An Architectural Approach to Safety of Component-based Robotic Systems

Min Yang Jung and Peter Kazanzides

Abstract— **Medical and surgical robot systems are examples of safety-critical systems due to the potential hazards that can lead to severe injury or the loss of human life. Furthermore, robot safety is becoming increasingly important in other domains as robots begin to share their workspace with humans. At the same time, the complexity of robot hardware and software has been increasing to endow them with capabilities for safe human-robot collaboration and other demanding tasks. While component-based frameworks have been widely adopted in robotics to manage the increasing scale and complexity of these systems, it is still challenging for both academic researchers and industrial developers to develop robot systems with safety in a reusable and structured manner. To address these issues, we 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. Specifically, we present a state-based model and a safety-oriented software architecture that allow us to reuse the design of safety features across different systems and different applications without relying on a particular component model. To demonstrate and evaluate the proposed methods, we built SAFECASS, an open source software framework that enables reuse of experience and knowledge on safety, and applied it to two medical robot systems, one of which is presented in this paper.**

I. Introduction

As human-robot collaboration enters the robotics mainstream, increasing attention has been focused on safety properties and mechanisms. This is because human-robot collaboration requires the human to enter the workspace of a potentially dangerous robot, so safety must generally rely on software for sensing and control, rather than on hardware mechanisms that either prevent humans from entering the workspace or shut down the robot when humans approach.

Medical robotics is an application domain with a long history of human-robot collaboration. Since an industrial robot was first used as a surgical tool holder in 1985 [1], a variety of medical robot systems have been developed both in academia and in industry. Various commercially available robot systems are also being used in the modern operating room in various surgical areas [2]. Within medical robotics, safety has been perceived as a crucial system property since the early days. Pioneering work with focus on safety is found in the literature as early as 1991 (Taylor, 1991 [3]; Kazanzides, 1992 [4]; Davies, 1993 [5] and 1996 [6]; Taylor, 1996 [7]; Fei, 2001 [8]). However, the majority of prior work on safety in medical robotics have been system- or application-specific, with little consideration on what constitutes safety and how to systematically represent its design.

Authors are with the Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA. Min Yang Jung is now with THINK Surgical Inc., Fremont, CA 94539, USA, and Peter Kazanzides can be contacted at pkaz@jhu.edu.

Despite these early efforts and the recognition of the importance of safety, system designers still face the following issues [9]: (1) inability to systematically capture the knowledge and experience with safety of medical robot systems, and (2) significant amount of engineering effort to build medical robot systems that conform to medical device standards such as IEC-60601 and IEC-62304. In our prior work [10], we addressed the first challenge by proposing the Safety Design View, a conceptual framework that can systematically capture and describe the design of safety features. Still, the engineering aspect of medical robot systems is a practical issue and is becoming even more challenging as the scale and complexity of medical robot systems increase to achieve both challenging functional and non-functional system requirements.

In robotics, component-based software engineering (CBSE) has been widely adopted as an effective solution to this problem [11], [12]. Numerous open source packages have been developed to facilitate robotics research both in general robotics (e.g., Robot Operating System (ROS) [13] and Orocos [14]) and in medical robotics (e.g., *cisst* [15]). A few of them address the safety aspect of robot systems by providing error handling mechanisms (e.g., Orocos) or error management facilities (e.g., OPRoS [16]), but to the best of our knowledge, none of them provide a complete framework that supports the development and implementation of safety mechanisms across the entire system.

Meantime, various approaches to achieving safety have been proposed, such as integration with safety standards [17] and application of formal methods to the verification of safety properties of robot systems [18], [19] (a more comprehensive survey on safety in various domains is presented in [9]). Among those methods, our focus is on software architectural approaches. Specifically, this work proposes a safety-oriented software framework that (1) enforces a particular structure that can represent the design of safety features in a structured and systematic manner, and (2) provides the system designers with a set of component model-independent tools for designing and deploying safety features. The idea of (re)use of architectures with a set of tools is conceptually similar to that of the HyperFlex toolchain [20].

