Component-based software for dynamic configuration and control of computer assisted intervention systems

Release 1.0

Peter Kazanzides, Min Yang Jung, Anton Deguet, Balazs Vagvolgyi, Marcin Balicki, and Russell H. Taylor

August 1, 2011

Johns Hopkins University, Baltimore, MD

Abstract

This paper presents the rationale for the use of a component-based architecture for computer-assisted intervention (CAI) systems, including the ability to reuse components and to easily develop distributed systems. We introduce three additional capabilities, however, that we believe are especially important for research and development of CAI systems. The first is the ability to deploy components among different processes (as conventionally done) or within the same process (for optimal real-time performance), *without requiring source-level modifications to the component*. This is particularly relevant for real-time video processing, where the use of multiple processes could cause perceptible delays in the video stream. The second key feature is the ability to dynamically reconfigure the system. In a system composed of multiple processes on multiple computers, this allows one process to be restarted (e.g., after correcting a problem) and reconnected to the rest of the system, which is more convenient than restarting the entire distributed application and enables better fault recovery. The third key feature is the availability of run-time tools for data collection, interactive control, and introspection, and offline tools for data analysis and playback. The above features are provided by the open-source *cisst* software package, which forms the basis for the Surgical Assistant Workstation (SAW) framework. A complex computer-assisted intervention system for retinal microsurgery is presented as an example that relies on these features. This system integrates robotics, stereo microscopy, force sensing, and optical coherence tomography (OCT) imaging to transcend the current limitations of vitreoretinal surgery.

Contents

1 Introduction **2**

2 Component Model **3**

3 Deployment Scenarios **3**
3.1 Multi-threaded in a single process 3
3.2 Multi-process using network layer 4

4 Dynamic Component Configuration **4**

5 Tools and Utilities **5**
5.1 Tools for managing distributed applications 5
5.2 Tools for data collection and replay 6

6 Retinal Microsurgery System **7**

7 Conclusions **8**

1 Introduction

Component-based software engineering (CBSE) has been widely adopted as a paradigm for dealing with complex systems [11]. Within CBSE, the evolution and increasing complexity of software systems has focused research attention on the dynamic properties of systems, such as run-time system modification (i.e. dynamic reconfiguration) [8, 9, 2] and dynamic and flexible components [7]. These issues also apply to the development of systems for computer-assisted interventions (CAI), especially as their complexity increases due to the integration of devices such as robots, cameras, and medical imaging systems.

Proponents of component-based software often cite an analogy to electronic design, where the components (integrated circuits) have interfaces (pins) with signals that are precisely defined by the datasheets. Thus, an electronics designer can create large, complex systems by wiring together a number of these existing components. This both simplifies design and enables significant reuse from one design to the next. We note, however, that modern electronic designs often include devices such as field programmable gate arrays (FPGAs), which essentially correspond to programmable hardware. In these systems, it is now common for designers to reuse “function blocks” (or IP cores) within the FPGA design. We contend that this manner of component composition (multiple components in a single integrated circuit) can also be useful in the software domain (multiple components in a single process), especially in cases where high bandwidth and low latency are required (e.g., image feedback). Thus, it is necessary to have a framework that enables composition of components that exist within a single process or across several processes. Ideally, the mechanism (programming model) should be identical between these two cases. Furthermore, because CAI systems must often adapt to changing environments, it is desirable have a dynamic mechanism so that the system can be reconfigured as needed. These features are provided by the *cisst* open-source software framework[5, 4], available at `https://trac.lcsr.jhu.edu/cisst`.

This paper is organized as follows. In Section 2, we review the *cisst* component model. This is followed by a discussion of the different deployment scenarios (multi-threaded in a single process and multi-process in a distributed system) and a description of the dynamic component configuration capability. Section 5 presents the provided tools for interactive management of deployed components and for data collection and offline analysis. Finally, Section 6 presents an application of the *cisst* framework – a robot system for retinal microsurgery[1].

2 Component Model

A *cisst* component contains lists of *provided interfaces*, *required interfaces*, *outputs*, and *inputs*, as shown in Figure 1. 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. Each *required interface* has multiple *function objects* that are bound to command objects to use the services that the connected component provides. It may also have *event handlers* to respond to events generated by the connected component. 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. The *output* and *input* interfaces provide real-time data streams; typically, these are used for image data (e.g., video, ultrasound).

