 JR3

Code B.3 is a SAFECASS specification of safety features for the `JR3` component of the research ROBODOC system. This specification is based on the official documentation of the JR3 force sensor (JR3 Inc., Woodland, CA, USA). Refer to Sec. 6.2.3.2.2 for more details.

Code B.3: SAFECASS artifact for `JR3` component of ROBODOC

```
{
    "component": "JR3",
    "event": [
        // Device access error (due to error with device driver or hardware itself)
        {   "name"             : "EVT_JR3_DEVICE_ACCESS_ERROR",
            "severity"         : 20,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"             : "/EVT_JR3_DEVICE_ACCESS_ERROR",
            "severity"         : 20,
            "state_transition": [ "E2N", "W2N" ]
        },
        // Non-zero error count (error_count).  See JR3.cpp for details.
        {
            "name"             : "EVT_JR3_ERROR_COUNT",
            "severity"         : 15,
            "state_transition": [ "N2W" ]
        },
        {   "name"             : "/EVT_JR3_ERROR_COUNT",
            "severity"         : 15,
            "state_transition": [ "W2N" ]
        },
        // Non-zero warning bit(s) (WARNING_BITS).  See JR3.cpp for details.
        {   "name"             : "EVT_JR3_WARNING",
            "severity"         : 10,
            "state_transition": [ "N2W" ]
        },
        {   "name"             : "/EVT_JR3_WARNING",
            "severity"         : 10,
            "state_transition": [ "W2N" ]
        },
        // Non-zero error bit(s) (ERROR_BITS).  See JR3.cpp for details.
        {   "name"             : "EVT_JR3_ERROR",
            "severity"         : 20,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"             : "/EVT_JR3_ERROR",
            "severity"         : 20,
            "state_transition": [ "E2N", "W2N" ]
        }
    ],
    "filter" : [
        {   // common fields
            "class_name"           : "FilterChangeDetect",
            "target"               : {
                "type"             : "s_A",
                "component"        : "JR3"
            },
            "type"                 : "ACTIVE",
            //"debug"             : true,
            // fields specific to on/off state filter
            "argument" : {
                "input_signal"     : "Warning",
                "baseline"         : 0,
                "event_onset"      : "EVT_JR3_WARNING",
                "event_completion": "/EVT_JR3_WARNING"
            }
        },
        {
            "class_name"           : "FilterChangeDetect",
            "target"               : {
                "type"             : "s_A",
                "component"        : "JR3"
            },
            "type"                 : "ACTIVE",
            "argument" : {
                "input_signal"     : "ErrorCount",
                "baseline"         : 0,
                "event_onset"      : "EVT_JR3_ERROR_COUNT",
                "event_completion": "/EVT_JR3_ERROR_COUNT"
            }
        },
        {   "class_name"           : "FilterChangeDetect",
            "target"               : {
                "type"             : "s_A",
                "component"        : "JR3"
            },
            "type"                 : "ACTIVE",
            "argument" : {
                "input_signal"     : "Error",
                "baseline"         : 0,
                "event_onset"      : "EVT_JR3_ERROR",
                "event_completion": "/EVT_JR3_ERROR"
            }
        }
    ],
    "service" : [
        {   "name"     : "JR3Interface",
            "dependency" : {
                "s_R" : [ ],
                "s_A" : true,
                "s_F" : true
            }
        }
    ]
}
```

B.4 Control

Code B.4 is a SAFECASS specification of safety features for the `CONTROL` component of the research ROBODOC system. The GCM events defined here correspond to a subset of events that are used for the commercial ROBODOC system. Refer to Sec. 6.2.3.2.3 for more details.

Code B.4: SAFECASS artifact for `CONTROL` component of ROBODOC

```
{
    "component" : "CONTROL",
    "event" : [
        // Generic MCS event
        {   "name"            : "EVT_CONTROL_MCS",
            "severity"        : 10,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/EVT_CONTROL_MCS",
            "severity"        : 10,
            "state_transition": [ "E2N", "W2N" ]
        },
        // E_FORCE_FREEZE
        {   "name"            : "EVT_CONTROL_MCS_FORCE_FREEZE",
            "severity"        : 20,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/EVT_CONTROL_MCS_FORCE_FREEZE",
            "severity"        : 20,
            "state_transition": [ "E2N", "W2N" ]
        },
        // E_FORCE_EPO
        {   "name"            : "EVT_CONTROL_MCS_FORCE_EPO",
            "severity"        : 30,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/EVT_CONTROL_MCS_FORCE_EPO",
            "severity"        : 30,
            "state_transition": [ "E2N", "W2N" ]
        },
        // E_FORCE_HW_EPO
        {   "name"            : "EVT_CONTROL_MCS_FORCE_HW_EPO",
            "severity"        : 40,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/EVT_CONTROL_MCS_FORCE_HW_EPO",
            "severity"        : 40,
            "state_transition": [ "E2N", "W2N" ]
        },
        // E_MANIPWR
        {   "name"            : "EVT_CONTROL_MCS_MANIPWR",
            "severity"        : 10,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/EVT_CONTROL_MCS_MANIPWR",
            "severity"        : 10,
            "state_transition": [ "E2N", "W2N" ]
        },
        // E_PEND_PAUSE
        {   "name"            : "EVT_CONTROL_MCS_PEND_PAUSE",
            "severity"        : 50,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"            : "/EVT_CONTROL_MCS_PEND_PAUSE",
            "severity"        : 50,
            "state_transition": [ "E2N", "W2N" ]
        }
    ],
    "service" : [
        {   // provided interface name
            "name"    : "Robot",
            // service state dependency information
            "dependency" : {
                "s_R" : [ "ForceSensor", "AMCInterface", "PendantRequired" ],
                "s_A" : true,
                "s_F" : true
            }
        },
        {   "name"    : "prmRobot",
            "dependency" : {
                "s_R" : [ "ForceSensor", "AMCInterface", "PendantRequired" ],
                "s_A" : true,
                "s_F" : true
            }
        }
    ]
}
```

B.5 Cutting

Code B.5 is a SAFECASS specification of safety features for the `Cutting` component of the research ROBODOC system. Refer to Sec. 6.2.3.2.4 for more details.

Code B.5: SAFECASS artifact for `Cutting` component of ROBODOC

```
{
    "component": "Cutting",
    "event": [
        {   "name"              : "EVT_Cutting_OUTSIDE_WORKSPACE",
            "severity"          : 10,
            "state_transition": [ "N2E", "W2E" ]
        },
        {   "name"              : "/EVT_Cutting_OUTSIDE_WORKSPACE",
            "severity"          : 10,
            "state_transition": [ "E2N", "W2N" ]
        }
    ]
}
```

Glossary

G | S

G

Generic Component Model is a generic model that can be used to represent the operational status of component-based robot systems at *run-time* in an explicit, systematic, and structured manner. The model comprises (1) the *structural elements* that only contain components, interfaces, and connections between interfaces, and (2) the *state-based semantics* that use the structural elements to explicitly describe the run-time status of the system with support for *error propagation* across the component boundary. The use of minimal structural elements makes the model inherently generic, extensible, and customizable so that it can be specialized in a flexible manner to adapt to existing component models in robotics, thereby achieving reusability and interoperability. The state-based semantics defines states, events, filters, and inputs, thereby enabling a structured, systematic approach to designing and implementing safety features in accordance with the mechanism view of the Safety Design View. 100

S

Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS) is a run-time environment for the Generic Component Model. The SAFECASS provides an implementation of both the structural elements and the state-based semantics of the Generic Component Model. The fundamental design principle of the SAFECASS is the decomposition of safety features into reusable mechanisms and configurable specifications. This principle governs its safety-oriented layered architecture and overall design that consider the domain characteristics of medical robotics. The SAFECASS shows that it is possible to build a run-time environment for the Generic Component Model with the four design requirements: (1) conformity to the Generic Component Model, (2) flexibility and reusability, (3) testability, and (4) traceability. The SAFECASS aims to establish a run-time software environment for safety research in medical robotics, and to facilitate the development of safe medical robot systems in accordance with the Generic Component Model. A collection of reusable mechanisms and configurable specifications for application-specific requirements will enable the accumulation of safety knowledge and experiences across various applications and systems. 153

Safety Design View is a conceptual framework that can capture and describe both the run-time mechanisms and design decisions of safety features of medical robot systems in a systematic and structured manner. This conceptual framework is presented as the two-dimensional plane that represents the design space of safety features, with the Mechanism View on the horizontal axis and the System View on the vertical axis. The goal of Safety Design View is to (1) explicitly and intuitively describe safety features in a consistent and structured manner, (2) collect “good” practices on the design of safety features, and (3) facilitate sharing of knowledge and experience on safety within the community. 76

Acronyms

C | G | S

C

CBSE Component-based Software Engineering. 33

CBSS Component-based Software Systems. 99

G

GCM Generic Component Model. 100

S

SAFECASS Safety Architecture for Engineering Computer-Assisted Surgical Systems. 137