We first describe two key elements of our design approach in Sec. II: *framework independence* to address the diversity of component models in robot software frameworks, and *safety design decomposition* to reuse the design of safety features. The former necessitates a generic abstraction of various component models (Sec. III), and the latter is enabled within a safety-oriented layered architecture (Sec. IV). Based on the proposed methods, we built an open source framework, called *Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)*. To demonstrate the concept and feasibility of the proposed methods, Sec. V presents an illustrative example using a force sensor-based safety feature, followed by a brief overview of the application of SAFECASS to a research version of a commercial medical robot system.

II. Design Approaches

We attempt to design and implement a safety-oriented software architecture that can effectively address those challenges mentioned earlier. Our fundamental design principle that governs the overall design of the architecture is *reusability*. This is primarily due to the two aforementioned issues: a proliferation of component models in robotics, i.e., numerous robot software frameworks, and an inability to reuse experience and knowledge with safety design in a systematic and structured manner. Thus, it would be desirable if this work (1) can be applicable to various component-based frameworks in the same manner, and (2) allow us to capture, represent, and reuse the design of safety features across different applications, systems, and component-based frameworks. These considerations lead to the two design approaches: *framework independence* and *safety design decomposition*.

A. Framework Independence

We seek to design a “stand-alone” software framework that is not dependent on any particular component-based framework, so that it can be reused by different frameworks. To accomplish this goal, we separate the *framework-specific* parts from the *framework-independent* parts that maintain the key information about all the components in the system, thereby allowing us to define and represent the system status solely based on those key information. As shown in Fig. 1, this necessitates (1) a *framework extension* within the framework that provides framework-specific functionalities for the framework-independent part, and (2) *generic APIs* through which component frameworks can access the key information maintained by the framework-independent part. The key element of this design is the *framework-independent part* that enables us to represent the structure and operational status of component-based systems in a systematic and component model-independent manner. This abstraction comprises two elements: the *structural elements* that are an abstraction of various component models, and the *semantics* that systematically captures and represents the operational status of components. Sec. III presents further details.

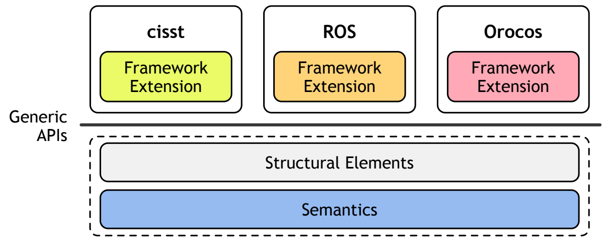


Fig. 1: Reusable and framework-independent design; framework extension for *cisst* has been implemented; framework extensions for ROS and Orocos are not yet implemented.

B. Safety Design Decomposition

Another approach to accomplish the reusability of safety features is the decomposition of safety features. Essentially, we decompose a safety feature into a *reusable mechanism* and a *configurable specification*, and handle each part separately within the proposed architecture. Fig. 2 illustrates this concept. 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. Another benefit 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 behavior is correct in terms of outputs. In Fig. 2, this is represented as *safety feature testing*. Once safety features are decomposed into mechanisms and specifications, we can test each part individually (e.g., unit tests on the mechanism to verify the correctness, validation of consistency of the specification). These concepts are illustrated as *mechanism testing* and *specification validation* in Fig. 2. 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 (as represented by *safety feature testing*). An illustrative example that demonstrates this approach is described in Sec. V.

III. Model: Generic Component Model

We first define a generic, abstract component model that allows us to present any component-based system without delving into the details of a particular component model or implementation. This model is called the *Generic Component Model (GCM)*. The GCM comprises (1) the *structural*

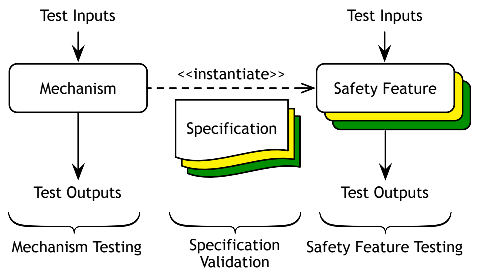


Fig. 2: Decomposition and testing of safety features: With the same safety mechanism, different safety features can be defined and deployed by using different safety specifications. This decomposition of safety features allows us to test mechanisms and specifications separately, thereby enabling more extensive, modular, and verifiable testing processes.

