Journal of Software Engineering for Robotics 5(2), September 2014, 25-41
ISSN: 2035-3928

Lessons Learned from the Development of Component-Based Medical Robot Systems

Min Yang JUNG[1] Marcin BALICKI[1] Anton DEGUET[1] Russell H. TAYLOR[1] Peter KAZANZIDES[1]
[1] Department of Computer Science, Johns Hopkins University, Baltimore, Maryland 21218, USA

Abstract—Over the past two decades, a variety of medical robot systems have been developed and commercial products are actively used in modern operating rooms. These systems often comprise a variety of sensors and devices with real-time constraints, and the size and complexity of these systems have significantly increased to achieve both desired medical functionality and non-functional requirements such as safety, integrity, and reliability. Medical robot systems, as in other robotics domains, have reached a point that requires careful consideration of strategies for more effective and maintainable system development or rapid prototyping. This paper describes practical problems and challenges that we have faced throughout the development of several medical and surgical robot systems, using a robotic system for retinal microsurgery as a representative example. We created a component-based software framework, called *cisst*, with a collection of reusable components, known as the Surgical Assistant Workstation (SAW), to support these development efforts. The main contribution of this paper is the discussion of our design approaches (best practices), which are summarized as lessons learned. Although our main focus has been on medical robot systems, these best practices are not specific to this domain and can be shared with the robotics community.

Index Terms—component, component-based software engineering, medical robotics, surgical robotics, *cisst*, Surgical Assistant Workstation, best practice

1 INTRODUCTION

Robot systems are information intensive systems in that a variety of sensor data are acquired, processed, and exchanged within the system. The amount of sensor data typically increases if the workspace is unknown or unstructured, as is the case for most medical and surgical robot systems. These robots operate on or inside the patient's body (e.g., internal organs or structures), which move due to involuntary physiological functions such as heart beat or respiration, and unpredictable gross body motions. Anatomical structures such as bones and internal organs also vary in size and shape between patients, and these structures may change or significantly deform during surgery, especially when parts of them are resected, punctured, or sutured. These issues require medical robot systems to have diverse sensing capabilities that can capture real-time information about the surrounding environment, and to have a software system that can incorporate this feedback into an overall control architecture that often includes a human (surgeon) in the loop.

We have focused on medical robotics for over 25 years, starting with the development of the Robodoc® System[1] for orthopaedic surgery in the mid-1980s. These efforts significantly diversified into different medical domains in 1998, with the formation of the NSF Engineering Research Center for Computer-Integrated Surgical Systems and Technology (CISST ERC) at The Johns Hopkins University and other collaborating institutions. From the beginning, we recognized the importance of a system infrastructure to support this research area, and leveraged existing open source software, such as the 3D Slicer package (www.slicer.org) for medical image visualization developed by Brigham and Women's Hospital, one of the core partners in the CISST ERC. Within the robotics domain, we did not find a suitable open-source package and therefore initially developed the Modular Robot Controller (MRC) [2], [3]. MRC provided a basic abstraction of a robot control system in C++ by providing "wrappers" for off-the-shelf motor control hardware that performed the low-level servo control and adding higher-level capabilities such as kinematics and Cartesian control. MRC was useful for enabling integration of various robot systems with other devices and software (e.g., 3D Slicer), but did not provide a real-time

Regular paper – *Manuscript received November 11, 2013; revised September 6, 2014.*

- *This work was supported by NSF EEC 9731748, EEC 0646678, MRI 0722943, NRI 1208540, NASA NNX10AD17A, and NIH R01 EB007969.*

environment for low-level robot control. This motivated the creation of the *cisst* software package as a successor to MRC. One key requirement was to be able to support robotic systems of varying complexity, from systems that require hard-real-time performance for low-level servo control, to systems that have no real-time requirements (because they use external control hardware) but may require a specific platform, such as Microsoft Windows, due to device driver availability or to interface with other software systems.

This paper summarizes some of the key lessons that we learned during the development of the *cisst* package, which was driven by the needs of several different medical robot systems, some of which are shown in Table 1. We now take for granted the first lesson learned, which was that a component-based software approach was the most effective means to create a reusable software infrastructure [4]. As our component-based framework matured, we added the Surgical Assistant Workstation (SAW) [5]. SAW was conceived as a framework for telesurgical robotics research and was jointly developed by The Johns Hopkins University (JHU) and Intuitive Surgical, Inc., manufacturer of the da Vinci ® Surgical System. It has now become a repository for a diverse collection of reusable software components that are particularly suited to support research in medical robotics and computer-assisted surgery. As one of major use cases, we first present the *EyeSAW* system, a system for retinal microsurgery. This is followed by a brief overview of the *cisst* package, to provide sufficient background for the main contribution of this paper, which is a summary of some of the best practices we have learned over the years. Most of these best practices are represented by specific features in *cisst* and/or SAW. In many cases, these features were not designed in advance, but rather added in response to limitations in the initial design. Thus, they are best described as "lessons learned".

TABLE 1
Some application areas of *cisst* and/or SAW within medical and surgical robotics

Application Area	Authors
Neurosurgery	T. Xia *et al.* [6]
Laryngeal surgery	K. Olds *et al.* [7]
Cone-beam CT-guided surgery	A. Uneri *et al.* [8]
Ultrasound-guided beating heart surgery	P. Thienphrapa *et al.* [9]
Transoral robotic surgery	W. Liu *et al.* [10]
Retinal microsurgery	M. Balicki [11]
Minimally invasive surgery	P. Kazanzides *et al.* [12]

2 Robot System for Retinal Microsurgery

2.1 System Overview

The EyeSAW is an application-specific instance of SAW and is a representative system built using the *cisst* libraries. The Eye-

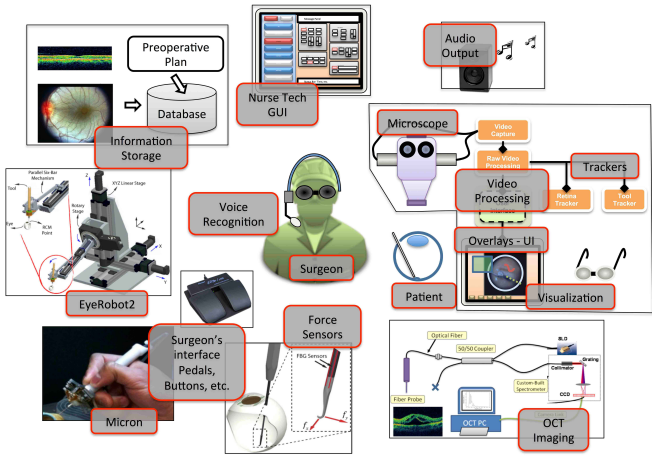

Fig. 1. Various devices and modules in EyeSAW

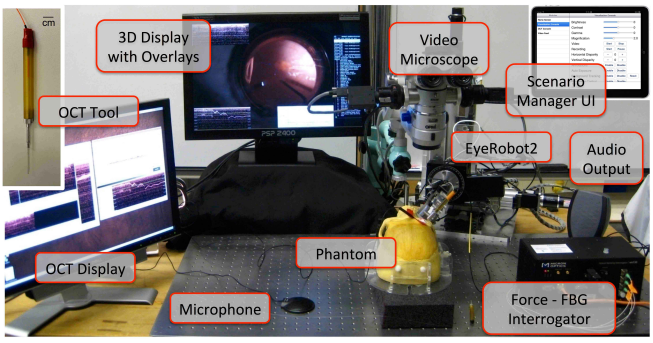

Fig. 2. EyeSAW hardware setup

SAW objective is to develop technologies and an associated system addressing many fundamental challenges in vitreo-retinal surgery [13]. A vitreoretinal surgeon uses an operating stereo microscope with free-hand instrumentation to execute technically demanding procedures to address sight-threatening conditions. The surgeons are faced with poor visual resolution and proximity sensing, physiological hand tremor, fatigue, no tactile feedback, and lack of real-time sensing of physiological parameters of the delicate retina. All of these factors contribute to extended operating times, attendant light toxicity, and higher than desired complication rates. We have developed novel technologies that address the particular limitations independently. However, together they can be combined into a modular, synergistic, and extendable system that enables computer-interfaced technology and information processing to work in partnership with surgeons to improve clinical care and enable novel therapeutic approaches. Our surgical workstation system comprises a stereo video-microscopy subsystem and a family of novel sensors, instruments, and robotic devices, as in Fig. 1. It is built on top of the *cisst* libraries, relying heavily on the component-based framework and real-time video library.