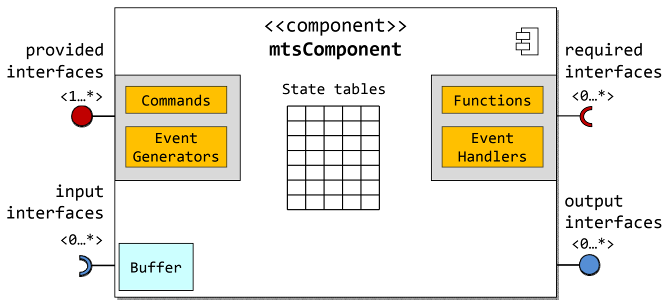


Figure 1: Overview of *cisst* base component, `mtsComponent`

3 Deployment Scenarios

The primary rationale for component-based software engineering is to support reuse of components in different system configurations. Nevertheless, many component-based systems rely on a middleware package to provide the component interconnections, which assumes that components are binary entities that are executed as separate processes. While this may be fine when connecting large-scale components, it is not ideal for building an efficient system from smaller-scale components. In those cases, it is preferable to also allow component connections within a single process. To the best of our knowledge, the *cisst* framework is unique in that it allows any component to be connected to another component in the same process or in a different process, *without any source-level modifications to any of the components*. This section presents these two deployment scenarios.

3.1 Multi-threaded in a single process

All components in the same process are internally connected to a special system component, the *Local Component Manager (LCM)*, which is unique within the process. Connection between local components is established by the LCM. The LCM also controls (e.g. start, stop, resume) the state of the components and provides dynamic component management services such as create, remove, connect, and disconnect (see Section 4). The LCM is connected to another system component, the *Global Component Manager (GCM)*, which is globally unique to the whole system. The GCM manages all LCM(s) in a system and maintains system-wide information such as a list of processes, components, interfaces, and connections for dynamic component management services. Figure 2 shows the multi-threaded architecture which has the best real-time performance. This architecture is particularly relevant for real-time video processing, where the use of multiple processes could cause perceptible delays in the video stream.