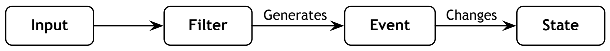


Fig. 3: Internal processing pipeline of Generic Component Model

elements and (2) the *semantics* to represent the *operational status* of component-based robot systems in an explicit, structured, and component model-independent manner.

Based on a widely accepted definition by Szyperski [21], the structural elements are defined as a set of a *component*, a *provided interface* and a *required interface*. With these structural elements, the semantics, specifically *state-based* semantics, systematically captures and represents the operational status of components with support for error propagation. In essence, the semantics is represented by the *internal processing pipeline* (Fig. 3) where *inputs* from other components or from the environment are processed by *filters*, which generate *events* that may possibly initiate *state* changes.

To represent the operational status of the system based on different component models, we need a generic and expressive abstraction of the run-time information of the system so that we can capture and describe various properties of the system in a comprehensive and consistent manner. Our approach to achieving expressiveness and generality is to use *states* as a means of abstraction. The GCM defines three states that are adapted from the dependability formalism [22] of the dependable computing domain: *Normal*, *Warning*, and *Error*. *Normal* (N) is an initial state where no threat is present and the system is operating correctly. *Warning* (W) 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). In this state, the system can attempt to proactively prevent errors by reacting to events (e.g., fault prevention using prognostic information). *Error* (E) is a state where the delivered service deviates from the correct service. With these three states and transitions between them, we define the *GCM state machine*, which is an event-driven finite-state machine.

The *state variable* indicates the current state of the GCM state machine. We define a set of state variables and associate them with each key element within the component so that a complete set of state variables represents the instantaneous status of the entire system. These key elements include the GCM structural elements, the processing algorithm and logic of the application, and the services provided by component frameworks, as illustrated in Fig. 4.

We also define the *state product operator*, $\otimes$, that, given two states, chooses the one that is more "severe" (the severity

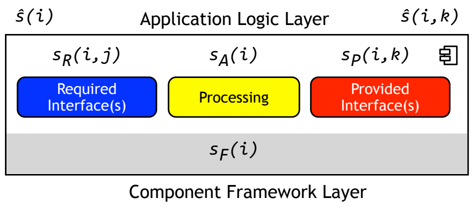


Fig. 4: GCM state variables for component i, required interface j, and provided interface k.

increases in the order of *Normal*, *Warning*, and *Error*). This operator is used to "summarize" multiple states into one state at any level in the system (e.g., system-, component-, and interface-level), and plays a key role in error propagation from one component to another. One special state variable, called the *extended component state*, $\hat{s}_{ext}(i)$, is defined to consolidate all states within a component into a single state.

The GCM *event* initiates state changes. System designers define GCM events after performing a safety analysis that identifies potential hazards of the system. Events are associated with hazards and their attributes, such as severity and possible transitions due to the event, are determined based on the results of the safety analysis as well as the system designer's knowledge and experience.

The GCM events are generated by the GCM *filter*. A filter is an abstract unit of computation that can represent any arbitrary algorithm and can generate GCM events as a result of computation. Generated events are dispatched to a specified state machine and may initiate state transitions. Also, filters can be cascaded into a filter pipeline to implement more complex filtering algorithms. The filter is based on the filter mechanism for the fault detection and diagnosis of component-based robotic systems [23].