Fig. 2 presents an example hardware configuration that

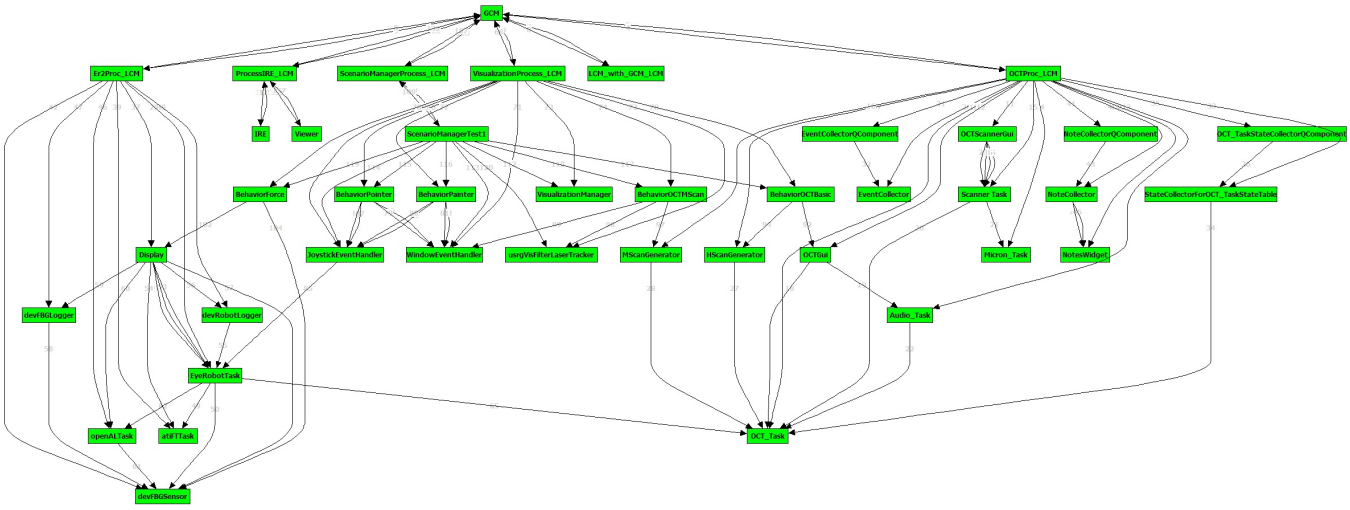

Fig. 3. Example configuration of EyeSAW System (generated by the *cisst* Component Viewer; see Sec. 5.5 for details). The actual configuration is more complex than this because this diagram only shows substantial components that are not involved in vision processing. Hidden components and connections include proxy components for inter-process communication over the network, vision processing components, and connections between them.

includes: (1) the SteadyHand EyeRobot, a cooperatively controlled microsurgical robot [14]; (2) an optical coherent tomography (OCT) subsystem that provides cross-sectional imaging and serves as a range finding sensor integrated into the shaft of the surgical instrument [15]; (3) a surgical console subsystem that captures, manipulates, and displays 3D stereo video from the surgical microscope [16]; and (4) a console or iPad application for the technician (shown as the "Scenario Manager UI"), which is used for configuring components to provide particular behaviors of the whole system. Each one of these subsystems, composed of one or more software components, runs on a separate machine due to hardware and software dependencies, specific computing requirements, different operating systems and also for convenience and portability reasons. For example, the OCT process requires specific data acquisition hardware and operates at 4 kHz on large data sets. Simultaneously, the surgical console visualization runs on a separate machine and processes up to around half a gigabyte of video data (30+ FPS). Although it would be feasible to build a single machine executing a single application that combines OCT processing and the visualization pipeline, the contention for processor resources and bus bandwidth would significantly affect video frame latency and frame rate, while degrading real-time OCT performance. Such compromise is not an option in high risk surgical applications. We have found that our surgeon colleagues can detect even a single frame delay (30ms).

The technician console provides a graphical user interface to the *Scenario Manager* (SM), which is responsible for configuring and monitoring the system's components for a particular surgical scenario, such as epiretinal membrane peeling. A scenario comprises a variety of system configurations, called *behaviors*, to assist the surgeon in executing specific steps to complete the surgical procedure.

Fig. 3 shows the components used for the *Robot Assisted B-Scan Behavior*, which produces a cross-sectional OCT image (B-Scan) that can be used by the surgeon to locate the starting point for a membrane peeling maneuver (this figure was generated at run-time by the interactive *cisst* Component Viewer). The B-Scan is performed by the robot automatically scanning an imaging probe, while keeping a constant surface offset distance (measured by the OCT). The OCT image is created and displayed as a picture-in-picture overlay on the surgical console. Once this step of the procedure is complete, the surgeon may request to switch to a different scenario or choose another behavior from the list of available behaviors in the current scenario.

2.2 System Requirements

The requirements of the EyeSAW system were primarily derived from surgical use cases that our surgeon colleagues identified. While many requirements were satisfied by the development of application-specific software components, there were several system requirements that drove the development of the *cisst* and SAW software infrastructure. These motivated the adoption of many of the best practices identified in Section 5 and are summarized below:

- **REQ 1**: Real-time imaging and robot control
- **REQ 2**: Flexible integration of various software components
- **REQ 3**: Run-time data collection and offline analysis
- **REQ 4**: Integration of different hardware platforms

- **REQ 5**: Interactive environment for prototyping and testing

The vitreoretinal surgeon relies heavily on visual information and this leads to one of the aims of the EyeSAW system, which is to augment the surgeon's visual perception and hand motion accuracy by using real-time vision and robotic assistance (**REQ 1**). The EyeSAW system requires multiple high-resolution real-time image streams (e.g., stereo microscope video and OCT) and these images must be acquired, processed, and displayed with minimal latency. The processing may provide information for robot control; for example, the OCT image can be used to measure the distance from the surgical instrument to the surface of the retina, as described above for the B-Scan behavior.

The system consists of a variety of modules and devices as shown in Fig. 1, and the optimal integration of those devices requires exploration of the design space, which is facilitated by support for flexible integration of components (**REQ 2**).

Data collection and offline analysis (**REQ 3**) can be valuable for any type of experiment, but is extremely important for medical systems that are used in animal or human clinical experiments. Time is a key factor in these experiments, since anesthesia is usually administered, and also the quality of the biological tissue can degrade with time. We have especially found this to be true with retinal tissue. Animal experiments are also expensive to perform due to the high cost for the animal preparation process and the amount of surgeon/student/engineer's time spent for performing the experiments. More importantly, animals are sacrificed after each experiment and thus we have to minimize the number of animal experiments as much as possible. Data collection and offline analysis helps to reduce the number of animal experiments by enabling thorough (re)use of collected data from prior experiments.

The EyeSAW system requires the use of different computing platforms. Specifically, the robot control is implemented on Linux to obtain better real-time performance, whereas some of the video acquisition has to be performed on Windows because it relies on hardware that is only supported on Microsoft Windows. This leads to the requirement for cross-platform support (**REQ 4**).

While most of the EyeSAW system is implemented in C++, we found that this did not provide a convenient environment for small student projects. Thus, it is necessary to have an interactive (scripting) interface to portions of the system to enable students to quickly implement and test new features (**REQ 5**). Also, access to run-time system information, the ability to inspect and change key parameters, and the support for dynamic component composition can facilitate the system development and debugging process.

One design decision that we made was to provide features and mechanisms at the framework level (i.e., in an application independent manner) as much as possible, rather than at the application level. Because the EyeSAW system is built on top of *cisst*, this allows the EyeSAW system to achieve the technical requirements with minimal implementation overhead, and this would be beneficial to other applications that use *cisst*.

3 Overview of *cisst*

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 [17], [18]. 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[1], to facilitate the development of various medical and surgical robot systems. Although the *cisst* package was originally developed for computer-assisted intervention (CAI) systems, we also use it for other robotics applications, such as space robotics [19].

TABLE 2
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

3.1 The *cisst* Component-based Framework

The *cisst* package [20] is a collection of open-source, cross-platform libraries [4]. 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*).

The *cisst* libraries form the basis of the Surgical Assistant Workstation (SAW) package [13]. 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

1. 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.