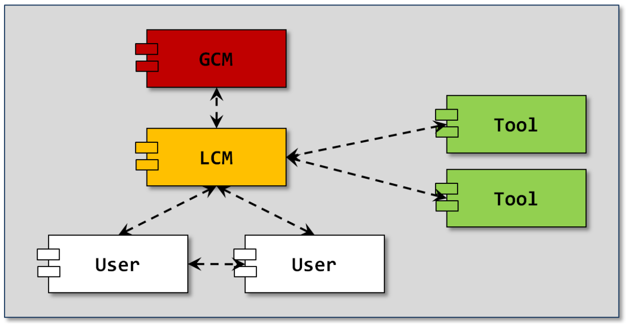


Figure 2: Multi-threaded in a single process

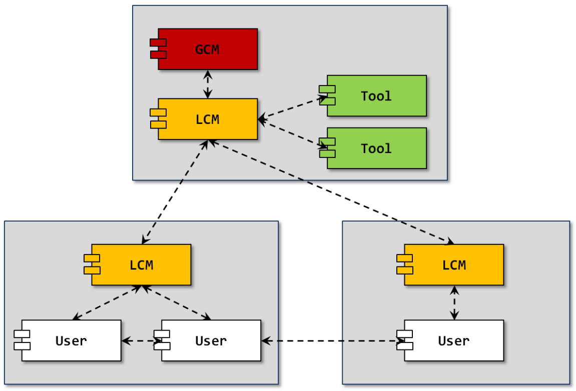


Figure 3: Multi-process (distributed) over network

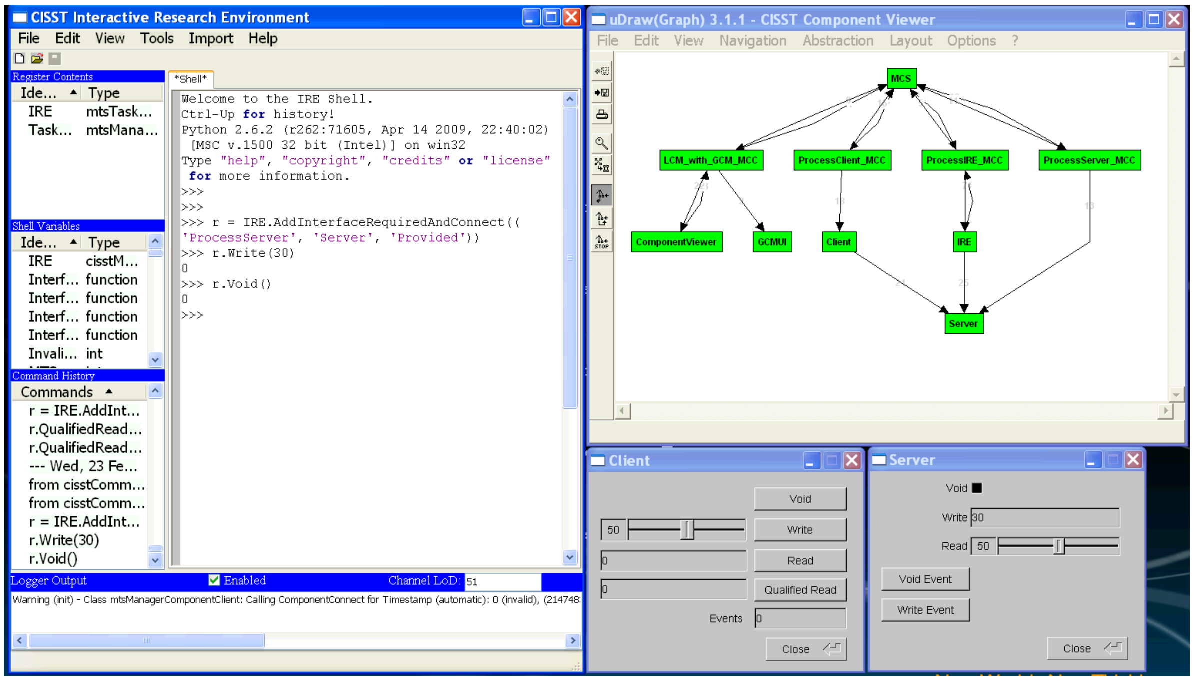
3.2 Multi-process using network layer

The multi-process architecture is achieved by extending the *local* connection between components to the *logical local* connection by the introduction of "proxy" components (i.e., the Proxy Pattern [10]). The proxy components mediate data exchange between the original components across the network. For example, if the *original* client component and the *original* server component are in different processes, the *proxy* server component is created in the same process as the client component and the *proxy* client component is created in the same process as the server component. The network layer uses the Internet Communication Engine (ICE) [3] 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.

4 Dynamic Component Configuration

The main goal of dynamic component configuration is to make a system more flexible by allowing it to be constructed by connecting and configuring dynamically created components at run-time (i.e., without recompilation or re-booting). In the *cisst* framework, these capabilities are provided by two special system components, the LCM and GCM described in Section 3. The GCM is a globally unique component in a system that provides system-wide component control and management services for the LCM(s). One LCM is created in each process (which has a unique process name) and all components in that process connect to it. Similary, all the LCMs in a system connect to the GCM. These internal connections are hidden from the application (user) layer and are automatically established when the LCM starts up. Through this chain of connections between user components and the LCM and between the LCMs and the GCM, a user component can access a set of dynamic component services such as `Create`, `Configure`, `Start`, `Stop`, `Resume`, `Connect`, `Disconnect`, and `GetState`. All the service requests made to the LCM are passed to and coordinated by the GCM. In this way, it is guaranteed that the system maintains a globally consistent state. For example, if a `Connect` request is made, the GCM checks the validity of the request (e.g. if the components specified exist, if the interfaces specified exist, if the connection is already established) and accepts it only if the request is confirmed to be valid. Then, the GCM coordinates the LCMs involved in the connection to create any required proxy objects and establish the requested connection.

As mentioned above, each user component gets connected to the LCM automatically and internally when it is registered to the system. This adds a required interface for dynamic component services, which contains function objects that invoke services such as `Create` and `Connect`. Thus, an application does not need



CISST Interactive Research Environment

Ide...	Type
IRE	mtsTask...
Task...	mtsMana...