SDV Safety Design View. 76

Bibliography

[1] H. M. Shao, J. Y. Chen, T. K. Truong, I. S. Reed, and Y. S. Kwoh, “A New CT-Aided Robotic Stereotaxis System,” in *Annu. Symp. Comput. Appl. Med Care.*, vol. 13, Nov. 1985, pp. 668–672. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2578058 1, 22, 49

[2] R. A. Beasley, “Medical robots: Current systems and research directions,” *J. Robotics*, vol. 2012, pp. 1–14, 2012, article ID 401613. 1, 40

[3] R. Taylor, “A Perspective on Medical Robotics,” *Proceedings of the IEEE*, vol. 94, no. 9, pp. 1652–1664, Sep. 2006. 1, 40, 143, 274

[4] R. Taylor, H. Paul, P. Kazanzides, B. Mittelstadt, W. Hanson, J. Zuhars, B. Williamson, B. Musits, E. Glassman, and W. Bargar, “Taming the bull: Safety in a precise surgical robot,” in *Intl. Conf. on Advanced Robotics (ICAR)*, Jun. 1991, pp. 865–870. 2, 42, 43, 49, 219, 221

[5] P. Kazanzides, J. Zuhars, B. Mittelstadt, B. Williamson, P. Cain, F. Smith, L. Rose, and B. Musits, “Architecture of a surgical robot,” in *IEEE Intl. Conf. on Systems, Man and Cybernetics*, vol. 2, Oct. 1992, pp. 1624–1629. 2, 42, 49, 70, 79, 95, 144, 219, 225, 230, 239, 245

[6] B. Davies, “Safety of medical robots,” in *Intl. Conf. on Advanced Robotics (ICAR)*, vol. 11, 1993, pp. 311–317. 2

[7] B. L. Davies, *A discussion of safety issues for medical robots*, ser. Computer-Integrated Surgery. Cambridge, MA: MIT Press, 1996, pp. 287–298. 2, 41, 44

[8] R. H. Taylor, *Safety*, ser. Computer-Integrated Surgery. Cambridge, MA: MIT Press, 1996, pp. 283–286. 2

[9] B. Fei, W. S. Ng, S. Chauhan, and C. K. Kwoh, "The safety issues of medical robotics," *Reliability Engineering & System Safety*, vol. 73, no. 2, pp. 183–192, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/B6V4T-43MC8WS-8/2/a628b91a46303fb16c0c094bbf066853 2, 52, 56, 138

[10] T. Bray. (2014, Mar.) The JavaScript Object Notation (JSON) Data Interchange Format. Internet Engineering Task Force (IETF). [Online]. Available: https://tools.ietf.org/html/rfc7159 15, 157, 182

[11] R. Taylor, P. Kazanzides, B. Mittelstadt, and H. Paul, "Redundant consistency checking in a precise surgical robot," in *Proc. of the 12th Annual Intl. Conf. of the IEEE*, Nov. 1990, pp. 1933–1935. 22, 44, 49, 219, 221

[12] J. Knight, "Safety Critical Systems: Challenges and Directions," in *Intl. Conf. on Software Engineering (ICSE) 2002*, May 2002, pp. 547–550. 22

[13] N. G. Leveson, "Safety as a system property," *ACM Communications*, vol. 38, no. 11, p. 146, Nov. 1995. [Online]. Available: http://doi.acm.org/10.1145/219717.219816 23, 30, 60, 64

[14] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan. 2004. 24, 25, 26, 28, 99, 105

[15] N. G. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, 1995. 24, 30, 31, 41, 42, 60, 63, 67, 90, 155, 271

[16] M. Lipow, "Prediction of software failures," *Journal of Systems and Software*, vol. 1, pp. 71–75, 1979. 25

[17] B. Marick, "A survey of software fault surveys," Univ. of Illinois at Urbana-Champaign, Tech. Rep. UIU CD CS-R-90-1651, Dec. 1990. 25

[18] M. Y. Jung, "A layered approach for identifying systematic faults of component-based software systems," in *Proc. of the 16th Intl. Workshop on Component-Oriented Programming*. New York, NY, USA: ACM, 2011, pp. 17–24. 25

[19] Y. Zhang and J. Jiang, "Bibliographical review on reconfigurable fault-tolerant control systems," *Annual Reviews in Control*, vol. 32, no. 2, pp. 229–252, 2008. 26, 27, 29

[20] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. Kavuri, “A review of process fault detection and diagnosis, Part I: Quantitative model-based methods,” *Computers & Chemical Engineering*, vol. 27, no. 3, pp. 293–311, 2003. 27

[21] V. Venkatasubramanian, R. Rengaswamy, and S. Kavuri, “A review of process fault detection and diagnosis, Part II: Qualitative models and search strategies,” *Computers & Chemical Engineering*, vol. 27, no. 3, pp. 313–326, 2003. 27

[22] V. Venkatasubramanian, R. Rengaswamy, S. Kavuri, and K. Yin, “A review of process fault detection and diagnosis, Part III: Process history based methods,” *Computers & Chemical Engineering*, vol. 27, no. 3, pp. 327–346, 2003. 27

[23] R. Isermann, “Supervision, fault-detection and fault-diagnosis methods–an introduction,” *Control engineering practice*, vol. 5, no. 5, pp. 639–652, 1997. 27, 66

[24] ——, “Model-based fault-detection and diagnosis-status and applications,” *Annual Reviews in control*, vol. 29, no. 1, pp. 71–85, 2005. 27

[25] ——, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Verlag, 2006. 27

[26] A. Avizienis, “Toward systematic design of fault-tolerant systems,” *Computer*, vol. 30, no. 4, pp. 51–58, Apr. 1997. 28, 29

[27] A. Avižienis, “Design of fault-tolerant computers,” in *Proc. 1967 Fall Joint Computer Conf., AFIPS Conf. Proc., Vol. 31*, Thompson Books, Washington, D.C., USA, Nov. 1967, pp. 733–743. [Online]. Available: http://doi.acm.org/10.1145/1465611.1465708 28

[28] A. Avizienis, G. Gilley, F. Mathur, D. Rennels, J. Rohr, and D. Rubin, “The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design,” *IEEE Trans. on Computers*, vol. C-20, no. 11, pp. 1312–1321, Nov. 1971. 28

[29] B. Randell, “System structure for software fault tolerance,” in *Proc. of the Intl. Conf. on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 437–449. 28

[30] N. Dowler, “Applying software dependability principles to medical robotics,” *Computing Control Engineering Journal*, vol. 6, no. 5, pp. 222–225, oct 1995. 29

[31] G. Duchemin, P. Poignet, E. Dombre, and F. Peirrot, “Medically safe and sound [human-friendly robot dependability],” *IEEE Robotics & Automation Magazine*, vol. 11, no. 2, pp. 46–55, Jun. 2004.

[32] A. Sánchez, P. Poignet, E. Dombre, A. Menciassi, and P. Dario, “A design framework for surgical robots: Example of the araknes robot controller,” *Robotics and Autonomous Systems*, vol. 62, no. 9, pp. 1342–1352, 2014, intelligent Autonomous Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921889014000645 29, 43, 53, 58, 138, 139

[33] P. Kazanzides, “Safety design for medical robots,” in *IEEE Intl. Conf. on Engineering in Medicine and Biology Society (EMBS)*, Sep. 2009, pp. 7208–7211. 29, 40, 55, 56, 57, 68, 69, 211

[34] N. Leveson, “Software Safety: Why, What, and How,” *ACM Comput. Surv.*, vol. 18, no. 2, pp. 125–163, June 1986. [Online]. Available: http://doi.acm.org/10.1145/7474.7528 30

[35] M. P. E. Heimdahl, “Safety and software intensive systems: Challenges old and new,” in *Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 137–152. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.18 30, 32

[36] R. R. Lutz, “Software engineering for safety: a roadmap,” in *Proc. of the Conf. on The Future of Software Engineering*, ser. ICSE ’00. New York, NY, USA: ACM, 2000, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/336512.336556 30, 31, 32

[37] T. P. Kelly, “Arguing safety - a systematic approach to managing safety cases,” Ph.D. dissertation, Department of Computer Science, University of York, 1998. 30

[38] T. Kelly and R. Weaver, “The Goal Structuring Notation – A Safety Argument Notation,” in *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004. 31, 57

[39] T. Kelly and J. McDermid, “Safety case construction and reuse using patterns,” in *Proc. of 16th Intl. Conf. on Computer Safety, Reliability and Security (SAFECOMP’97)*, P. Daniel, Ed. Springer-Verlag London, Sep 1997, pp. 55–69. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-0997-6_5 31

[40] N. Leveson, "A new accident model for engineering safer systems," *Safety Science*, vol. 42, no. 4, pp. 237–270, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092575350300047X 32

[41] ——, *Engineering a safer world: Systems thinking applied to safety*. MIT Press, 2011. 32