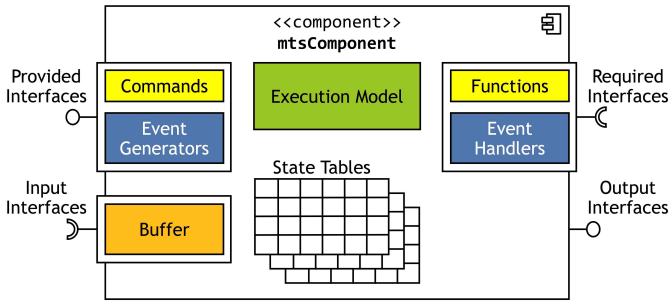

Fig. 4. Block diagram of the structure of the *cisst* component. A *cisst* component contains a list of interfaces for data exchange and time-indexed circular buffers (i.e., state tables [21]) for data archival and lock-free data retrieval.

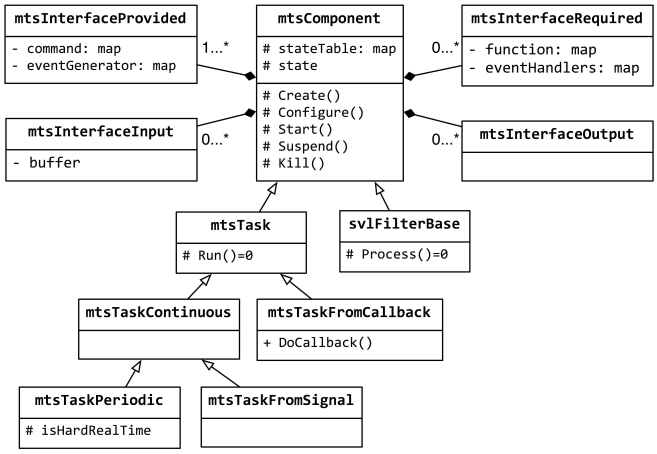

Fig. 5. UML class diagram of the *cisst* components

(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 [12]. 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*.

3.2 *cisst* Component Model

The *cisstMultiTask* library defines the *cisst* component model, which is implemented by the `mtsComponent` base class. Fig. 4 and 5 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 [22], 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 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 [21]. 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 [23].

The connection between components in the same process is established by the *Manager Component Client (MCC)* shown in Fig. 6, 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[2], 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.

The architecture of *cisstMultiTask* has been extended to support various system configurations, from multi-threaded scenarios to multi-process and multi-host systems [24]. This extension primarily relies on the component-based data exchange mechanism (i.e., the encapsulation of services) via a

2. 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.

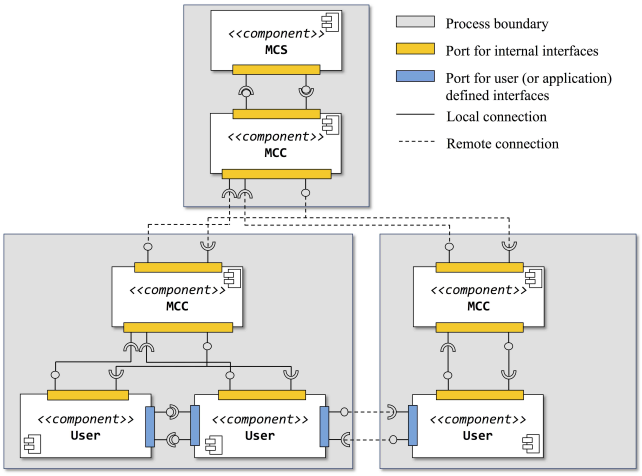

Fig. 6. Managing *cisst* components distributed over a network. The Manager Component Client (MCC) component, unique within a process, connects to every component in the same process. The Manager Component Server (MCS) component is unique in the entire system and connects to all MCC components in the 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.

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 [22], [25]). The proxy components mediate data exchange between the original components across the network. The network layer uses the Internet Communication Engine (ICE) [26] 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. 6. 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. In Sec. 5.2.1, we further discuss the semantics of the level of standardization or granularity regarding the connection process.

3.3 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 3 summarizes the *cisst* thread execution model based on these criteria.

TABLE 3
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. Additional details are provided in Section 5.7.

3.4 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 [21]. 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 [27]. 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 [28], [29].

4 Related Works

A variety of component-based software packages and middlewares have been developed and released to facilitate robotics research by providing libraries and tools for building robot applications. A very short list of such packages includes ROS [30], Orocos [31], OPRoS [32], CLARAty [33], *cisst* [4], and OpenRTM-aist [34]. One recent survey presented a more complete list of such packages [35].

Two widely used packages are ROS and Orocos. The Robot Operating System (ROS) [36] helps robot system developers by providing libraries and tools such as hardware abstraction, device drivers, libraries, visualizers, message-passing, and package management. The Open Robot Control Software (Orocos) [37] is a set of C++ libraries for machine and robot control and consists of the Kinematics and Dynamics Library (KDL), the Bayesian Filtering Library (BFL), and the Orocos Real-Time Toolkit (RTT) (now part of the Orocos Toolchain).

TABLE 4
Comparison of *cisst* with other robot software frameworks in terms of component modeling primitives (original table from [38]).

	Component	Port		Data Types	Connection
		Data Flow	Service		
OpenRTM	✓	✓	✓	✓	✓
Genom	✓	-	-	✓	-
ROS	✓	✓	-	✓	-
Orocos	✓	✓	✓	✓	✓
cisst	✓	✓	✓	✓	✓

Table 4 shows the *cisst* framework in comparison with other popular robot software frameworks in terms of essential structural elements, i.e., the component modeling primitives. This table is derived from the one presented by Shakhimardanov *et al.* [38], where we have added *cisst* as the last row of the table. *cisst* defines a component model (Sec. 3.2) where data is exchanged via four different types of interfaces. It also provides a set of services for user components via *internal* connections, such as component start/stop or component state retrieval (Sec. 3.2). A collection of domain-specific data types defined by the *cisstParameterTypes* library facilitates data exchange between *cisst* components (Secs. 3.4 and 5.2.1), and connections between components are explicitly modeled and represented as the connection (Sec. 3.2).

One recent trend among the community of robot software frameworks is the addition of support for other robotics frameworks via extensions, adapters, or bridges, in order to facilitate data exchange and component reuse between different frameworks. Orocos has extensions to ROS, Rock [39], and Yarp [40]. OpenRTM-aist and *cisst* also support ROS. The BRICS [41] framework supports Orocos at the meta-model level, rather than the code level, via the BRIDE (BRICS IDE).

5 Best Practices

This section summarizes some best practices that are derived from lessons we learned during the development of various medical robotics system. Many of them can be traced to the requirements of the EyeSAW system presented earlier, which is one of the more advanced systems developed with the *cisst* and SAW infrastructure.

5.1 Components in the same process, with efficient data exchange

Medical robot systems are information-intensive systems and require efficient data exchange and timely data processing to achieve both functional and nonfunctional requirements. Timely processing of acquired information is also important for the surgeon to make proper decisions based on the currently available information because delayed sensory feedback may lead to adverse effects on surgical performance or clinical outcomes. This leads to a requirement for minimal processing latency, which is particularly challenging for large data sets such as real-time vision data. For example, the EyeSAW system provides a visualization module for the surgeon, which processes a stream of high-definition stereo vision data (from a stereo microscope) at 900 Mbps (30+ FPS).

While some component-based systems require components to be in separate processes, or support only "second class" components (e.g., ROS *nodelets*) within the same process, we believe that the requirements for high-bandwidth control and low-latency video processing are best fulfilled by allowing multiple components to exist as separate threads within a single process. However, one challenge with multi-threaded applications is the management of access to shared resources. One can use locking mechanisms such as critical sections or semaphores, but these mechanisms introduce run-time overhead due to operating system calls, and can lead to deadlocks if not carefully used.

The *cisst* component-based framework, i.e., the *cisstMultiTask* library, provides lock-free, thread-safe, and efficient data exchange mechanism between components within a process, as described in Sec. 3.4. When building real-time robot control systems with vision processing capabilities, the *cisst* data exchange mechanism was extremely helpful for us to achieve both the real-time control and real-time imaging requirements with minimal processing latency. We note that some of the other software packages, such as Orocos [37], also support efficient data exchange mechanisms within a process. Another important feature of *cisst* is that the same components can be used locally (within the same process) or remotely (in a different process) without requiring any changes to the component source code. The *cisst* package achieves this by automatically creating proxy components to provide the necessary networking layer between components in different processes [27].