Shell Variables

Ide...	Type
IRE	cisstM...
Interf...	function
Interf...	function
Interf...	function
Interf...	function
Invali...	int

Command History

```
r = IRE.AddInt...
r.QualifiedRead...
r.QualifiedRead...
--- Wed, 23 Fe...
from cisstComm...
from cisstComm...
r = IRE.AddInt...
r.Write(30)
r.Void()
```

```
Welcome to the IRE Shell.
Ctrl-Up for history!
Python 2.6.2 (r262:71605, Apr 14 2009, 22:40:02)
 [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license"
 for more information.
>>>
>>>
>>> r = IRE.AddInterfaceRequiredAndConnect((
'ProcessServer', 'Server', 'Provided'))
>>> r.Write(30)
0
>>> r.Void()
0
>>>
```

Logger Output

```
Warning (init) - Class mtsManagerComponentClient: Calling ComponentConnect for Timestamp (automatic): 0 (invalid), (214748
```

Figure 4: Demonstration system with four processes: GCM with Component Viewer (top right), IRE (left), Client (bottom middle), Server (bottom right). Result shown after dynamic creation of IRE required interface and connection to Server provided interface, followed by executing Void and Write commands from IRE.

any extra code to use the services. Also, this pre-defined required interface is available to the Python shell environment (the Interactive Research Environment (IRE)) and to the Component Viewer, which are described in Section 5. These two tools allow users to interactively and dynamically manipulate system configurations and invoke the dynamic component confiuration services.

An interesting aspect of this dynamic composition capability is that we can build systems from a single large binary module (executable) and/or from a set of dynamically loaded plug-in modules. For example, if the entire *cisstStereoVision* library is pre-built, packaged, and distributed as a single binary, any number of components can be instantiated within it and can build a new system. Also, if we have a binary that contains different types of sensor components, we can adaptively instantiate them depending on changing environments. As long as we have processes based on the *cisst* component framework, we can build a new system from a single binary module or across several binary modules even when the processes initially contain no components (other than the LCM).

5 Tools and Utilities

5.1 Tools for managing distributed applications

Figure 4 depicts a demonstration system that consists of four processes: the Global Component Manager, the (Python-based) Interactive Research Environment (IRE), and a simple Client and Server. The Component Viewer is shown in the upper right of Fig. 4; this component (based on the uDrawGraph program from the University of Bremen) graphically depicts all the components and connections in the system. It is also possible to dynamically change the state of the components and connect or disconnect them by interacting

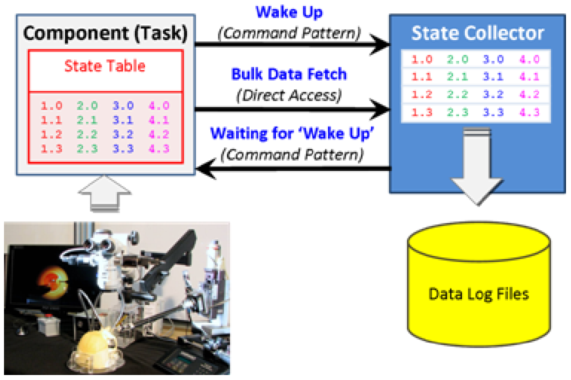


Figure 5: Overview of State Collector

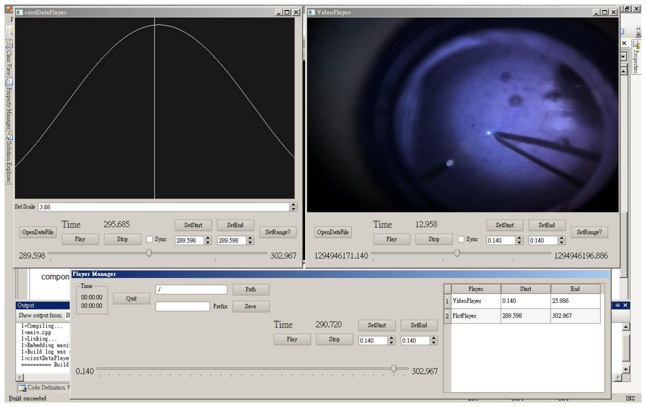


Figure 6: Data replay tool (cisstDataPlayer), showing Player Manager (bottom) and loaded players for numeric values (top left) and recorded video (top right).

with this graph. Specifically, a pop-up menu is displayed when the user right-clicks on a component; using this menu it is possible to start or stop the component or view its interfaces. The Interactive Research Environment (IRE) is shown in the left of Figure 2. Through this Python shell, it is possible to dynamically connect to any provided interface in the system, even if it is in a different process. For example, it is possible to dynamically create a required interface and connect it to the specified component's provided interface (in this case, the Provided interface of the Server component) as follows:

```
r = IRE.AddInterfaceRequiredAndConnect(('ProcessServer', 'Server', 'Provided'))
```

The new connection (between dynamically created IRE required interface and server provided interface) is shown in Fig. 4. It is now possible to invoke commands on the Server via the Python shell, such as `r.Write(30)` and `r.Void()`.