[42] J. Hatcliff, A. Wassyng, T. Kelly, C. Comar, and P. Jones, "Certifiably safe software-dependent systems: Challenges and directions," in *Proc. of the on Future of Software Engineering*, ser. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 182–200. [Online]. Available: http://doi.acm.org/10.1145/2593882.2593895 32, 90

[43] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002. 33, 100

[44] W. Frakes and K. Kang, "Software reuse research: status and future," *IEEE Trans. Software Eng.*, vol. 31, no. 7, pp. 529–536, Jul. 2005. 33

[45] D. Brugali and P. Scandurra, "Component-Based Robotic Engineering (Part I)," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, Dec. 2009. 33, 86, 87, 276

[46] C. Pons, R. Giandini, and G. Arévalo, "A systematic review of applying modern software engineering techniques to developing robotic systems," *INGENIERÍA E INVESTIGACIÓN*, vol. 32, no. 1, pp. 58–63, 2012. 34, 276

[47] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *J. of Robotics*, 2012. 34, 87

[48] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS component model: a model-based development paradigm for complex robotics software systems," in *Proc. of the 28th Ann. ACM Symp. on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1758–1764. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480693 34, 88, 99

[49] M. Y. Jung, M. Balicki, A. Deguet, R. H. Taylor, and P. Kazanzides, "Lessons learned from the development of component-based medical robot systems," *J. of Software Engineering for Robotics (JOSER)*, Sep. 2014. 34, 88, 96, 99, 101, 137, 157, 158, 226, 256, 276

[50] 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)*, vol. 2, Sep. 2003, pp. 2766–2771. 34, 87, 99

[51] 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. 34

[52] 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 *IEEE Intl. Conf. on Robotics and Automation (ICRA), Workshop on Open Source Software*, 2009. 34, 87, 99, 101

[53] B. Song, S. Jung, C. Jang, and S. Kim, “An Introduction to Robot Component Model for OPRoS (Open Platform for Robotic Services),” in *Workshop Proceedings of Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots*, 2008, pp. 592–603. 34, 87

[54] 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*, ser. LNCS. Springer Berlin / Heidelberg, 2008, vol. 5325, pp. 87–98. 34

[55] A. Shakhimardanov and E. Prassler, “Comparative evaluation of robotic software integration systems: A case study,” in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2007. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4399375&tag=1 34

[56] A. Shakhimardanov, J. Paulus, N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschmar, “Best practice assessment of software technologies for robotics (deliverable d-2.1),” BRICS, Bonn-Rhein-Sieg University (BRSU), Tech. Rep., Jan. 2010. 34

[57] A. Shakhimardanov, N. Hochgeschwender, and G. K. Kraetzschmar, “Component models in robotics software,” in *Proc. of the 10th Performance Metrics for Intelligent Systems Workshop*, ser. PerMIS ’10. New York, NY, USA: ACM, 2010, pp. 82–87. [Online]. Available: http://doi.acm.org/10.1145/2377576.2377592 34

[58] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey, “Towards integrated safety analysis and design,” *SIGAPP Appl. Comput. Rev.*, vol. 2, no. 1, pp. 21–32, Mar. 1994. [Online]. Available: http://doi.acm.org/10.1145/381766.381770 34, 94

[59] 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*, ser. SCS ’03, vol. 33. Australian Computer Society, Inc., 2003, pp. 37–46. [Online]. Available: http://portal.acm.org/citation.cfm?id=1082051.1082054 34, 94

[60] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, *Fault Tree Handbook*. U. S. Nuclear Regulatory Commission, Jan. 1981. 34

[61] L. Grunske, B. Kaiser, and R. Reussner, "Specification and evaluation of safety properties in a component-based software engineering process," in *Component-Based Software Development for Embedded Systems*. Springer, 2005, pp. 249–274. 35, 88, 90, 91, 92, 93, 127

[62] L. Grunske, B. Kaiser, and Y. Papadopoulos, "Model-driven safety evaluation with state-event-based component failure annotations," in *Component-Based Software Engineering*. Springer, 2005, pp. 33–48. 35, 90, 93, 94, 131

[63] L. Grunske, "Towards an integration of standard component-based safety evaluation techniques with saveccm," in *Quality of Software Architectures*. Springer, 2006, pp. 199–213. 35

[64] D. Domis and M. Trapp, "Integrating safety analyses and component-based design," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, M. Harrison and M.-A. Sujan, Eds. Springer Berlin / Heidelberg, 2008, vol. 5219, pp. 58–71. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87698-4_8 35, 94

[65] ——, "Component-based abstraction in fault tree analysis," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, B. Buth, G. Rabe, and T. Seyfarth, Eds. Springer Berlin / Heidelberg, 2009, vol. 5775, pp. 297–310. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04468-7_24 35

[66] P. Corke, "Celebrating 50 years of robotics [from the editor's desk]," *IEEE Robotics & Automation Magazine*, vol. 18, no. 4, p. 4, Dec. 2011. 36, 87

[67] A. D. Santis, B. Siciliano, A. D. Luca, and A. Bicchi, "An atlas of physical human–robot interaction," *Mechanism and Machine Theory*, vol. 43, no. 3, pp. 253–270, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0094114X07000547 36

[68] J. Heinzmann and A. Zelinsky, "Quantitative safety guarantees for physical human-robot interaction," *Intl. Journal of Robotics Research*, vol. 22, no. 7–8, pp. 479–504, Jul. 2003. 36

[69] A. Bicchi and G. Tonietti, "Fast and "soft-arm" tactics [robot arm design]," *IEEE Robotics & Automation Magazine*, vol. 11, no. 2, pp. 22–33, Jun. 2004. 36

[70] S. Haddadin, A. Albu-Schaffer, A. De Luca, and G. Hirzinger, "Collision detection and reaction: A contribution to safe physical Human-Robot Interaction," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, Sep. 2008, pp. 3356–3363. 36

[71] S. Haddadin, A. Albu-Schäffer, and G. Hirzinger, "Requirements for safe robots: Measurements, analysis and new insights," *Intl. Journal of Robotics Research*, vol. 28, no. 11–12, pp. 1507–1527, 2009. 36, 82

[72] S. Haddadin, A. Albu-Schaffer, F. Haddadin, J. Rosmann, and G. Hirzinger, "Study on soft-tissue injury in robotics," *Robotics Automation Magazine, IEEE*, vol. 18, no. 4, pp. 20–34, Dec. 2011. 36

[73] S. Haddadin, *Towards Safe Robots - Approaching Asimov's 1st Law*. Springer Tracts in Advanced Robotics, 2014, vol. 90. 36

[74] S. Haddadin, S. Parusel, R. Belder, and A. Albu-Schäffer, "It Is (Almost) All about Human Safety: A Novel Paradigm for Robot Design, Control, and Planning," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8153, pp. 202–215. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40793-2_19 37

[75] M. Visinsky, I. Walker, and J. Cavallaro, "Layered dynamic fault detection and tolerance for robots," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. IEEE, 1993, pp. 180–187. 37, 70

[76] M. Visinsky, J. Cavallaro, and I. Walker, "Robotic fault detection and fault tolerance: A survey," *Reliability Engineering & System Safety*, vol. 46, no. 2, pp. 139–158, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0951832094901325 37

[77] V. Verma, G. Gordon, R. Simmons, and S. Thrun, "Real-time fault diagnosis [robot fault diagnosis]," *IEEE Robotics & Automation Magazine*, vol. 11, no. 2, pp. 56–66, Jun. 2004. 37

[78] J. Carlson, "Technical Report for the Safety Security Rescue Research Center: ICRA 2004 Robot Fault Diagnosis Workshop," 2004. 37

[79] G. Doukas, K. Thramboulidis, and Y. Koveos, "Using the function block model for robotic arm motion control," in *14th Mediterranean Conf. on Control and Automation*, June 2006, pp. 1–6. 37

[80] R. Hanai, H. Saito, Y. Nakabo, K. Fujiwara, T. Ogure, D. Mizuguchi, K. Homma, and K. Ohba, “RT-component based integration for IEC61508 ready system using SysML and IEC61499 function blocks,” in *IEEE/SICE Intl. Symp. on System Integration*, 2012, pp. 105–110. 37

[81] R. Woodman, A. F. Winfield, C. Harper, and M. Fraser, “Building safer robots: Safety driven control,” *Intl. J. of Robotics Research*, vol. 31, no. 13, pp. 1603–1626, 2012. [Online]. Available: http://ijr.sagepub.com/content/31/13/1603.abstract 38, 87, 140

[82] T. Tadele, T. de Vries, and S. Stramigioli, “The safety of domestic robotics: A survey of various safety-related publications,” *IEEE Robotics & Automation Magazine*, vol. 21, no. 3, pp. 134–142, Sept 2014. 38, 87, 97

[83] M. Balicki, “Augmentation of human skill in microsurgery,” Ph.D. dissertation, The Johns Hopkins University, Baltimore, Maryland, USA, Sep. 2013. 39

[84] R. Howe and Y. Matsuoka, “Robotics for surgery,” *Annual Review of Biomedical Engineering*, vol. 1, no. 1, pp. 211–240, 1999. 40