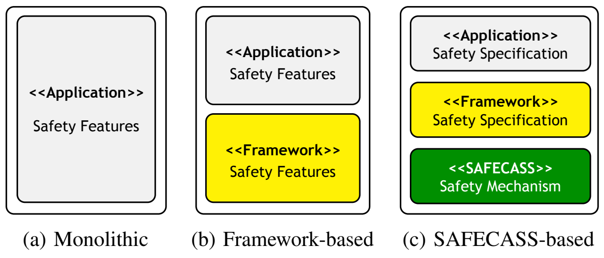
More detailed description and design rationale of the GCM is available in [9].

IV. Architecture: SAFECASS

With our design approaches described in Sec. II, we designed and built a software framework called *Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)*. This framework aims to provide a run-time software environment based on the GCM, and allows us to dynamically verify the design of the GCM and evaluate its effectiveness.

A. Architecture

Fig. 5 shows different architectural styles with which safety features can be implemented. The first style is the *monolithic* architecture (Fig. 5a), where safety features are "embedded" in the system along with other functional elements 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. 5b) that represents typical component-based systems.



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

Fig. 5: Evolution of the architectures of robot systems in terms of safety

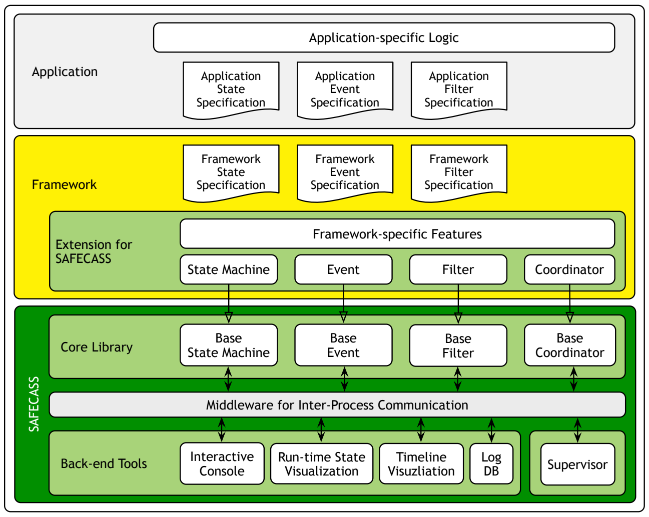

Application-specific Logic
Application
Application State Specification
Application Event Specification
Application Filter Specification
Framework
Framework State Specification
Framework Event Specification
Framework Filter Specification
Extension for SAFECASS
Framework-specific Features
State Machine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

Fig. 6: Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS)

A system with this architecture consists of two layers: the *framework layer* and the *application layer*. Although each layer maintains its own safety features, this structure is similar to the monolithic architecture from the safety perspective, in that safety features are still embedded in the system. The third style is what we propose, called the *SAFECASS-based* architecture. As depicted in Fig. 5c, 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.

Fig. 6 shows a more detailed view of the architecture of SAFECASS. The SAFECASS layer at the bottom is mainly comprised of three parts: (1) the *core library* that provides a code-level (C++) implementation of the internal processing pipeline of the GCM and its elements in a form of the *base mechanisms* (i.e., base state machine, base event, base filter, base coordinator), (2) the *middleware* that enables inter-process communication within SAFECASS, and (3) the *back-end tools* that enable access (read/write) to the internal processing pipeline of the GCM, thereby allowing the system designers to easily inject faults into the system, manually generate events, and introspect the system status in terms of the GCM elements. SAFECASS relies on IceStorm [24], a publish-subscribe event distribution service from ZeroC, as its middleware.

The framework layer enables a component-based environment for the system, which is typically provided by an existing component-based framework (e.g., *cisst*, ROS, Orocos). This layer is augmented by the *extension for SAFECASS* that provides (1) a set of *framework-specific mechanisms* derived from the core library of the SAFECASS layer (i.e., state machine, event, filter, coordinator), and (2) the *framework-specific features* required by SAFECASS. Depending on the framework, the base mechanisms may be directly used without specialization. The framework-specific features represent "glue" code that provides framework-specific services required by SAFECASS, such as SAFECASS bootstrapping code and access to internal data structures of the framework. Currently, we use the *cisst* package [15] as a component-based framework, and implemented the *cisst* extension for SAFECASS.