The EyeSAW system fully exploits these mechanisms. Within the *cisst* framework, both imaging/vision processing modules and robot control blocks are implemented as *cisst* components, and this design enabled us to achieve the challenging real-time and high-performance requirements (**REQ 1**). We also found it extremely useful to be able to switch between the two configurations (single process vs. distributed) with minimal source code changes (no changes to the component source code).

Lesson Learned: Multiple Components in the Same Process with Efficient Data Exchange

Enabling multiple components to exist in a single process, with efficient data exchange mechanisms, is essential for implementing robot systems with challenging real-time control and vision processing requirements. It is even better if the same components can be used within a single process, or distributed across multiple processes, without making any changes to the component source code.

5.2 Component Reuse via Standardized Messages and Repository

One of the major benefits of CBSE is the ability to reuse existing components in different applications or in different contexts. This enables rapid prototyping of new systems by reducing engineering efforts for designing and deploying components. Component reuse is also an important topic in modern medical and surgical robot systems where a variety of devices such as robots, haptic interfaces, tracking devices, and imaging devices are integrated into the system. From the viewpoint of integration and rapid prototyping, we consider two aspects of component reuse: *composition flexibility* and *component availability*.

5.2.1 Composition Flexibility

Composition flexibility refers to how easily a component can be deployed in application-specific use cases. Component composition requires the existence of shared contracts or specifications between two components, such as payload types, timing constraints (e.g., asynchronous vs. synchronous), and a list of services. Practically, strict specifications can reduce component usability but may be required to deliver optimal performance for specific use case scenarios, whereas loose specifications may improve usability but can lead to unnecessarily complex implementation to deal with a range of possible use cases.

The *cisst* component model supports composition flexibility with (1) standardized domain-specific data types, and (2) service specialization by *optional* structural elements. As described in Sec. 3.4, the *cisst* data exchange model uses strongly-typed payloads and the *cisstParameterTypes* library provides a collection of domain-specific payload types that

are frequently used in the robotics domain. This is analogous to the ROS `common_msgs` package.

The *optional* feature enables the *partial use of services*. Normally, all required interfaces of a component must be connected to valid provided interfaces (in other components) before the component can enter the READY state, at which point execution can begin. However, a required interface can be set as *optional* to enable the component to enter the READY state even if the optional required interface is not connected.

The same concept also applies at a more fine-grained level, i.e., at the command level. First, a required interface can use any *subset* of the services of the provided interface. In other words, the number of functions in the required interface may be less than the number of commands in the provided interface. Second, a function can be declared as *optional*, which means that the connection process does not fail if there is no corresponding command in the provided interface. In this case, the component with the required interface should check whether the function is valid before attempting to invoke it. As an example, consider a robot control component that uses both position and velocity feedback. This component would have a required interface, with "GetPosition" and "GetVelocity" functions, that connects to a provided interface in a hardware interface component. If the "GetVelocity" function is declared to be optional, the control component can estimate the velocity from the position feedback if the "GetVelocity" function is not valid (i.e., if the hardware does not provide velocity feedback). We note that our support for partial use of services, via the declaration of *optional* required interfaces and functions, is a consequence of the *cisst* component model, which groups related services (commands and events) into interfaces and connects components at the interface level. Other frameworks, such as ROS, that connect at the service level (e.g., by invoking a service or publishing/subscribing to a topic) inherently enable partial use of services.

Medical robot systems require the integration of diverse devices by nature, and we found that the standardized payload types and the partial use of services are useful for improving component composition flexibility, thereby facilitating the system development process.

5.2.2 Component Availability

Component reuse is only possible if existing components are readily available, such as via a component repository. Reusable components should be carefully designed, correctly implemented, and thoroughly tested, considering both intended and possible use cases as well as requirements. The idea of a component repository is to reuse component designers' prior experience, knowledge, and engineering efforts that are condensed into the design and implementation of components. With a quality component repository, system designers can deploy existing components into new systems with minimal engineering effort.

Reusable components can be deployed as class libraries or binary files[3], which is called white-box reuse and black-box reuse, respectively [43]. White-box reuse allows component users to modify or customize the component design or implementation, whereas black-box reuse restricts such customization for the purpose of delivering its intended services.

SAW is one example of a domain-specific component repository with the concept of white-box reuse. Currently, there are 26 SAW components. This includes interfaces to hardware devices such as robots (e.g., Barrett WAM, da Vinci surgical robot via its research interface), haptic devices (e.g., Phantom Omni, Novint Falcon), tracking systems (e.g., NDI Polaris and Aurora, Claron Micron Tracker), and navigation systems (e.g., Medtronic Stealthstation via Stealthlink). The repository also includes components that interface to software packages that provide functionality such as visualization, physics-based simulation, speech recognition, text-to-speech generation, and robot control. Finally, an important class of components provides interfaces to other protocols; key examples are a ROS bridge (translating between *cisst* commands/events and ROS messages/topics) and an OpenIGTLink bridge [44] that enables integration with Image Guided Therapy software and systems, such as 3D Slicer. In addition, there currently are two SAW applications that can be used out-of-the-box: *sawDataPlayer* (see Sec. 5.3) and *sawIntuitiveResearchKit* [12]. The complete list of SAW components is available at [45].

Each SAW component has the same file system structure, with separate folders for its implementation (`include` and `code` folders) and examples (`examples` folder). Examples are simple demo programs showing how to use SAW components and there can be different versions of the example based on dependencies. For example, *saw3Dconnexion* (an interface component for the 3D Connexion Space Navigator, a.k.a., 3D mouse) has three variations: one using FLTK, another one using Qt, and the other with minimal dependency on *cisst*. Other variations include terminal-based vs. GUI-based applications.

Based on our experience, the structure of the SAW component repository facilitated the component development process by providing "templates" or "guidelines" for component designers. Furthermore, SAW component users find that the consistent organization of SAW components and simple examples with variations depending on dependencies or application types lowered the learning curve when using new SAW components.

The EyeSAW system uses various hardware and software modules, and this leads to a large number of components and connections, as shown in Fig. 3. Furthermore, this configuration dynamically changes to support different surgical use cases or scenarios, i.e., behaviors, at run-time.

3. There is a work that proposed a classification of software component models based on how its component repository, if any, is used for component composition [42].

Use of consistent and standardized parameter types within the system facilitated component composition for multiple behaviors, and the SAW component repository significantly reduced implementation overhead (**REQ 2**).

Lesson Learned: Component Reuse via Standardized Messages and Repository

Component reusability can be improved by (1) supporting composition flexibility (e.g., use of standardized payload types, options for selective use of services) and (2) making quality components available via a (domain-specific) component repository.

5.3 Data Collection and Replay

At run-time, a variety of different data in terms of size and rate are generated, processed, and exchanged within robot systems. Collecting such diverse data *efficiently* at run-time is not trivial. For example, a typical servo-level robot control module processes *small* data of a few tens of bytes at *high* frequencies up to a couple kHz, whereas a real-time imaging module with vision processing streams may handle *large* data, such as a couple hundred MB per second, at relatively *low* frequencies (e.g., 20-30 FPS). It becomes more challenging if the number of interacting components increases, which is often the case for modern robot systems. In the case of surgical robot systems, a wide variety of data types are typically generated during surgery, including occasional textual notes by surgeons or medical personnel, continuous audio and video streams, and diverse sensor data.

One key design requirement of a data collection mechanism is *minimal run-time overhead* on the target component from which data are collected, so as to avoid any performance degradation of the target component due to the data collection activity. Data collection involves run-time overhead for data retrieval (via the data exchange mechanism) and file system access (mostly for write), which consumes part of the system resources such as CPU time, network bandwidth, and disk access. Thus, a data collection mechanism should be able to collect and archive data *locally*, whenever possible.