[85] B. Davies, “A review of robotics in surgery,” *Proc. of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine*, vol. 214, no. 1, pp. 129–140, 2000. [Online]. Available: http://pih.sagepub.com/content/214/1/129.abstract 40, 47

[86] K. Cleary and C. Nguyen, “State of the art in surgical robotics: Clinical applications and technology challenges,” *Computer Aided Surgery*, vol. 6, no. 6, pp. 312–328, 2001. [Online]. Available: http://dx.doi.org/10.1002/igs.10019 40

[87] R. Taylor and D. Stoianovici, “Medical robotics in computer-integrated surgery,” *IEEE Trans. on Robotics and Automation*, vol. 19, no. 5, pp. 765–781, Oct. 2003. 40, 143

[88] P. P. Pott, H.-p. Scharf, and M. L. R. Schwarz, “Today’s state of the art in surgical robotics,” *Computer Aided Surgery*, vol. 10, no. 2, pp. 101–132, 2005. [Online]. Available: http://informahealthcare.com/doi/abs/10.3109/10929080500228753 40

[89] J. Troccaz, “Computer and robot-assisted medical intervention,” *Springer Handbook of Automation*, pp. 1451–1466, 2009. 40

[90] A. Wolf and M. Shoham, “Medical automation and robotics,” *Springer Handbook of Automation*, pp. 1397–1407, 2009. 40, 47

[91] G. Dogangil, B. L. Davies, and F. Rodriguez y Baena, “A review of medical robotics for minimally invasive soft tissue surgery,” *Proc. of the Institution of Mechanical Engineers, Part H: J. of Engineering in Medicine*, vol. 224, no. 5, pp. 653–679, 2010. [Online]. Available: http://pih.sagepub.com/content/224/5/653.abstract 40

[92] M. D. O’Toole, K. Bouazza-Marouf, D. Kerr, M. Gooroochurn, and M. Vloeberghs, “A methodology for design and appraisal of surgical robotic systems,” *Robotica*, vol. 28, no. Special Issue 02, pp. 297–310, 2010. [Online]. Available: http://dx.doi.org/10.1017/S0263574709990658 40

[93] G. P. Moustris, S. C. Hiridis, K. M. Deliparaschos, and K. M. Konstantinidis, “Evolution of autonomous and semi-autonomous robotic surgical systems: a review of the literature,” *Intl. J. of Medical Robotics and Computer Assisted Surgery (MRCAS)*, vol. 7, no. 4, pp. 375–392, 2011. [Online]. Available: http://dx.doi.org/10.1002/rcs.408 40

[94] J. Rosen, *Medical Devices: Surgical and Image Guided Technologies*, 1st ed. John Wiley & Sons, Inc., 2013, ch. 5, pp. 63–98. 40, 71

[95] R. Taylor, J. Funda, B. Eldridge, S. Gomory, K. Gruben, D. LaRose, M. Talamini, L. Kavoussi, and J. Anderson, “A telerobotic assistant for laparoscopic surgery,” *IEEE Engineering in Medicine and Biology Magazine*, vol. 14, no. 3, pp. 279–288, May/Jun. 1995. 41, 50

[96] D. Stoianovici, L. Whitcomb, J. Anderson, R. Taylor, and L. Kavoussi, “A modular surgical robotic system for image guided percutaneous procedures,” in *Medical Image Computing and Computer-Assisted Interventation (MICCAI)*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Nov. 1998, vol. 1496, pp. 404–410. [Online]. Available: http://dx.doi.org/10.1007/BFb0056225 41, 51

[97] W.-H. Zhu, S. Salcudean, S. Bachmann, and P. Abolmaesumi, “Motion/force/image control of a diagnostic ultrasound robot,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, vol. 2, 2000, pp. 1580–1585. 41, 52, 79

[98] E. Degoulange, L. Urbain, P. Caron, S. Boudet, J. Gariepy, J.-L. Megnien, F. Pierrot, and E. Dombre, “HIPPOCRATE: an intrinsically safe robot for medical applications,” in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, vol. 2, Oct. 1998, pp. 959–964. 41, 51, 79

[99] G. Duchemin, E. Dombre, F. Pierrot, P. Poignet, and E. Dégoulange, “SCALPP: A Safe Methodology to Robotize Skin Harvesting,” in *MICCAI*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, vol. 2208, pp. 309–316. [Online]. Available: http://dx.doi.org/10.1007/3-540-45468-3_37 42, 44, 52

[100] E. Dombre, G. Duchemin, P. Poignet, and F. Pierrot, “Dermarob: A safe robot for reconstructive surgery,” *IEEE Trans. on Robotics and Automation*, vol. 19, no. 5, pp. 876–884, Oct. 2003. 42, 52, 79

[101] P. Bast, A. Popovic, T. Wu, S. Heger, M. Engelhardt, W. Lauer, K. Radermacher, and K. Schmieder, “Robot- and computer-assisted craniotomy: resection planning, implant modelling and robot safety,” *The International Journal of Medical Robotics and Computer Assisted Surgery*, vol. 2, no. 2, pp. 168–178, 2006. [Online]. Available: http://dx.doi.org/10.1002/rcs.85 42, 53

[102] G. Guthart and J. Salisbury, J.K., “The Intuitive™ Telesurgery System: Overview and Application,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, vol. 1, 2000, pp. 618–621. 42, 43, 51, 96, 144

[103] U. Hagn, M. Nickl, S. Jörg, G. Passig, T. Bahls, A. Nothhelfer, F. Hacker, L. Le-Tien, A. Albu-Schäffer, R. Konietschke *et al.*, “The DLR MIRO: a versatile lightweight robot for surgical applications,” *Industrial Robot: An International Journal*, vol. 35, no. 4, pp. 324–336, 2008. 42, 53, 70

[104] E. Kobayashi, K. Masamune, I. Sakuma, T. Dohi, and D. Hashimoto, “A New Safe Laparoscopic Manipulator System with a Five-Bar Linkage Mechanism and an Optical Zoom,” *Computer Aided Surgery*, vol. 4, no. 4, pp. 182–192, 1999. [Online]. Available: http://informahealthcare.com/doi/abs/10.3109/10929089909148172 43, 51

[105] J. Guiochet and A. Vilchis, “Safety analysis of a medical robot for tele-echography,” *Proc. of the 2nd IARP/IEEE RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, pp. 217–227, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.74.4749&rep=rep1&type=pdf 43, 44, 52, 79, 96

[106] R. Taylor, H. Paul, B. Mittelstadt, E. Glassman, B. Musits, and W. Bargar, “Robotic total hip replacement surgery in dogs,” in *Proc. of the 11th IEEE Medicine & Biology Conference (EMBC)*, vol. 3, Seattle, Washington, Nov. 1989, pp. 887–889. 43, 49

[107] P. Kazanzides, J. Zuhars, B. Mittelstadt, and R. Taylor, “Force sensing and control for a surgical robot,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, vol. 1, May 1992, pp. 612–617. 43, 49, 79, 219, 232

[108] T. Lueth, A. Hein, J. Albrecht, M. Demirtas, S. Zachow, E. Heissler, M. Klein, H. Menneking, G. Hommel, and J. Bier, “A surgical robot system for maxillofacial surgery,” in *Proc. of the 24th Annual Conf. of the IEEE Industrial Electronics Society (IECON)*, vol. 4, Aug.-Sep. 1998, pp. 2470–2475. 43, 51

[109] F. Rodriguez, S. Harris, M. Jakopec, A. Barrett, P. Gomes, J. Henckel, J. Cobb, and B. Davies, “Robotic clinical trials of uni-condylar arthroplasty,” *Intl. J. of Medical Robotics and Computer Assisted Surgery*, vol. 1, no. 4, pp. 20–28, 2005. [Online]. Available: http://dx.doi.org/10.1002/rcs.52 43, 53

[110] J. Troccaz, S. Lavallee, and E. Hellion, “A passive arm with dynamic constraints: a solution to safety problems in medical robotics,” in *Intl. Conf. on Systems, Man and Cybernetics*, vol. 3, Oct. 1993, pp. 166–171. 43, 50

[111] S. J. Harris, F. Arambula-Cosio, Q. Mei, R. D. Hibberd, B. L. Davies, J. E. A. Wickham, M. S. Nathan, and B. Kundu, “The probot—an active robot for prostate resection,” *Proc. of the Inst. of Mech. Engineers, Part H: J. of Eng. in Med.*, vol. 211, no. 4, pp. 317–325, 1997. [Online]. Available: http://pih.sagepub.com/content/211/4/317.abstract 43, 50

[112] U. Laible, T. Bürger, and G. Pritschow, “A fail-safe dual channel robot control for surgery applications,” *Safety science*, vol. 42, no. 5, pp. 423–436, 2004. 44, 52, 56, 65, 79, 96