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 the SAFECASS layers – provide most of the necessary mechanisms. Typically, this layer contains components that implement application-specific logic and the functional behavior of the system.

Both the framework and the application layers define the specifications of framework- and application-specific safety features. SAFECASS uses these software artifacts – currently, configuration files in the JSON format – to deploy and configure framework- and application-specific safety features. Once enabled, SAFECASS not only provides the system designers with a means to construct a system with deterministic and structured behaviors at *design-time*, but also enables systematic event handling and testing of safety features at *run-time*.

B. Tool Support

SAFECASS provides tools to facilitate the development process by enabling direct access to the internal processing pipeline of the GCM. Because these tools represent the entire system in terms of the GCM, they have no dependency on the underlying component-based framework. This self-contained and GCM-based design allows these tools to be used with different frameworks in the same manner. Currently, SAFECASS supports three tools: *console* (an interactive terminal-based utility for system introspection and manipulation), *viewer* (a run-time state visualization tool), and *timeline* (a state transition history visualization tool). The services that these tools provide are essentially read and write operations on the internal processing pipeline of the GCM. They include: fault injection into different levels of the system (i.e., "deep" and "shallow" fault injections), manual generation of events, and introspection of the system status via visualization. Figures 7 and 8 show screenshots of the *viewer* and *timeline* tools, respectively, for the research ROBODOC system presented in Section V-B.

V. Illustrative Examples

To illustrate our methods, we first present a simple but representative safety feature (force sensor safety check), followed by application to a medical robot system for orthopaedic surgery.

A. Simple Example

Fig. 9 shows the typical design of a force sensor safety check, where an error event is generated when the measured force exceeds a pre-defined threshold. Despite its simple

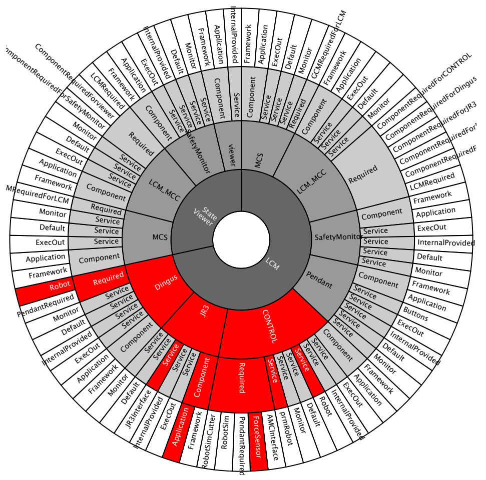


Fig. 7: SAFECASS *viewer* tool provides run-time visualization of states (screenshot from research ROBODOC System)

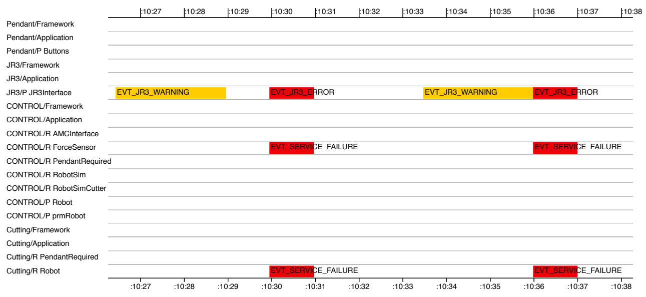


Fig. 8: SAFECASS *timeline* tool provides visualization of state change history (from research ROBODOC System)

design, this is one of the most widely used safety features in the medical robotics domain [9].

Fig. 10 depicts the new design of the example, based on SAFECASS. The major design changes are represented in the yellow box: (1) the introduction of the internal processing pipeline of the GCM, and (2) a SAFECASS artifact in the JSON format that specifies the parameters of the GCM elements. 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 – by either physical force or faults injected by SAFECASS – is detected. It initiates a state transition from *Normal* to *Error* and results in error propagation to other connected components. The state-based semantics of the GCM governs this sequence of actions, and thus no additional code from the developer is required for, for example, event propagation or state transitions. Although SAFECASS internally handles these actions, SAFECASS also allows the system designers to perform custom actions in response to the original actions by providing APIs to register custom callback functions. It should be noted that this design only relies on the GCM and thus is not dependent on a particular component model.