The *cisst* framework provides a data collection tool that includes a state collector (Fig. 7), an event collector, and a video recorder with timestamps. The data collector is designed as a component, rather than an object, so that system designers can deploy the collectors using the component-based facilities, i.e., the Command Pattern. It is possible to control its activities (e.g., start, stop) or get/set a log file name through its provided interface. When fetching data, the collector directly accesses the state table of the target component to obtain maximum efficiency (without violating the thread safety) but with minimal run-time overhead. Each collector instance uses a separate log file and thus there is no contention between collectors to access the same log file. The data collector relies on the lock-free and

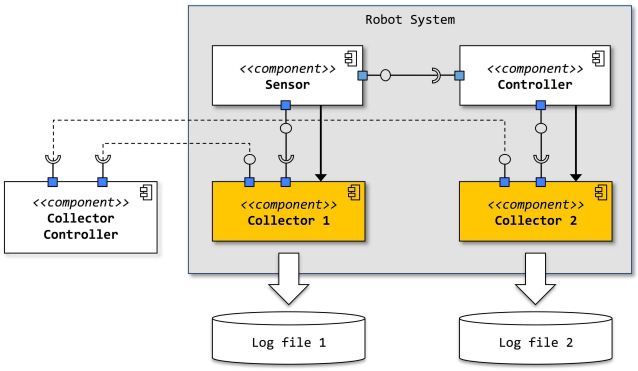

Fig. 7. Overview of State Collector: The collector is designed as a component, rather than an object. This helps the system designer flexibly deploy the collectors to the system. Each collector has a provided interface that allows another component to control its activity (e.g., start, stop) at run-time via the Command Pattern.

thread-safe data exchange mechanism between components, as described in Sec. 3.4, and this allows high-bandwidth data collection with minimal (almost zero) execution overhead on the target component. Multiple instances of the data collector can be created and distributed to each process, if it is necessary to collect large datasets from a networked system (e.g., joint positions of high degree-of-freedom systems with video streams from multiple cameras). We have found that this is often preferable to having a single data collector that would rely on network data transfers.

Another tool that the *cisst* framework provides is a data replay tool, called *cisstDataPlayer* (Fig. 8), which can be used to analyze multi-media data collected during experiments. The *cisstDataPlayer* can handle multiple data files, with different types of data (e.g., state variables, video and images, or events) via multiple instances of data player components (plug-ins) that are deployed for each target data. All collected data is time-stamped, and each data player component is synchronized. On a networked system, it is necessary to synchronize the different computer clocks using a protocol such as Network Time Protocol (NTP).

Our experience with real-time data collection and replay tools identified a few desirable features of data collection mechanisms, in general, as follows:

- **Preservation of original data**: When archiving collected data into log files, it is desirable to preserve original data as much as possible, and the use of a binary representation with data serialization is one effective approach.
- **Support for data export in standard formats**: We found that support for data conversion or export in standard formats such as human-readable representation (e.g., ASCII text or CSV files) are particularly useful for data analysis and post-processing using external tools

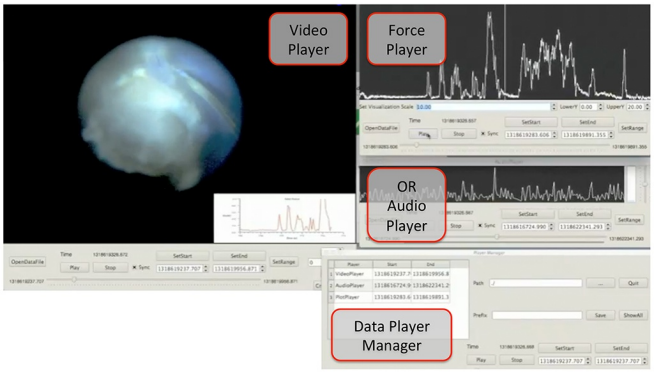

Fig. 8. Screenshot of Data Replay Tool (*cisstDataPlayer*): Data Player Manager (bottom right) controls data player instances for recorded video (left), recorded force data (top right) and recorded audio (middle right).

such as Matlab.

- **Flexible deployment of data collector**: Being able to easily and flexibly deploy multiple instances of data collectors to any component in the system significantly facilitates the data collection process. We found that this feature is extremely useful for data collection in distributed, complex, or large systems.

As part of the EyeSAW system development process, we performed numerous experiments, including phantom studies, in-vitro, and in-vivo experiments. In many situations (e.g., during in-vivo experiments), it is not feasible to analyze the data as it is collected, and thus the *cisst* data collection tool and *cisstDataPlayer* were two essential tools that enabled offline "data reuse" such as post-operative data analysis and experiment review (**REQ 3**). For example, we were able to visually identify (using the Video Player in Fig. 8), with audio annotations (of the OR Audio Player), what was happening when excessive force feedback was applied (via the Force Player) during retinal membrane peeling experiments, because the three data players are all synchronized, i.e., can be navigated using the timeline control UI of any one of the data players. We have found that such "data reuse" is extremely useful for reviewing experiments, and reproducing and debugging problems, thereby improving the design of the system and applications.

Lesson Learned: Data Collection and Replay

Support for lightweight, flexible, and efficient data collection mechanisms is essential for enabling post-analysis and "data reuse", especially in large and complex component-based systems.

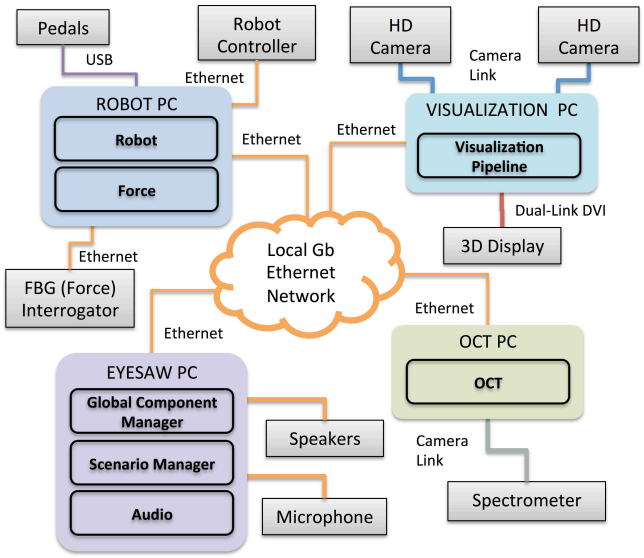

Fig. 9. Process view of EyeSAW. The Robot and OCT PCs use Linux for real-time performance, and the Visualization and EyeSAW PCs run on Windows.

5.4 Cross-Platform Support

A robot control system with a graphical user interface (GUI) is often a combination of a real-time OS (e.g., QNX, VxWorks, Linux RTAI or Xenomai) for robot control and a conventional OS (e.g., Windows, Mac, Linux) for GUI or visualization. This provides hard-real time performance for the robot controller, and can also increase system safety by isolating or protecting the robot controller from the GUI module.

The use of multiple OSs is sometimes unavoidable where device drivers or software packages are (un)available only on specific OSs with no other option. Such strong OS dependencies restrict the way the system is designed, and thus can significantly limit the way components are deployed in the system. This restriction is particularly challenging for medical robot systems because not only robotic devices (e.g., sensors, actuators, controllers) but also various *medical* devices, such as optical trackers, ultrasound imaging devices, and a robot system itself, have to be integrated into a single system. To be used for clinical tests or released as commercial products, these medical devices must be approved by regulatory agencies such as the U.S. Food and Drug Administration (FDA), and medical device manufacturers have to invest a significant amount of resources for this process. Researchers in academia cannot afford such a demanding effort, and thus need to rely on commercial medical devices, which may introduce OS dependencies.

Another important aspect of medical robot systems is that medical robotics research requires interdisciplinary efforts; it involves individuals from many disciplines with diverse backgrounds, such as clinical researchers, biomedical engi-

neers, mechanical engineers, computer scientists, or roboticists. Some are comfortable with any platform, whereas others may not be able to use particular operating systems. This also applies to undergraduate research, where students may only be familiar with a particular operating system. This motivates the need for cross-platform support.

Ideally, a component-based framework should support multiple platforms. If the framework does not support a desired platform, developers are forced to either use a supported platform or to manually implement an inter-process communication module that transfers data from the unsupported platform. This task of "reinventing-the-wheel", however, involves not only significant development overhead, but also requires comprehensive understanding of how a specific OS works as a whole. Without expertise, run-time issues such as performance, threading, or synchronization may be introduced to the system.

Cross-platform robot software frameworks use a cross-platform build tool, such as CMake (cmake.org), and have an internal layer that replaces OS-specific function calls with a set of platform independent APIs. It is also desirable to use cross-platform packages for external dependencies as much as possible (e.g., FLTK or Qt instead of MFC).

The *cisst* framework uses CMake as its build tool and most third-party dependencies are cross-platform packages. It also includes an OS abstraction library, *cisstOSAbstraction*, which supports Windows, Linux, Mac, real-time Linux (RTAI and Xenomai), and QNX (see Table 2). The *cisstOSAbstraction* library provides platform-independent APIs, which is particularly helpful when switching to a new platform.