[113] P. Varley, “Techniques for development of safety-related software for surgical robots,” *IEEE Trans. on Information Technology in Biomedicine*, vol. 3, no. 4, pp. 261–267, Dec. 1999. 44, 51, 55, 56, 96

[114] A. Rovetta, “Telerobotic surgery control and safety,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, vol. 3, 2000, pp. 2895–2900. 44, 51, 55, 56

[115] W. Korb, M. Kornfeld, W. Birkfellner, R. Boesecke, M. Figl, M. Fuerst, J. Kettenbach, A. Vogler, S. Hassfeld, and G. Kornreif, “Risk analysis and safety assessment in surgical robotics: A case study on a biopsy robot,” *Minimally Invasive Therapy & Allied Technologies*, vol. 14, no. 1, pp. 23–31, 2005. [Online]. Available: http://informahealthcare.com/doi/abs/10.1080/13645700510010827 44, 52, 56, 96

[116] J. Guiochet, Q. A. Do Hoang, M. Kaâniche, and D. Powell, "Applying Existing Standards to a Medical Rehabilitation Robot: Limits and Challenges," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), Workshop on Safety in Human-Robot Coexistence & Interaction*, 2012. 44, 53, 56, 57

[117] K. Gary, L. Ibanez, S. Aylward, D. Gobbi, M. Blake, and K. Cleary, "IGSTK: an open source software toolkit for image-guided surgery," *IEEE Computer*, vol. 39, no. 4, pp. 46–53, April 2006. 44, 52, 95

[118] M. J. H. Lum, D. C. W. Friedman, G. Sankaranarayanan, H. King, K. Fodero, R. Leuschke, B. Hannaford, J. Rosen, and M. N. Sinanan, "The RAVEN: Design and Validation of a Telesurgery System," *Intl. J. of Robotics Research*, vol. 28, no. 9, pp. 1183–1197, 2009. [Online]. Available: http://ijr.sagepub.com/content/28/9/1183.abstract 44, 53

[119] R. Muradore, D. Bresolin, L. Geretti, P. Fiorini, and T. Villa, "Robotic surgery - formal verification of plans," *IEEE Robotics & Automation Magazine*, vol. 18, no. 3, pp. 24–32, Sep. 2011. 44, 53

[120] P. Kazanzides, Y. Kouskoulas, A. Deguet, and Z. Shao, "Proving the correctness of concurrent robot software," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May. 2012, pp. 4718–4723. 44, 53, 185, 286

[121] Y. Kouskoulas, D. Renshaw, A. Platzer, and P. Kazanzides, "Certifying the safe design of a virtual fixture control algorithm for a surgical robot," in *Hybrid Systems: Computation and Control (part of CPS Week 2013), HSCC'13, Philadelphia, PA, USA, April 8-13, 2013*. ACM, 2013. 44, 53

[122] J. L. Garbini, R. G. Kaiura, J. A. Sidles, R. V. Larson, and F. A. Matson, "Robotic instrumentation in total knee arthroplasty," in *Proc. 33rd Annu. Meeting, Orthopaedic Research Society*, San Francisco, CA, Jan. 1987, p. 413. 47

[123] E. Watanabe, T. Watanabe, S. Manaka, Y. Mayanagi, and K. Takakura, "Three-dimensional digitizer (neuronavigator): New equipment for computed tomography-guided stereotaxic surgery," *Surgical Neurology*, vol. 27, no. 6, pp. 543–547, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0090301987901522 47

[124] Y. Kosugi, E. Watanabe, J. Goto, T. Watanabe, S. Yoshimoto, K. Takakura, and J. Ikebe, "An articulated neurosurgical navigation system using mri and ct images," *IEEE Trans. on Biomedical Engineering*, vol. 35, no. 2, pp. 147–152, Feb. 1988. 47

[125] J. Troccaz and Y. Delnondedieu, “Semi-active guiding systems in surgery. A two-DOF prototype of the passive arm with dynamic constraints (PADyC),” *Mechatronics*, vol. 6, no. 4, pp. 399–421, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0957415896000037 47, 50

[126] Y. Kwoh, J. Hou, E. Jonckheere, and S. Hayati, “A Robot with Improved Absolute Positioning Accuracy for CT Guided Stereotactic Brain Surgery,” *IEEE Trans. on Biomedical Engineering*, vol. 35, no. 2, pp. 153–160, Feb. 1988. 49

[127] B. L. Davies, R. D. Hibberd, M. J. Coptcoat, and J. E. A. Wickham, “A surgeon robot prostatectomy - a laboratory evaluation,” *Journal of Medical Engineering & Technology*, vol. 13, no. 6, pp. 273–277, 1989. [Online]. Available: http://informahealthcare.com/doi/abs/10.3109/03091908909016201 49

[128] S. Lavallee, “A new system for computer assisted neurosurgery,” in *IEEE Intl. Conf. on Engineering in Medicine and Biology Society (EMBS)*, vol. 3, Nov. 1989, pp. 926–927. 49

[129] S. Lavallee, J. Troccaz, L. Gaborit, P. Cinquin, A. Benabid, and D. Hoffmann, “Image guided operating robot: a clinical application in stereotactic neurosurgery,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Nice, France, May 1992, pp. 618–624. 49

[130] R. Taylor, B. Mittelstadt, H. Paul, W. Hanson, P. Kazanzides, J. Zuhars, B. Williamson, B. Musits, E. Glassman, and W. Bargar, “An image-directed robotic system for precise orthopaedic surgery,” *IEEE Trans. on Robotics and Automation*, vol. 10, no. 3, pp. 261–275, Jun. 1994. 49, 219

[131] B. L. Davies, R. D. Hibberd, W. Ng, A. Timoney, and J. E. A. Wickham, “The development of a surgeon robot for prostatectomies,” *Proc. of the Institution of Mechanical Engineers, Part H: J. of Engineering in Medicine*, vol. 205, pp. 35–38, Mar 1991. 49

[132] B. Davies, R. Hibberd, W. Ng, A. Timoney, and J. Wickham, “A surgeon robot for prostatectomies,” in *Intl. Conf. on Advanced Robotics (ICAR)*, vol. 1, Jun. 1991, pp. 871–875. 49

[133] J. Drake, M. Joy, A. Goldenberg, D. Kreindler *et al.*, “Computer-and robot-assisted resection of thalamic astrocytomas in children.” *Neurosurgery*, vol. 29, no. 1, pp. 27–33, 1991. 49

[134] R. Taylor, C. Cutting, Y.-Y. Kim, A. Kalvin, D. Larose *et al.*, “A model-based optimal planning and execution system with active sensing and passive manipulation for augmentation

of human precision in computer-integrated surgery," in *Proc. Intl. Symposium on Experimental Robotics*. Toulouse, Jun. 1991. [Online]. Available: http://dx.doi.org/10.1007/BFb0036139 49

[135] T. Kienzle III, S. Stulberg, M. Peshkin, A. Quaid, and C.-h. Wu, "An integrated CAD-robotics system for total knee replacement surgery," in *IEEE Intl. Conf. on Systems, Man and Cybernetics*, vol. 2, Oct. 1992, pp. 1609–1614. 49

[136] P. Kazanzides, B. Mittelstadt, J. Zuhars, P. Cain, and H. Paul, "Surgical and industrial robots: Comparison and case study," in *Proc. of the Intl. Robots and Vision Automation Conf.*, Detroit, MI, Apr. 1993, pp. 1019–1026. 49, 79, 142, 219, 223

[137] P. Kazanzides, B. Mittelstadt, B. Musits, W. Bargar, J. Zuhars, B. Williamson, P. Cain, and E. Carbone, "An integrated system for cementless hip replacement," *IEEE Engineering in Medicine and Biology Magazine*, vol. 14, no. 3, pp. 307–313, May/Jun. 1995. 49, 79, 219

[138] P. Kazanzides, "Robot Assisted Surgery: The ROBODOC® Experience," in *Intl. Symp. on Robotics (ISR)*, vol. 30, Tokyo, Japan, 1999, pp. 281–286. 49, 219, 221

[139] P. Cain, P. Kazanzides, J. Zuhars, B. Mittelstadt, and H. Paul, "Safety considerations in a surgical robot," in *Proc. of 30th Annual Rocky Mountain Bioengineering Symp.*, vol. 29, Apr. 1993, pp. 291–294. 49, 79, 219

[140] B. Mittelstadt, P. Kazanzides, J. Zuhars, P. Cain, and B. Williamson, "Robotic surgery: achieving predictable results in an unpredictable environment," in *Intl. Conf. on Advanced Robotics (ICAR)*, vol. 11, 1993, pp. 367–372. 49, 142, 219

[141] N. Villotte, D. Glauser, P. Flury, and C. Burckhardt, "Conception of stereotactic instruments for the neurosurgical robot minerva," in *IEEE Intl. Conf. on Engineering in Medicine and Biology Society (EMBS)*, vol. 3, Nov. 1992, pp. 1089–1090. 49