5.2 Tools for data collection and replay

The *cisst* framework includes tools for data collection, including a state collector (Fig. 5), an event collector, and a video recorder with time-stamping. The state collector uses a separate thread to efficiently store buffered data from the target task. On a multi-core computer, this allows high-bandwidth data collection with minimal (almost zero) execution overhead on the target task.

In many situations (e.g., in-vivo experiments), it is not feasible to analyze the data as it is collected. Thus, the *cisst* framework also includes a data replay tool, called *cisstDataPlayer* (Fig. 6), that can be used to analyze the volumes of multi-media data collected during a typical experiment. The basic concept is to define data player components (plug-ins) for the different types of data (e.g., collected by the state collector, the video recorder, or other collectors), which are loaded by a *Player Manager* component. All collected data is time-stamped, and each data player component can be synchronized with the Player Manager (see the *Sync* checkboxes in Fig. 6). If they are synchronized, the Player Manager can simultaneously sequence through all data sources; if not synchronized, each data player provides a control to sequence through just its own data. Subsets of data can be output as separate files for analysis by external software, such as Matlab. Conceptually, cisstDataPlayer is analogous to a standard movie editing tool, but with the ability to support any type of data, not just video, audio, and subtitles.

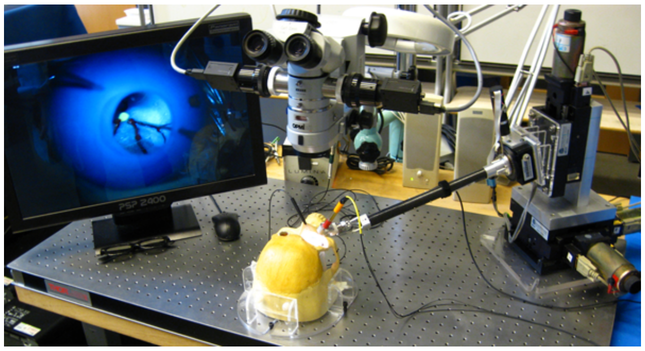


Figure 7: EyeSAW Hardware Setup

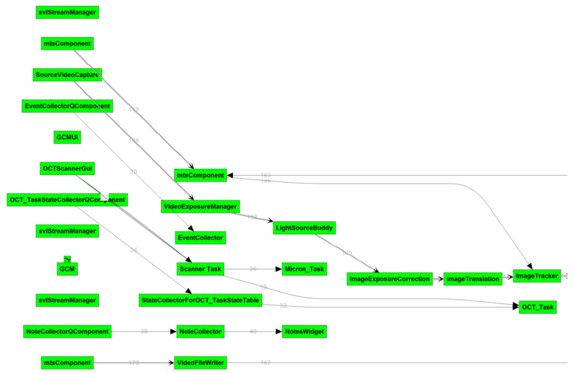


Figure 8: Partial view of EyeSAW Architecture, generated by the *cisst* Component Viewer

6 Retinal Microsurgery System

As a representative system built using the *cisst* libraries, we describe the Eye Surgical Assistant Workstation (EyeSAW). The EyeSAW is an application-specific instance of the *Surgical Assistant Workstation (SAW)* [6], which is an effort to develop an architecture for systems that combine robots and other devices to create a human-in-the-loop system. Beyond the component-based framework provided by *cisst*, the SAW project includes a 3D user interface library (for 3D visualization and interaction) as well as the definition of standard interfaces to devices, such as robots and imaging systems, used for computer assisted intervention systems.

The objective for the EyeSAW is to develop technology and systems addressing fundamental challenges to successful vitreoretinal surgery [6]. 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. Our surgical workstation system comprises a stereo video-microscopy subsystem and a family of novel sensors, instruments, and robotic devices. It is built on top of the *cisst* libraries, relying heavily on the component-based framework and real-time video library.