The EyeSAW system uses multiple computers with different operating systems, as shown in 9. The Robot PC uses Ubuntu x64 for hard real-time robot control, the Visualization PC and the EyeSAW PC run on Windows 7 x64, and the OCT PC used to be on Ubuntu x64 but is now running under Windows 7 x64. The use of Windows is primarily due to Windows-specific hardware. For example, the high-performance OCT imaging hardware uses a Matrox frame grabber and the driver is available only on Windows. The data from the grabber is immediately processed, displayed, and archived locally because of bandwidth limitations. At the same time, the robot application running on a Linux machine accesses the reduced OCT image data to perform the OCT-based feedback control.

This OS dependency required cross-platform support (**REQ 4**). Furthermore, this enables easy migration to a new platform; for example, if we later decide that we need hard real-time performance, it is only necessary to recompile the software for one of the real-time operating systems supported by the *cisstOSAbstraction* library (e.g., Linux/RTAI, Linux/Xenomai, or QNX). Migration to a new (currently unsupported) operating system would only require changes to *cisstOSAbstraction* and certain hardware-dependent SAW components, rather than having to port the rest of the *cisst* library or EyeSAW application.

Lesson Learned: Cross-Platform Support

Robotics software packages that can provide an operating system agnostic development environment are desirable because they increase the system design flexibility by relieving software dependencies on specific operating systems. If possible, it helps to choose cross-platform packages for any dependencies.

5.5 Interactive Interfaces and Utilities

The development process of robot systems usually involves repetitive trial-and-error testing of algorithms, parameter optimization, debugging, and so on. Without proper tool support, system developers must go through a series of steps, such as compiling updated source code, warming up or homing the robot, progressing through the workflow to the desired state for testing, and, finally, performing tests and collecting data. Obviously, this is time-consuming and inefficient.

Within the robotics community, there has been a recognition of such practical issues that repeatedly occur during the system development process, and this accelerated the adoption of CBSE [17], [18] by the community. Recent component-based robot software frameworks (e.g., ROS, OROCOS, BRICS, OPRoS) support tool suites that range from terminal-type interactive shells to GUIs and visualization modules. Ideally, it is desirable for these tools to be able to "probe" or query various run-time information of the system, such as a list of components in the system, topology of component connections, and current state of a specific component. The tools would be even more useful if they allow system developers to interact with the system for dynamic component composition or connection topology manipulation (e.g., hot swapping of components at run-time, establishing new connections, disconnecting failed connections).

The *cisst* component-based framework, i.e., the *cisstMultiTask* library, provides three tools that allow system developers to inspect and interact with the system: the Interactive Research Environment (IRE), the *cisst* Component Viewer, and the *cisst* Component Manager. The IRE is shown in the left of Fig. 10. It includes an embedded Python shell with GUI support and can directly access (read/write) application data. It can also dynamically connect to any provided interface in the system, even if it is in a different process.The *cisst* Component Viewer depicts all the components and connections in the system, as in the top right of Fig. 10. The color of components encodes its run-time state. It is also possible to dynamically change the state of the components and connect or disconnect them by interacting with this graph. For example, a pop-up menu is displayed when the user right-clicks on a component; using this menu the user can start or stop the component, or view a list of its interfaces. The *cisst* Component Manager is a console-based utility that provides a set of similar functionali-

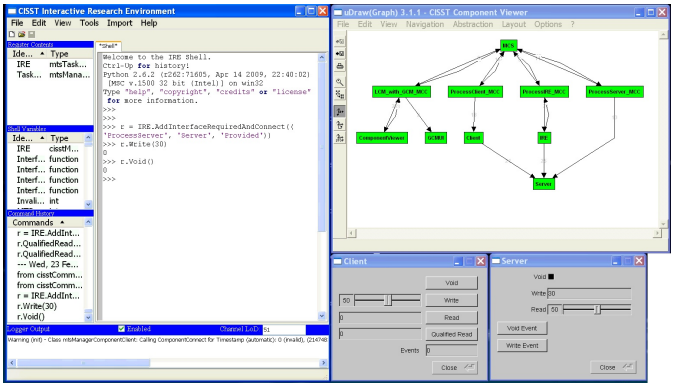

Fig. 10. Interactive tools in *cisst*: Interactive Research Environment (IRE) on the left and the *cisst* Component Viewer on the right. The IRE is a Python shell with GUI support. The *cisst* Component Viewer uses the uDrawGraph program from the University of Bremen for visualization, and shows run-time information about the system such as components and connection topology. Both allow system developers to interact with the system.

ties and also enables users to dynamically load new component libraries and create instances of these components.

In the EyeSAW system, we developed a simple Python interface to a subset of system functionalities. This provided a convenient environment for small student projects, which allowed students to quickly implement and test new features without modifying and recompiling the existing system (**REQ 5**). For example, one student used the Python interface to the EyeSAW system when developing an automatic calibration method for a custom force sensor.

Lesson Learned: Interactive Interfaces and Tools

Support for run-time debugging and system inspection tools significantly reduces engineering overhead, thereby facilitating the robot system development process. In CBSE, it is desirable to have support for a scripting language, as well as tools for dynamic component composition and connection topology manipulation (e.g., creating new connections, disconnecting existing connections).

5.6 Use Type Traits and Support Any Data Type for Messages

Because components may (or in some cases, must) exist in different processes, it is necessary for the sending component to serialize the data and for the receiving component to deserialize it. This raises two issues: (1) the code for serializing and deserializing each message data type must be written, and (2) the system must be able to invoke the serialization and deserialization functions. The first issue is usually solved by defining the message types using a special language, such as the CORBA Interface Description Language (IDL), the Specification Language for ICE (SLICE), or ROS *msg* files (this also enables cross-language support, which is not a requirement for *cisst*). The second issue can be solved by deriving all message types from a common base class, such as `Ice::Object` and the now-deprecated `ros::Message`, or by using type traits. Underlying these issues is the question of what data types are allowed for messages. Using a specification language or deriving from a common base class are not amenable to allowing the use of any type (e.g., a fixed-size vector from your favorite vector package).

The initial implementation of *cisst* relied on a common base class (`mtsGenericObject`, which in turn was derived from `cmnGenericObject`), but did not use a specification language and therefore required the programmer to manually create the serialization and deserialization methods (as part of the *class services*). It became evident that this combination was a poor design choice. While manually creating the class services provided the greatest flexibility (at some programming cost), the use of a common base class prevented the potential benefit of allowing messages of any data type. For example, the *cisstVector* library contains a typedef, `vctDouble3` (or just `vct3`), for a fixed-size vector of three double-precision floating point numbers. This class is not derived from `cmnGenericObject`; thus, it could not be used in a message. One solution is to create a new class, `mtsVct3`, that uses multiple inheritance to derive from both `vct3` and `mtsGenericObject`. This, however, leads to code that alternates between `vct3` and `mtsVct3` objects, which complicates programming and reduces readability. We therefore adopted a *type traits* approach, where the class services are available via the definition of a templated class; for example, `cmnData<vct3>` provides the class services for `vct3`. Internally, *cisst* still requires a common base class for a few features, such as the State Table described in Section 3.2 and dynamic object creation during deserialization, but these are handled by a "wrapper" class that is derived from `mtsGenericObject` and is templated by the data type (e.g., `vct3`). In addition, the type traits approach allows users to use both primitive data types and custom (user) data types transparently in their code, thereby improving readability.

We also found that while manually creating the class services provides the most flexibility, it is not only a significant burden on programmers, but also an error-prone process due to mistakes or typos with copy-and-paste code. Furthermore, much of the time this flexibility is not needed because the programmer is creating a new type rather than using a type from an existing library. We therefore introduced a specification language and a corresponding data generator that parses the data type specification and automatically creates the code for all class services. We introduced a new specification language (*cdg* file) and data generator (*cisstDataGenerator*) to be able to support some unique features in *cisst*, such as the ability to query the number of scalars and retrieve the value of a

specific scalar for the purpose of data collection. Of course, it is still possible to manually create the class services to be able to support existing types, such as the mathematical types in *cisstVector* or types from an external package such as VTK. We note that this "best practice" has been identified in other robot software packages; for example, ROS initially derived all messages from a common base class (`ros::Message`), but now recommends a template-based serialization and traits system and provides documentation on how to manually create the serialization methods for existing types.

Lesson Learned: Use Type Traits and Support Any Data Type for Messages

It is better to use a *type traits* approach for class services, such as serialization and deserialization, rather than deriving from a common base class. Also, while it is convenient to have automatic generation of the class services, based on a high-level specification of the data type, it is also useful to be able to manually create the class services to support the use of existing types (e.g., from another library) in messages.