Read and write operations that the SAFECASS tools provide 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 or

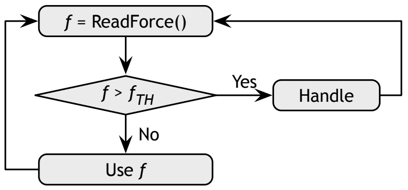


Fig. 9: Simple example: force sensor-based safety feature. Force feedback, f, is read from the force sensor and is checked against the pre-defined threshold, f_{TH}. Typically, an error event is generated if f exceeds f_{TH}.

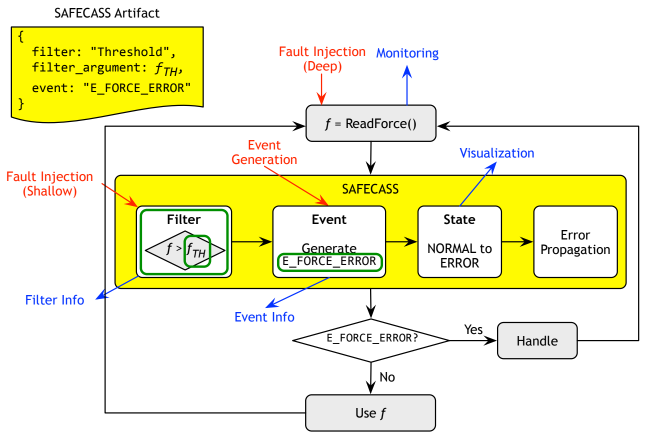


Fig. 10: Simple example implemented with SAFECASS.

behavior of the safety feature. In this example, some of these parameters are shown: a filter type (“Threshold”), a threshold value (“f_{TH}”), and a name of the event (“`E_FORCE_ERROR`”). A simplified SAFECASS artifact of this example is partially shown below:

```
{   "component": "Force",
    "filter": [
        {  "type": "Threshold",
           "argument": {
               "threshold": 5.0,
               "event_onset": "E_FORCE_ERROR" } } ],
    "event": [
        {  "name": "E_FORCE_ERROR",
           "severity": 20,
           "state_transition": [ "N2E", "W2E" ] } ],
    "service": [ ... ]
}
```

B. Case Study: Research ROBODOC® System

To empirically evaluate the benefits and effectiveness of the developed methods, we applied SAFECASS to two medical robotics research systems [9]. This section describes the application of SAFECASS to one of them: a research version of the ROBODOC® System for orthopaedic surgery.

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. As part of a collaborative research project with the manufacturer, we have one research ROBODOC system and have full access to the source code of the original product software. We created the research system by adapting the commercial system software to the *cisst* component-based framework, which required an architectural style change from the *monolithic* architecture to the *framework-based* architecture. We then introduced the SAFECASS layer to the system by (1) decomposing safety features into reusable mechanisms (i.e., filters) and JSON specifications, and (2) performing code-level refactoring to benefit from SAFECASS services. With these changes, the system achieved the *SAFECASS-based* architecture.

Fig. 11 depicts a simplified structure of the research ROBODOC system. The three components refactored for the case study are presented in yellow. This section focuses on the `CONTROL` component that implements the main control loop of the system. This component coordinates other components in the system and controls the overall system behaviors, including safety features.