[142] D. Glauser, P. Flury, M. Epitaux, Y. Piguet, and C. Burckhardt, "Neurosurgical operation with the dedicated robot minerva," *IEEE Engineering in Medicine and Biology Magazine*, pp. 347–351, 1993. 49

[143] D. Glauser, P. Flury, and C. W. Burckhardt, "Mechanical concept of the neurosurgical robot 'Minerva'," *Robotica*, vol. 11, no. 6, pp. 567–575, Oct. 1993. [Online]. Available: http://dx.doi.org/10.1017/S0263574700019421 49

[144] D. Glauser, H. Fankhauser, M. Epitaux, J.-L. Hefti, and A. Jaccottet, “Neurosurgical Robot Minerva: First Results and Current Developments,” *Computer Aided Surgery*, vol. 1, no. 5, pp. 266–272, 1995. [Online]. Available: http://informahealthcare.com/doi/abs/10.3109/10929089509106332 49

[145] M. Fadda, T. Wang, B. Allota, P. Dario, M. Marcacci, and S. Martelli, “Safety requirements in a robotic surgical system: First analysis and approach,” in *Intl. Symp. on Measurement and Control in Robotics*, 1993, pp. 21–24. 49

[146] F. A. Matsen, J. L. Garbini, J. A. Sidles, B. Pratt, D. Baumgarten, and R. Kaiura, “Robotic assistance in orthopaedic surgery: A proof of principle using distal femoral arthroplasty,” *Clinical Orthopaedics and Related Research*, vol. 296, pp. 178–186, Nov. 1993. 50

[147] W. Ng, B. Davies, R. Hibberd, and A. Timoney, “Robotic surgery–a first-hand experience in transurethral resection of the prostate,” *IEEE Engineering in Medicine and Biology Magazine*, vol. 12, no. 1, pp. 120–125, Mar. 1993. 50

[148] O. Schneider and J. Troccaz, “A six-degree-of-freedom passive arm with dynamic constraints (PADyC) for cardiac surgery application: Preliminary experiments,” *Computer Aided Surgery*, vol. 6, no. 6, pp. 340–351, 2001. [Online]. Available: http://dx.doi.org/10.1002/igs.10020 50

[149] L. R. Kavoussi, R. G. Moore, A. W. Partin, J. S. Bender, M. E. Zenilman, and R. M. Satava, “Telerobotic assisted laparoscopicsurgery: Initial laboratory and clinical experience,” *Urology*, vol. 44, no. 1, pp. 15–19, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0090429594800030 50

[150] J. Sackier and Y. Wang, “Robotically assisted laparoscopic surgery,” *Surgical Endoscopy*, vol. 8, pp. 63–66, 1994. [Online]. Available: http://dx.doi.org/10.1007/BF02909496 50

[151] S. Ho, R. Hibberd, and B. Davies, “Robot assisted knee surgery,” *IEEE Engineering in Medicine and Biology Magazine*, vol. 14, no. 3, pp. 292–300, May/Jun. 1995. 50, 79

[152] K. Masamune, E. Kobayashi, Y. Masutani, M. Suzuki, T. Dohi, H. Iseki, and K. Takakura, “Development of an MRI-Compatible Needle Insertion Manipulator for Stereotactic Neurosurgery,” *Computer Aided Surgery*, vol. 1, no. 4, pp. 242–248, 1995. [Online]. Available: http://informahealthcare.com/doi/abs/10.3109/10929089509106330 50

[153] W. Ng and C. Tan, “On safety enhancements for medical robots,” *Reliability Engineering & System Safety*, vol. 54, no. 1, pp. 35–45, 1996. 50, 70

[154] J. Funda, R. Taylor, B. Eldridge, S. Gomory, and K. Gruben, “Constrained cartesian motion control for teleoperated surgical robots,” *IEEE Trans. on Robotics and Automation*, vol. 12, no. 3, pp. 453–465, Jun. 1996. 50

[155] G. Brandt, K. Radermacher, S. Lavallée, H. Staudte, and G. Rau, “A compact robot for image guided orthopedic surgery: Concept and preliminary results,” in *Proc. 1st Joint Conf. CVRMed and MRCAS*, 1997, vol. 1205, pp. 767–776. [Online]. Available: http://dx.doi.org/10.1007/BFb0029302 50

[156] G. Brandt, A. Zimolong, L. Carrat, P. Merloz, H.-W. Staudte, S. Lavallee, K. Radermacher, and G. Rau, “Crigos: a compact robot for image-guided orthopedic surgery,” *IEEE Trans. on Information Technology in Biomedicine*, vol. 3, no. 4, pp. 252–260, Dec. 1999. 50

[157] B. Davies, K. Fan, R. Hibberd, M. Jakopec, and S. Harris, “Acrobot - using robots and surgeons synergistically in knee surgery,” in *Proc. of the Int. Conf. on Advanced Robotics (ICAR)*, Jul. 1997, pp. 173–178. 50

[158] S. Harris, W. Lin, K. Fan, R. Hibberd, J. Cobb, R. Middleton, and B. Davies, “Experiences with robotic systems for knee surgery,” in *CVRMed-MRCAS*. Springer Berlin / Heidelberg, 1997, pp. 757–766. [Online]. Available: http://dx.doi.org/10.1007/BFb0029301 50

[159] J. Cadeddu, D. Stoianovici, R. Chen, R. Moore, and L. Kavoussi, “Stereotactic mechanical percutaneous renal access,” *J. of Endourology*, vol. 12, no. 2, pp. 121–125, Apr. 1998. 51

[160] M. Cavusoglu, F. Tendick, M. Cohn, and S. Sastry, “A laparoscopic telesurgical workstation,” *IEEE Trans. on Robotics and Automation*, vol. 15, no. 4, pp. 728–739, Aug. 1999. 51

[161] F. Pierrot, E. Dombre, E. Degoulange, L. Urbain, P. Caron, S. Boudet, J. Gariepy, and J.-L. Megnien, “Hippocrate: a safe robot arm for medical applications with force feedback,” *Medical Image Analysis*, vol. 3, no. 3, pp. 285–300, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1361841599800255 51, 96

[162] H. Reichenspurner, R. J. Damiano, M. Mack, D. H. Boehm, H. Gulbins, C. Detter, B. Meiser, R. Ellgass, and B. Reichart, “Use of the voice-controlled and computer-assisted surgical system ZEUS for endoscopic coronary artery bypass grafting,” *J. Thoracic and Cardiovascular Surgery*, vol. 118, no. 1, pp. 11–16, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022522399701340 51

[163] M. Ghodoussi, S. Butner, and Y. Wang, “Robotic Surgery - The Transatlantic Case,” in *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2002, pp. 1882–1888. 51

[164] R. Taylor, P. Jensen, L. Whitcomb, A. Barnes, R. Kumar, D. Stoianovici, P. Gupta, Z. Wang, E. Dejuan, and L. Kavoussi, “A steady-hand robotic system for microsurgical augmentation,” *Intl. J. of Robotics Research*, vol. 18, no. 12, pp. 1201–1210, 1999. [Online]. Available: http://ijr.sagepub.com/content/18/12/1201.abstract 51

[165] R. Z. Tombropoulos, J. R. Adler, and J.-C. Latombe, “CARABEAMER: a treatment planner for a robotic radiosurgical system with general kinematics,” *Medical Image Analysis*, vol. 3, no. 3, pp. 237–264, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S136184159980022X 51

[166] D. Engel, J. Raczkowsky, and H. Worn, “A safe robot system for craniofacial surgery,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, vol. 2, 2001, pp. 2020–2024. 52, 74, 79

[167] A. Vilchis Gonzales, P. Cinquin, J. Troccaz, A. Guerraz, B. Hennion, F. Pellissier *et al.*, “TER: A System for Robotic Tele-echography,” in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, vol. 2208, pp. 326–334. [Online]. Available: http://dx.doi.org/10.1007/3-540-45468-3_39 52

[168] M. Jakopec, S. Harris, F. Rodriguez y Baena, P. Gomes, J. Cobb, and B. Davies, “The first clinical application of a “hands-on” robotic knee surgery system,” *Computer Aided Surgery*, vol. 6, no. 6, pp. 329–339, 2001. [Online]. Available: http://dx.doi.org/10.1002/igs.10023 52

[169] M. Jakopec, F. Rodriguez y Baena, S. Harris, P. Gomes, J. Cobb, and B. Davies, “The hands-on orthopaedic robot “Acrobot”: Early clinical trials of total knee replacement surgery,” *IEEE Trans. on Robotics and Automation*, vol. 19, no. 5, pp. 902–911, Oct. 2003. 52, 79

[170] K. Masamune, G. Fichtinger, A. Patriciu, R. C. Susil, R. H. Taylor, L. R. Kavoussi, J. H. Anderson, I. Sakuma, T. Dohi, and D. Stoianovici, “System for robotically assisted percutaneous procedures with computed tomography guidance,” *Computer Aided Surgery*, vol. 6, no. 6, pp. 370–383, 2001. [Online]. Available: http://dx.doi.org/10.1002/igs.10024 52