5.7 Support Changes to Component Execution Model during Deployment

By default, the *cisst* library combines the functional characteristics of a component with its execution model. As illustrated in Fig. 5, the *cisstMultiTask* library provides several component classes that contain a "hard-wired" execution model, such as *Continuous* or *Periodic*. A component designer creates a new component by inheriting from one of these classes, depending on the desired computation scheme. That is, the default thread execution model of the new component is determined by the base class. One advantage of this design is that the relationship between different computing schemes becomes explicit and obvious. It is also a user convenience to have a single component that provides both the logic and execution model.

The caveat of this initial design, however, is that the switch of thread execution model would require manual source code-level changes. For example, to switch the computation scheme of a user component from *Periodic* to *Continuous*, it would be necessary to change the base class in the source code and recompile the user component and application code. This is not convenient and cannot be done dynamically.

One option for improvement is to separate the thread execution model from the class implementation, and to use an argument to specify which computation scheme should be used when creating the component. This would be more convenient and enable the dynamic creation of a component with different computation schemes. An alternative solution, employed in some other robot frameworks such as Orocos [37] and OPRoS [32], is to completely separate the execution model from the component; for example, by creating an execution engine that provides the thread or threads for specified components.

For *cisst*, we chose a backward-compatible solution that is a compromise between the convenience of specifying the component execution model during design and the flexibility of selecting it during deployment. Specifically, components are still derived from one of the base classes shown in Fig. 5; this defines the default execution model for that component. But, all components now contain *ExecOut* and *ExecIn* interfaces. Connecting the *ExecIn* interface of one component to the *ExecOut* interface of another component causes the first component to adopt the execution model of the second component. Essentially, the component with the *ExecOut* interface provides the functionality of the execution engine available in other frameworks. In other words, all components in *cisst* can be components and execution engines. Because selection of an execution model may require the creation and configuration of resources, the ability to override the default execution model is only allowed before the component is started. Once the component begins execution, the system enforces this restriction by removing the *ExecIn* interface if it has not been connected.

Lesson Learned: Support Changes To Component Execution Model During Deployment

While it may be convenient to specify a default execution model during implementation, it is desirable to be able to easily change the computation scheme of a component during deployment. This improves the maintainability of the system by providing flexible options to choose or change the thread execution model, especially when the design of the computation scheme changes.

6 Discussion

We presented best practices (lessons learned) from our experience with the development of the *cisst* component-based framework, the SAW package, and various medical and surgical robot systems. These include: (1) support for components in the same process, with efficient data exchange mechanisms, (2) component reuse enabled by composition flexibility and a component repository, (3) tools for data collection and replay, (4) cross-platform support (both conventional and real-time operating systems), (5) interactive interfaces and utilities, (6) use of any data type for messages, and (7) deployment changes to the component execution model.

We do not argue that support for these features necessarily leads to the ideal robot software framework, nor that ideal robot software frameworks must have these features. They are derived primarily from our experience in the medical robotics domain, and can contain domain-specific content. However, modern medical and surgical robot systems have evolved enough to seriously consider challenging systems

issues such as complexity, scale, and functional and non-functional requirements, which are also commonly found in general robot systems. Over the years, we have repeatedly faced such issues, and the lessons described in Section 5 summarize our approaches and solutions to those issues, which we found to be effective and useful based on our experience.

Not surprisingly, many of these "best practices" can be found in other robot software frameworks, though the implementation details and level of support may vary widely. Table 5 summarizes how Orocos, ROS, and *cisst* support each best practice. Orocos and *cisst* have "native" (i.e., component model-level) support for efficient data exchange between components in the same process, whereas ROS provides a similar feature with an additional package (*nodelet*). All three frameworks maintain a component repository (Orocos component package, ROS package, SAW), and provide tools or facilities for data collection and replay (Orocos `OCL::ReportingComponent`, ROS *Bags*, *cisst* state collector and *cisstDataPlayer*). Orocos and *cisst* have good cross-platform support (at least Microsoft Windows, Mac OS X, Linux and real-time Linux variants), whereas ROS is primarily supported on Ubuntu Linux. The *cisst* libraries are also supported on QNX. The three frameworks support tool suites to interact with the system, and each framework has its own semantics or facilities for data types or payload types (Orocos *typekit*, ROS *msg* and *tf package*, *cisst cdg* files and *cisstDataGenerator*). Orocos and *cisst* allow the component execution model to be specified during deployment. Although ROS does not require users to specify a threading model, it allows users to execute their code in different ways, including periodic (using timer), single-threaded, and multi-threaded callbacks.

TABLE 5
Support for best practices: Orocos, ROS, and *cisst*.

	Best Practices	Orocos	ROS	*cisst*
1	Components in the same process	✓	*nodelet*	✓
2	Component reuse	✓	✓	✓
3	Data collection	✓	✓	✓
4	Cross-platform	✓		✓
5	Interactive tools	✓	✓	✓
6	Arbitrary message types	✓	✓	✓
7	Component execution model at deployment	✓	n/a	✓

Currently, our work focuses on medical robot systems that interact with humans, or involve the human as part of the system control loop. Due to the human factor, part of our research centers around non-functional properties of component-based software systems, especially safety. Although everyone accepts that safety is the most crucial property of medical robot systems, systematic approaches to system safety have not yet received much attention. Within the component-based environment, we are looking into systematic methods that help us improve the safety of component-based software systems [46], [47].

No single robot software framework is "perfect", in the sense that each framework has its own design and a list of features and services for which it is intended to be used. However, we believe we have been facing similar problems as the larger community of robot software framework developers. It is our belief that sharing experiences with others and other robot software frameworks will help the community to deal with these problems, and eventually come up with effective solutions, thereby taking us one step closer to the "ideal" robot software framework.

7 Conclusions

We described practical problems and challenges that we have faced throughout the development of component-based medical and surgical robot systems. As our approaches and solutions, we also discussed the design of the *cisst* package and the SAW component repository, and summarized best practices as follows:

- Supporting multiple components, with efficient data exchange, within a single process
- Component reuse via composition flexibility and an organized repository
- Tools for data collection and replay
- Cross-platform support
- Interactive interfaces and utilities
- Use of arbitrary data type for messages
- Flexible specification of component execution

Although these lessons are derived from our experience in medical robotics, our belief is that they can apply to robots in other application domains, and can add to the discussion of best practices in robotics.

One direction for future work in the field of robot software frameworks is a systematic approach to dependability, especially safety, in a component-based environment. Safety is a critical property of robot systems that physically interact with humans, but it has not yet received much attention in the field. Systematic methods to specify safety requirements, implement safety features, or verify and validate safety cases would be a significant contribution to the robotics domain, especially to medical and surgical robotics.

Supplemental Material

The *cisst* and SAW packages are open source and the entire source code is available at the following URLs:

- The *cisst* package: http://cisst.org/cisst
- The SAW package: http://cisst.org/saw

We recently moved to GitHub (github.com), an open source community, and the *cisst* and SAW packages are also available there:

- The *cisst* package: https://github.com/jhu-cisst/cisst

- The SAW package: https://github.com/jhu-saw

The complete list of SAW components is available at the SAW website.

ACKNOWLEDGMENTS

The *cisst* and SAW software packages have benefited from the contributions of many individuals. Key contributors include Balazs Vagvolgyi, Simon Leonard, Rajesh Kumar, Ankur Kapoor, Ofri Sadowsky, Zihan Chen, and Ali Uneri.