The first step is to define a JSON specification to register the component to SAFECASS. This creates a "shadow copy" of the component based on the GCM, thereby enabling SAFECASS to maintain and manage the component's run-time status. For illustration purposes, we continue the analysis of the force-based safety check (implemented in the lower-level force sensor component, named `JR3` in Fig. 11) and follow its propagation to the `CONTROL` component. The JSON configuration for the `CONTROL` component is partially shown as follows:

```
{   "component": "CONTROL",
    "event": [
        {  "name" : "EVT_CONTROL_FORCE_FREEZE",
           "severity" : 20,
           "state_transition": [ "N2E", "W2E" ] },
        // More event definitions
    ],
    "service": [
      // Service state dependency for provided interface
    ]
}
```

This specification captures (when refactoring an existing system) or defines (when designing a new system) what can happen within this component (items under `"event"`), how each event affects the run-time status (`"state_transition"`), and how state transitions impact the run-time states of other components (items under `"service"`; used for error propagation, omitted here for brevity).

The next step is to update the structure of the `CONTROL` component to take advantage of the state-based semantics of the GCM. Below is a highly simplified code of the *original* control loop of the component (`Run()` method) with focus on the force-based safety feature:

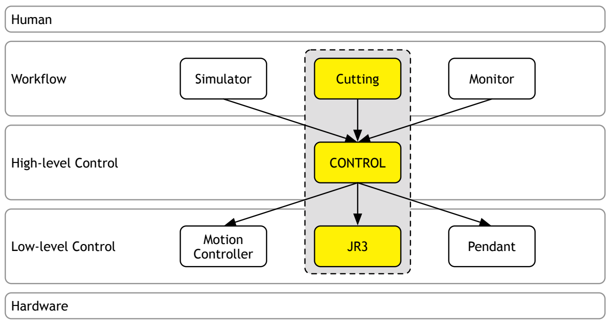


Fig. 11: Simplified structure of the research ROBODOC system. The three components refactored for the case study are presented in yellow.

```
void ControlTask::Run(void)
{
  ...
  // Read force feedback
  if (no_force_sensor_error)
    ReadForceFeedback();

  // Check excessive force
  if (force > force_threshold) {
    err |= excessive_force;
    // excessive force handling
  }
  ...
  // Send next goal position to low-level controller
  if (err == 0)
    Servo.LoadPVT(pvt);
}
```

To enable SAFECASS, we modify the structure of this control loop as follows:

```
void ControlTask::RunNormal(void)
{
  ...
  // Read force feedback
  ReadForceFeedback(); // Without error check

  // Check excessive force
  if (force > force_threshold) {
    SC::Generate("EVT_CONTROL_FORCE_FREEZE");
    this->RunError(SC::GetEvent("CONTROL"));
    return;
  }
  ...
  // Send next goal position to low-level controller
  if (NORMAL || WARNING) // State-based
    Servo.LoadPVT(pvt);
}

void ControlTask::RunWarning(const SC::Event * e)
{
  ...
  RunNormal();
}

void ControlTask::RunError(const SC::Event * e)
{
  // error handlers for each event defined in JSON
  ...
  ON_EVENT("EVT_CONTROL_FORCE_FREEZE") {
    // excessive force handling
  }
}
```

The key change is that the original `Run()` method is split into three methods: `RunNormal()`, `RunWarning()`, and `RunError()`, each explicitly representing *Normal*, *Warning*, and *Error*. Only one of these methods is executed at each iteration, depending on the current extended component state, $\hat{s}_{ext}(i)$ (Sec. III). This scheme eliminates the force sensor error check in the original implementation because SAFECASS propagates errors from the force sensor component (`JR3`) to the `CONTROL` component, thereby changing $\hat{s}_{ext}(i)$ to E. In case of error handling, it is sufficient for the component to generate `EVT_CONTROL_FORCE_FREEZE` on the detection of excessive force, after moving the error handling code from `Run()` to `RunError()`. This allows us to control the behavior of the component based on the GCM states, rather than application-specific error codes. As a result, error handling becomes simpler and more structured, thereby increasing testability, readability, and verifiability of this safety feature. The SAFECASS tools described in the previous section (Sec. IV-B) can also be used for this case. For example, using *console* at run-time, we can directly generate `EVT_CONTROL_FORCE_FREEZE` in order to test if state transitions are correct and the event is properly handled in a timely manner. Or, the same testing can be performed by indirectly generating the event, i.e., by changing force feedback values via *console* (fault injection). During these tests, *viewer* (Fig. 7) and/or *timeline* (Fig. 8) can be used for verification.