[171] W. Korb, D. Engel, R. Boesecke, G. Eggers, R. Marmulla, N. O’Sullivan, J. Raczkowsky, and S. Hassfeld, “Risk analysis for a reliable and safe surgical robot system,” in *Computer Assisted Radiology and Surgery (CARS), Proc. of the 17th*

Intl. Congress and Exhibition, vol. 1256, 2003, pp. 766–770. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0531513103004023 52, 96

[172] K. Cleary, L. Ibanez, S. Ranjan, and B. Blake, "IGSTK: a software toolkit for image-guided surgery applications," *International Congress Series*, vol. 1268, no. 0, pp. 473 – 479, 2004, computer Assisted Radiology and Surgery (CARS). Proc. of the 18th Intl. Congress and Exhibition. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0531513104003991 52, 95

[173] K. Gary, A. Enquobahrie, L. Ibanez, P. Cheng, Z. Yaniv, K. Cleary, S. Kokoori, B. Muffih, and J. Heidenreich, "Agile methods for open source safety-critical software," *Software: Practice and Experience*, vol. 41, no. 9, pp. 945–962, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1075 52, 88

[174] C. Plaskos, P. Cinquin, S. Lavallée, and A. J. Hodgson, "Praxiteles: a miniature bone-mounted robot for minimal access total knee arthroplasty," *Intl. J. of Medical Robotics and Computer Assisted Surgery*, vol. 1, no. 4, pp. 67–79, 2005. [Online]. Available: http://dx.doi.org/10.1002/rcs.59 52

[175] K. Fodero, H. H. King, M. J. Lum, C. Bland, J. Rosen, M. Sinanan, and B. Hannaford, "Control system architecture for a minimally invasive surgical robot," in *Medicine Meets Virtual Reality (MMVR) 14*, Jan. 2006, pp. 157–159. 53, 96

[176] M. Lum, D. Trimble, J. Rosen, K. Fodero, H. King, G. Sankaranarayanan, J. Dosher, R. Leuschke, B. Martin-Anderson, M. Sinanan, and B. Hannaford, "Multidisciplinary approach for developing a new minimally invasive surgical robotic system," in *Intl. Conf. on Biomedical Robotics and Biomechatronics (BioRob)*, Feb. 2006, pp. 841–846. 53

[177] L. Sanchez, M. Le, K. Rabenorosoa, C. Liu, N. Zemiti, P. Poignet, E. Dombre, A. Menciassi, and P. Dario, "A Case Study of Safety in the Design of Surgical Robots: The ARAKNES Platform," in *Intelligent Autonomous Systems 12*, ser. Advances in Intelligent Systems and Computing. Springer Berlin Heidelberg, 2013, vol. 194, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33932-5_12 53, 56, 57, 58

[178] D. Kortenkamp and R. Simmons, *Robotic Systems Architectures and Programming*, ser. Handbook of Robotics. Springer, 2009, ch. 8, pp. 187–206. 70, 139, 153

[179] J. Speich and J. Rosen, "Medical robotics," *Encyclopedia of Biomaterials and Biomed. Eng.*, pp. 983–993, 2004. 71

[180] S. Haddadin, S. Haddadin, A. Khoury, T. Rokahr, S. Parusel, R. Burgkart, A. Bicchi, and A. Albu-Schaffer, "A truly safely moving robot has to know what injury it may cause," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2012, pp. 5406–5413. 82

[181] H. Heinecke, K. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J. Maté, K. Nishikawa, and T. Scharnhorst, "AUTomotive Open System ARchitecture: an industry-wide initiative to manage the complexity of emerging automotive E/E-architectures," *Convergence*, pp. 18–20, 2004. 86, 99, 140

[182] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The Architecture Analysis & Design Language (AADL): An Introduction," CMU/SEI, Tech. Rep., Feb. 2006. 86, 94, 99

[183] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1–42, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1328671.1328672 87

[184] M. Y. Jung, R. H. Taylor, and P. Kazanzides, "Safety Design View: A conceptual framework for systematic understanding of safety features of medical robot systems," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2014. 87

[185] O. Lisagor, J. McDermid, and D. Pumfrey, "Towards a practicable process for automated safety analysis," in *Proc. of the 24th Intl. System Safety Conference (ISSC)*, 2006. 90, 91, 94, 95

[186] L. Grunske and J. Han, "A comparative study into architecture-based safety evaluation methodologies using aadl's error annex and failure propagation models," in *High Assurance Systems Engineering Symposium (HASE)*, Dec 2008, pp. 283–292. 90, 94, 96, 145

[187] R. Bloomfield and P. Bishop, "Safety and Assurance Cases: Past, Present and Possible Future – an Adelard Perspective," in *Making Systems Safer*, C. Dale and T. Anderson, Eds. Springer London, 2010, pp. 51–67. 90

[188] K. Jamboti and P. Liggesmeyer, "A framework for generating integrated component fault trees from architectural views," in *IEEE Intl. Symp. on High-Assurance Systems Engineering (HASE)*, Oct. 2012, pp. 114–121. 90, 94

[189] J. Atlee and J. Gannon, “State-based model checking of event-driven system requirements,” *IEEE Trans. on Software Eng.*, vol. 19, no. 1, pp. 24–40, 1993. 92

[190] M. P. E. Heimdahl and N. G. Leveson, “Completeness and consistency in hierarchical state-based requirements,” *IEEE Trans. on Software Eng.*, vol. 22, no. 6, pp. 363–377, 1996. 92

[191] L. Grunske, “Early quality prediction of component-based systems – A generic framework,” *J. of Systems and Software*, vol. 80, no. 5, pp. 678–686, 2007, component-Based Software Engineering of Trustworthy Embedded Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121206002238 94

[192] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner, “Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure,” *Reliability Engineering & System Safety*, vol. 71, no. 3, pp. 229–247, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0951832000000764 94

[193] M. Wallace, “Modular architectural representation and analysis of fault propagation and transformation,” *Electr. Notes in Theor. Comp. Sci.*, vol. 141, no. 3, pp. 53–71, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066105051650 94

[194] P. Feiler and A. Rugina, “Dependability Modeling with the Architecture Analysis & Design Language (AADL),” DTIC Document, Tech. Rep., 2007. 96

[195] K.-K. Lau and Z. Wang, “Software component models,” *IEEE Trans. on Software Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007. 99, 100

[196] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, “The FRACTAL component model and its support in Java,” *J. of Software: Practice and Experience*, vol. 36, no. 11–12, pp. 1257–1284, Oct. 2006. 99

[197] S. Becker, H. Koziolek, and R. Reussner, “The Palladio component model for model-driven performance prediction,” *J. of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009. 99

[198] D. Birkmeier, “On component identification approaches: Classification, state of the art, and comparison,” *LNCS*, vol. 5582, pp. 1–18, 2009. 99

[199] S. Han, M. Kim, and H. S. Park, “Open software platform for robotic services,” *IEEE Trans. on Automation Science and Engineering*, vol. 9, no. 3, pp. 467–481, Jul. 2012. 99

[200] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078 101

[201] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N Degrees of Separation: Multi-dimensional Separation of Concerns," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*. New York, NY, USA: ACM, 1999, pp. 107–119. 103

[202] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, no. 3, pp. 10:1–10:42, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1670679.1670680 105

[203] M. Y. Jung and P. Kazanzides, "Fault detection and diagnosis for component-based robotic systems," in *IEEE Intl. Conf. on Technologies for Practical Robot Applications (TePRA)*. Woburn, MA, USA: IEEE, Apr. 2012. 118, 119, 170, 172, 280

[204] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. 126, 180, 200, 279, 282

[205] R. Smits and H. Bruyninckx, "Composition of complex robot applications via data flow integration," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May. 2011, pp. 5576–5580. 132

[206] J. S. Kang, D. U. Yu, and H. S. Park, "A robot software bridge for interconnecting OPRoS with ROS," in *Intl. Conf. on Ubiquitous Robots and Ambient Intelligence (URAI)*, Nov. 2012, pp. 296–297. 132

[207] BRIDE package in ROS. [Online]. Available: http://wiki.ros.org/bride 132

[208] I. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I. Shu *et al.*, "CLARAty: Challenges and steps toward reusable robotic software," *Intl. J. of Advanced Robotic Systems*, vol. 3, no. 1, pp. 023–030, 2006. 139

[209] D. Stewart, D. Schmitz, and P. Khosla, "The Chimera II real-time operating system for advanced sensor-based control applications," *IEEE Trans. on Systems, Man and Cybernetics*, vol. 22, no. 6, pp. 1282–1295, Nov. 1992. 140, 197

[210] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac, and A. Wellings, "GUARDS: a generic upgradable architecture for real-time dependable

systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 580–599, 1999. 140

[211] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath, "Cadena: An integrated development, analysis, and verification environment for component-based systems," in *Proc. of the 25th Intl. Conf. on Software Engineering*. IEEE, 2003, pp. 160–172. 140