Fig. 7 presents an example hardware configuration that includes: (1) the SteadyHand EyeRobot, a cooperatively controlled microsurgical robot [13]; (2) an OCT subsystem that provides axial imaging and serves as a range finding sensor integrated into the shaft of the surgical instrument [1]; (3) a surgical console subsystem that captures, manipulates, and displays 3D stereo video from the surgical microscope; and (4) a technician console application (not shown) that is responsible for configuring components to provide a particular *behavior* of the whole system. Each one of these subsystems, composed of one or more components, runs on a separate machine due to hardware and software dependencies, 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 4kHz while performing computationally demanding signal processing. Simultaneously, the surgical console visualization stream runs on a separate machine and processes about 900 Mbps 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 stream, the contention for processor resources and bus bandwidth would significantly affect video frame latency and image resolution, 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 configures and monitors the 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. We recently ported the *cisst* framework to the iOS platform, thereby enabling us to use an iPad for the technician console. Fig. 8 shows some of 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 runtime by the Component Viewer described in Section 5). The B-Scan is performed by automatically scanning the robot across the retina, 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.

7 Conclusions

The *cisst* component-based software framework has been successfully used for several computer-assisted intervention systems, including the robot for retinal microsurgery presented in this paper. Because these systems often require real-time image feedback, such as video or ultrasound, the *cisst* package is unique in that it supports both robot control and real-time image processing within the same framework. This necessitates support for different execution models – robot control typically uses multiple periodic threads that exchange feedback and control information (often in a hierarchical manner), whereas real-time image processing provides the lowest latency when a single stream, consisting of a thread or thread pool, sequentially executes all components in the image feedback chain. Nevertheless, *cisst* provides a uniform interface for connecting and managing these components, whether they are deployed and used locally (within the same process) or remotely (distributed among several processes).

The *cisst* framework also provides dynamic component creation and configuration, which allows the system to be modified at run-time. This is facilitated by several general-purpose tools, such as the Python-based Interactive Research Environment (IRE) and the Component Viewer. Because CAI research systems are often used in complex environments, including in-vivo testing, the framework further includes tools for online data collection and offline analysis. Finally, an obvious benefit of an open-source component-based framework for CAI systems is the availability of an ever-increasing repository of components that provide device interfaces (e.g., robots, sensors, tracking systems, image capture, speech recognition, OpenIGTLink [12]) and processing (e.g., robot control, image filtering, graphical simulation, calibration, registration).

Acknowledgments

We thank the many developers who contributed to the *cisst* software framework. In particular, we thank Joshua Chuang for his work on the cisstDataPlayer tool. This work is supported in part by National Science Foundation grants EEC 9731748, EEC 0646678, MRI 0722943, in part by National Institutes of Health grants R01 EB007969, R42 RR019159, R21 EB009143, in part by the Swirnow Foundation, and in part by Johns Hopkins University internal funds.

References

[1] 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)*, pages 108–115, London, Sep 2009. 1, 6

[2] T. Batista, A. Joolia, and G. Coulson. *Managing Dynamic Reconfiguration in Component-Based Systems*, volume 3527 of *Lecture Notes in Computer Science*, chapter 1, pages 1–17. Springer Berlin / Heidelberg, 2005. 1

[3] M. Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, Jan-Feb 2004. 3.2

[4] 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. 1

[5] A. Kapoor, A. Deguet, and P. Kazanzides. Software components and frameworks for medical robot control. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 3813–3818, Orlando, FL, May 2006. 1

[6] 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. 6

[7] I. Kim, D. Bae, and J. Hong. A component composition model providing dynamic, flexible, and hierarchical composition of components for supporting software evolution. *Journal of Systems and Software*, 80(11):1797–1816, Nov 2007. 1

[8] M. Léger, T. Ledoux, and T. Coupaye. Reliable Dynamic Reconfigurations in a Reflective Component Model. In *Component Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 74–92. Springer Berlin / Heidelberg, 2010. 1

[9] J. Polakovic, S. Mazare, J. B. Stefani, and P. C. David. Experience with safe dynamic reconfigurations in component-based embedded systems. In *Component Based Software Engineering*, CBSE'07, pages 242–257. Springer-Verlag, 2007. 1

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

[11] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002. 1

[12] 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. *Intl. Journal of Medical Robotics and Computer Assisted Surgery*, 5(4):423–434, Dec 2009. 7

[13] 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. 6