VI. Conclusion

This work presented an open source software framework, Safety Architecture for Engineering Computer-Assisted Surgical Systems (SAFECASS), that supports the development and implementation of safety mechanisms by enabling reuse of knowledge and experience on safety. Essentially, SAFECASS aims to improve reusability of the design of safety features. The two key design concepts of SAFECASS are framework independence and safety design decomposition. SAFECASS relies on the generic component model (GCM) to represent the design of safety features. The GCM defines the internal processing pipeline that is comprised of inputs, filters, events, and states. All components and their structural elements (i.e., provided and required interfaces) in the application are represented by instances of the GCM within SAFECASS. These "proxy" objects maintain the safety states of the application components, thus avoiding the need to modify the component-based framework used within the application. Safety design decomposition facilitates reuse of safety features by providing generic safety mechanisms in the SAFECASS layer that can be configured from the application and framework layers via JSON format configuration files. SAFECASS also provides tools to access the internal processing pipeline, thereby enabling services for the system designers, such as system introspection, fault injection, event generation, and run-time system status visualization. To the best of our knowledge, no existing framework in robotics provide a comparable architecture and tools to facilitate the design and development process for safety features.

Several different directions are possible extensions of this work. One is to expand the list of use cases (i.e., applying SAFECASS to other robot systems based on other software frameworks, such as ROS or Orocos). Another is to develop automated run-time safety analysis and verification tools based on the GCM and SAFECASS. In addition, it is possible to apply formal methods to the state-based semantics of the GCM and the implementation of SAFECASS, thereby formally verifying the correctness and completeness of both the GCM and SAFECASS.

Acknowledgment

We thank Russell Taylor and Scott Smith for informative discussions and valuable feedback on this work. We also thank THINK Surgical, Inc. for their support.

References

[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.

[2] R. A. Beasley, "Medical robots: Current systems and research directions," *J. Robotics*, vol. 2012, pp. 1–14, 2012, article ID 401613.

[3] 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)*, vol. 1, Jun. 1991, pp. 865–870.

[4] 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.

[5] B. Davies, "Safety of medical robots," in *Intl. Conf. on Advanced Robotics (ICAR)*, vol. 11, 1993, pp. 311–317.

[6] B. L. Davies, *A discussion of safety issues for medical robots*, ser. Computer-Integrated Surgery. Cambridge, MA: MIT Press, 1996, pp. 287–298.

[7] R. H. Taylor, *Safety*, ser. Computer-Integrated Surgery. Cambridge, MA: MIT Press, 1996, pp. 283–286.

[8] 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.

[9] M. Y. Jung, "State-based Safety of Component-based Medical and Surgical Robot Systems," Ph.D. dissertation, The Johns Hopkins University, Baltimore, Maryland, USA, May. 2015.

[10] 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, pp. 1883–1888.

[11] D. Brugali and P. Scandurra, "Component-Based Robotic Engineering (Part I)," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, Dec. 2009.

[12] 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.

[13] 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.

[14] 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.

[15] 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)*, vol. 5, no. 2, pp. 25–41, Sep. 2014.

[16] 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.

[17] 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, Workshop on safety in human-robot coexistence & interaction*, 2012.

[18] 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.

[19] 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.

[20] L. Gherardi and D. Brugali, "Modeling and reusing robotic software architectures: the HyperFlex Toolchain," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2014, pp. 6414–6420.

[21] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.

[22] 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 Comp.*, vol. 1, pp. 11–33, Jan. 2004.

[23] M. Y. Jung and P. Kazanzides, "Fault detection and diagnosis for component-based robotic systems," in *IEEE Intl. Conf. on Tech. for Practical Robot Applications (TePRA)*, Woburn, MA, USA, Apr. 2012.

[24] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, Jan-Feb 2004.