REFERENCES

[1] 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. 1

[2] A. Bzostek, R. Kumar, N. Hata, O. Schorr, R. Kikinis, and R. H. Taylor, "Distributed modular computer-integrated surgical robotic systems: Implementation using modular software and networked systems," in *Intl. Conf. on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Pittsburgh, PA: Springer-Verlag, 2000, pp. 969–978. 1

[3] O. Schorr, N. Hata, A. Bzostek, R. Kumar, C. Burghart, R. Taylor, and R. Kikinis, "Distributed modular computer-integrated surgical robotic systems: Architecture for intelligent object distribution," in *Intl. Conf. on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Pittsburgh, PA: Med Image Comput Comput Assist Interv. (MICCAI), Oct 2000. 1

[4] 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. 1, 3.1, 4

[5] B. Vagvolgyi, S. DiMaio, A. Deguet, P. Kazanzides, R. Kumar, C. Hasser, and R. Taylor, "The Surgical Assistant Workstation: a software framework for telesurgical robotics research," in *MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Midas Journal, Sep 2008. [Online]. Available: http://hdl.handle.net/10380/1466 1

[6] T. Xia, C. Baird, G. Jallo, K. Hayes, N. Nakajima, N. Hata, and P. Kazanzides, "An integrated system for planning, navigation and robotic assistance for skull base surgery," *Intl. J. of Medical Robotics and Computer Assisted Surgery (MRCAS)*, vol. 4, no. 4, pp. 321–330, 2008. 1

[7] 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. 1

[8] A. Uneri, S. Schafer, D. Mirota, S. Nithiananthan, Y. Otake, R. Taylor, and J. Siewerdsen, "TREK: an integrated system architecture for intra-operative cone-beam CT-guided surgery," *Intl. J. of Computer Assisted Radiology and Surgery*, vol. 7, no. 1, pp. 159–173, 2012. 1

[9] P. Thienphrapa, B. Ramachandran, R. Taylor, and A. Popovic, "A system for 3D ultrasound-guided robotic retrieval of foreign bodies from a beating heart," in *IEEE RAS EMBS Intl. Conf. on Biomedical Robotics and Biomechatronics (BioRob)*, 2012, pp. 743–748. 1

[10] W. P. Liu, S. Reaugamornrat, A. Deguet, J. Sorger, J. Siewerdsen, J. Richmon, and R. Taylor, "Toward intraoperative image-guided transoral robotic surgery," *J. of Robotic Surgery*, vol. 7, no. 3, pp. 217–225, 2013. 1

[11] M. A. Balicki, "Augmentation of Human Skill in Microsurgery," Ph.D. dissertation, The Johns Hopkins University, 2013. 1

[12] 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. 1, 3.1, 5.2.2

[13] 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 Arch. for Computer Assisted Interventions*, Midas Journal, Jun 2010. [Online]. Available: http://hdl.handle.net/10380/3179 2.1, 3.1

[14] A. Uneri, M. A. Balicki, J. Handa, P. Gehlbach, R. H. Taylor, and I. Iordachita, "New steady-hand eye robot with micro-force sensing for vitreoretinal surgery," in *BIOROB*, Tokyo, Japan, Sep 2010. 2.1

[15] M. Balicki, J. H. Han, I. Iordachita, P. Gehlbach, J. Handa, R. Taylor, and J. Kang, "Single fiber optical coherence tomography microsurgical instruments for computer and robot-assisted retinal surgery," in *Medical Image Computing and Computer Assisted Intervention (MICCAI)*, London, Sep 2009, pp. 108–115. 2.1

[16] M. Balicki, R. Richa, B. Vagvolgyi, P. Kazanzides, P. Gehlbach, J. Handa, J. Kang, and R. Taylor, "Interactive OCT Annotation and Visualization for Vitreoretinal Surgery," in *MICCAI Workshop on Augmented Environments for Computer-Assisted Interventions*, C. Linte, Ed. Heidelberg: Springer, 2012, pp. 142–152. 2.1

[17] D. Brugali and P. Scandurra, "Component-Based Robotic Engineering (Part I)," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, Dec. 2009. 3, 5.5

[18] 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. 3, 5.5

[19] 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. 3

[20] The *cisst* package. (Accessed: 2013-11-07). [Online]. Available: http://www.cisst.org/cisst 3.1

[21] 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. 4, 3.2, 3.4

[22] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. 3.2, 3.2

[23] 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. 3.2

[24] M. Y. Jung, A. Deguet, and P. Kazanzides, "A component-based architecture for flexible integration of robotic systems," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2010. 3.2

[25] M. Shapiro, "Structure and encapsulation in distributed systems : the proxy principle," in *Intl. Conf. on Distributed Computing Systems (ICDCS)*, 1986, pp. 198–204. 3.2

[26] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, Jan-Feb 2004. 3.2

[27] 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. 3.4, 5.1

[28] 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. 3.4

[29] 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. 3.4

[30] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009. 4

[31] 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. 4

[32] C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim, and C.-H. Lee, "OPRoS: A new component-based robot software platform," *ETRI Journal*, vol. 32, no. 5, pp. 646–656, 2010. 4, 5.7

[33] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty architecture for robotic autonomy," in *IEEE Aerospace Conference*, vol. 1, Mar 2001, pp. 121–132. 4

[34] 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. 4

[35] A. Elkady and T. Sobh, "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography," *Journal of Robotics*, 2012. 4

[36] Robot Operating System (ROS). [Online]. Available: http://www.ros.org 4

[37] Orocos Real-Time Toolkit. [Online]. Available: http://www.orocos.org 4, 5.1, 5.7

[38] 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. 4

[39] Rock: The Robot Contruction Kit. [Online]. Available: http://rock-robotics.org 4

[40] Yet Another Robot Platform. [Online]. Available: http://eris.liralab.it/yarp 4

[41] 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. 4

[42] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. on Software Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007. 3

[43] D. Brugali and A. Shakhimardanov, "Component-Based Robotic Engineering (Part II)," *IEEE Robotics & Automation Magazine*, vol. 17, no. 1, pp. 100–112, Mar. 2010. 5.2.2

[44] J. Tokuda, G. S. Fischer, X. Papademetris, Z. Yaniv, L. Ibanez, P. Cheng, H. Liu, J. Blevins, J. Arata, A. J. Golby, T. Kapur, S. Pieper, E. C. Burdette, G. Fichtinger, C. M. Tempany, and N. Hata, "OpenIGTLink: an open network protocol for image-guided therapy environment," *The International Journal of Medical Robotics and Computer Assisted Surgery*, vol. 5, no. 4, pp. 423–434, 2009. 5.2.2

[45] The Surgical Assistant Workstation (SAW) components. (Accessed: 2013-11-07). [Online]. Available: http://www.cisst.org/cisst 5.2.2

[46] M. Y. Jung and P. Kazanzides, "Run-time Safety Framework for Component-based Medical Robots," in *Medical Cyber Physical Systems Workshop, CPSWeek 2013*, 2013. 6

[47] 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. 6

Min Yang Jung received his B.S. in Electrical Engineering and M.S. in Biomedical Engineering from Seoul National University, South Korea. He is currently a Ph.D. Candidate in Computer Science at The Johns Hopkins University, and his thesis focuses on the software system safety for component-based medical and surgical robot systems. His primary interest is the design and development of safety-critical real-time robot control systems.

Marcin Balicki received his B.Sc in Interdisciplinary Engineering and MS in Mechanical Engineering from The Cooper Union in New York City. For his Ph.D in Computer Science from The Johns Hopkins University, he investigated augmentation of human skill to address the extreme challenges of micro-surgery using robotic manipulators, micro-sensing devices, and visualization techniques.

Anton Deguet received his B. Sc. in Mathematics from Université François Rabelais in Tours, France and his MS in Computer Science from Université Joseph Fourier in Grenoble, France. He is now a Senior Software Engineer at The Johns Hopkins University, working with researchers and students on all things software related.

Russell H. Taylor received his Ph.D. in Computer Science from Stanford University in 1976. He joined IBM Research in 1976, where he developed the AML robot language and managed the Automation Technology Department and (later) the Computer-Assisted Surgery Group before moving in 1995 to Johns Hopkins University, where he is the John C. Malone Professor of Computer Science with joint appointments in Mechanical Engineering, Radiology, and Surgery and is also Director of the Engineering Research Center for Computer-Integrated Surgical Systems and Technology (CISST ERC). He is the author of over 275 peer-reviewed publications, a Fellow of the IEEE, of the AIMBE, of the MICCAI Society, and of the Engineering School of the University of Tokyo. He is a recipient of numerous awards, including the IEEE Robotics Pioneer Award, the MICCAI Society Enduring Impact Award, and the Maurice Müller Award for Excellence in Computer-Assisted Orthopaedic Surgery.

Peter Kazanzides received the B.S., M.S., and Ph.D. degrees in electrical engineering from Brown University in 1983, 1985, and 1988, respectively. He started working on surgical robotics in March 1989 as a postdoctoral researcher, with Russell H. Taylor, at the IBM T.J. Watson Research Center. He co-founded Integrated Surgical Systems (ISS) in November 1990 to commercialize the robotic hip replacement research performed at IBM and the University of California, Davis. As the director of robotics and software, he was responsible for the design, implementation, validation and support of the ROBODOC System. He joined the Engineering Research Center for Computer-Integrated Surgical Systems and Technology (CISST ERC) in December 2002, and currently is a Research Professor of Computer Science at The Johns Hopkins University.