[212] A. Casimiro, J. Kaiser, E. M. Schiller, P. Costa, J. Parizi, R. Johansson, and R. Librino, "The KARYON Project: Predictable and Safe Coordination in Cooperative Vehicular Systems," in *2nd Workshop on Open Resilient Human-aware Cyber-Physical Systems*, 2013. 140

[213] P. Hampton, "Survey of safety architectural patterns," in *Achieving Systems Safety: Proc. of the Twentieth Safety-Critical Systems Symposium*. Bristol, UK: Springer-Verlag London, Feb. 2012, pp. 137–158. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-2494-8_11 141

[214] K. Wika, "Safety kernel enforcement of software safety policies," Ph.D. dissertation, University of Virginia, May 1995. 155

[215] N. Leveson and T. Shimeall, "Safety assertions for process-control systems," in *Intl. Symp. on Fault Tolerant Computing*. Milan: IEEE, Jul. 1983, pp. 236–240. 155

[216] N. Leveson, T. Shimeall, J. Stolzy, and J. Thomas, "Design for safe software," in *American Institute for Astronautics and Aeronautics Space Sciences Meeting*. Reno, Nev.: AIAA, 1983. 155

[217] J. Ames, S. R., M. Gasser, and R. R. Schell, "Security kernel design and implementation: An introduction," *Computer*, vol. 16, no. 7, pp. 14–22, Jul. 1983. [Online]. Available: http://dx.doi.org/10.1109/MC.1983.1654439 155

[218] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, Jan-Feb 2004. 157, 282

[219] C. Henry, "The *Boost* C++ Libraries: Meta State Machine (MSM)." [Online]. Available: http://www.boost.org/doc/libs/1_57_0/libs/msm/doc/HTML 164

[220] T. Veldhuizen, "Using C++ template metaprograms," *C++ Report*, vol. 7, no. 4, pp. 36–43, May. 1995. 164

[221] 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*, vol. 9, pp. 25–40, 1989. 176, 177

[222] P. Kazanzides, A. Deguet, and A. Kapoor, "An architecture for safe and efficient multi-threaded robot software," in *IEEE Intl. Conf. on Technologies for Practical Robot Applications (TePRA)*. IEEE, Nov. 2008, pp. 89–93. 185, 278, 280, 285

[223] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-driven documents," *IEEE Trans. on Visualization & Comp. Graphics*, 2011. 192

[224] D. B. Stewart and P. K. Khosla, "Mechanisms for detecting and handling timing errors," *Commun. ACM*, vol. 40, no. 1, pp. 87–93, Jan. 1997. [Online]. Available: http://doi.acm.org/10.1145/242857.242883 196, 197, 198

[225] M. Caccamo, G. Buttazzo, and L. Sha, "Handling execution overruns in hard real-time control systems," *IEEE Trans. on Computers*, vol. 51, no. 7, pp. 835–849, Jul. 2002. 197

[226] O. M. dos Santos, "Run time detection of timing errors in real-time systems," Ph.D. dissertation, University of York, Oct. 2008. 197

[227] R. H. Taylor, B. D. Mittelstadt, H. A. Paul, W. Hanson, P. Kazanzides, J. F. Zuhars, B. Williamson, B. L. Musits, E. Glassman, and W. L. Bargar, *An Image-Directed Robotic System for Precise Orthopaedic Surgery*, ser. Computer-Integrated Surgery. Cambridge, MA: MIT Press, 1996, ch. 28, pp. 379–396. 219

[228] P. Kazanzides, P. W. Cain, and H. A. Wasti, "Distributed architecture for a fail-safe robot system," in *Proc. of the Signal Processing Applications Conference & Exhibition (DSPx)*, San Jose, CA, Mar. 1996. 219, 220, 225

[229] B. D. Mittelstadt, P. Kazanzides, J. F. Zuhars, B. Williamson, P. Cain, F. Smith, and W. L. Bargar, *The Evolution of a Surgical Robot from Prototype to Human Clinical Use*, ser. Computer-Integrated Surgery. Cambridge, MA: MIT Press, 1996, ch. 29, pp. 397–407. 219, 220

[230] W. Hanson, H. Paul, W. Williamson, and B. Mittelstadt, "Orthodock - an image driven orthopaedic surgical planning system," in *Proc. of the Twelfth Ann. Intl. Conf. of the IEEE Eng. in Medicine and Biology Society*, Nov. 1990, pp. 1931–1932. 220

[231] M. Y. Jung and P. Kazanzides, “Calibration of Kinematic Parameters of the ROBODOC® System,” Department of Computer Science, Johns Hopkins University, Tech. Rep., Oct. 2014. 227

[232] ——, “Calibration of Tool Parameters (Mini-Cal) in the ROBODOC® System,” Department of Computer Science, Johns Hopkins University, Tech. Rep., Oct. 2014. 227

[233] *JR3 DSP-Based Force Sensor Receivers - Software and Installation Manual*, JR3, Inc., 1993-94. 234, 251

[234] M. Hayashibe, N. Suzuki, and Y. Nakamura, “Laser-scan endoscope system for intraoperative geometry acquisition and surgical robot safety management,” *Medical Image Analysis*, vol. 10, no. 4, pp. 509–519, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1361841506000156 239

[235] K. Olds, A. T. Hillel, E. Cha, M. Curry, L. M. Akst, R. H. Taylor, and J. D. Richmon, “Robotic endolaryngeal flexible (Robo-ELF) scope: A preclinical feasibility study,” *The Laryngoscope*, vol. 121, no. 11, pp. 2371–2374, 2011. [Online]. Available: http://dx.doi.org/10.1002/lary.22341 255

[236] K. Olds, A. Hillel, J. Kriss, A. Nair, H. Kim, E. Cha, M. Curry, L. Akst, R. Yung, J. Richmon, and R. Taylor, “A robotic assistant for trans-oral surgery: the robotic endo-laryngeal flexible (robo-elf) scope,” *J. of Robotic Surgery*, vol. 6, no. 1, pp. 13–18, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11701-011-0329-9 255

[237] K. Olds *et al.*, “Robo-ELF Software Description,” in *Robo-ELF FDA Submission Document Package (Q130545: Study Determination for the Proposed Study titled, “Robotic Endolaryngeal Flexible (Robo-ELF) Scope.”).* 256, 258

[238] ——, “Robo-ELF FMEA,” in *Robo-ELF FDA Submission Document Package (Q130545: Study Determination for the Proposed Study titled, “Robotic Endolaryngeal Flexible (Robo-ELF) Scope.”).* 257

[239] P. Kazanzides, Z. Chen, A. Deguet, G. Fischer, R. Taylor, and S. DiMaio, “An Open-Source Research Kit for the da Vinci® Surgical System,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Jun. 2014. 274, 278

[240] T. Xia, S. Leonard, A. Deguet, L. Whitcomb, and P. Kazanzides, "Augmented reality environment with virtual fixtures for robotic telemanipulation in space," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2012, pp. 5059–5064. 277

[241] The *cisst* package. (Accessed: 2013-11-07). [Online]. Available: http://www.cisst.org/cisst 277

[242] A. Kapoor, A. Deguet, and P. Kazanzides, "Software components and frameworks for medical robot control," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2006, pp. 3813–3818. 277

[243] P. Kazanzides, S. DiMaio, A. Deguet, B. Vagvolgyi, M. Balicki, C. Schneider, R. Kumar, A. Jog, B. Itkowitz, C. Hasser, and R. Taylor, "The Surgical Assistant Workstation (SAW) in minimally-invasive surgery and microsurgery," in *MICCAI Workshop on Systems and Architecture for Computer Assisted Interventions (SACAI)*, Midas Journal, Sep. 2010. [Online]. Available: http://hdl.handle.net/10380/3179 278

[244] 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)*, Oct. 2010, pp. 6107–6112. 282, 286

[245] M. Shapiro, "Structure and encapsulation in distributed systems : the proxy principle," in *Intl. Conf. on Distributed Computing Systems (ICDCS)*, 1986, pp. 198–204. 282

[246] Y. Kouskoulas, F. Ming, Z. Shao, and P. Kazanzides, "Certifying the concurrent state table implementation in a surgical robotic system," in *Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability*, Chicago, IL, Apr. 2011. 286

Vita

Min Yang Jung received his B.S. in Electrical Engineering and the M.S. in Biomedical Engineering from Seoul National University (South Korea), and the M.S.E. degree in Computer Science from the Johns Hopkins University. Prior to JHU, he worked in industry in the area of high-performance networked systems and memory-centric database management systems. He is currently a Ph.D. candidate in Computer Science at JHU advised by Peter Kazanzides in the ERC CISST/LCSR. During his Ph.D., his research on safety is supported by collaborative research projects with THINK Surgical, Inc. (Fremont, CA, USA). He is one of the main contributors to the *cisst* open source component-based framework, which is used for various medical robotics research projects. His research interests include safety, software frameworks for robotics, software engineering and system integration for computer-integrated surgery, and real-time robot control systems.