The *cisst* libraries for computer assisted intervention systems

Release 1.00

Anton Deguet, Rajesh Kumar, Russell Taylor, Peter Kazanzides

July 25, 2008

Johns Hopkins University, Baltimore, MD

Abstract

Computer assisted intervention (CAI) systems require the integration of an increasing number of devices, including medical monitors, sensors, tracking devices and robots. This complexity makes applications harder to develop, more difficult to debug and the accumulation of ad hoc interfaces reduces the overall portability. We describe a set of libraries, the *cisst* libraries, developed at the Johns Hopkins University to address some of the problems encountered when integrating devices for CAI. We focus on three main characteristics of the *cisst* libraries: software architecture, multi-threading and CAI specific interfaces.

Latest version available at the Insight Journal [http://hdl.handle.net/1926/1338]

Contents

1 Introduction

Computer assisted intervention (CAI) systems integrate disparate devices such as medical monitors, sensors, imagers, tracking devices and robots. The goal of the *cisst* libraries is to support a wide range of devices while preserving flexibility. In particular, we wish to support interchangeability of devices that meet the minimum requirements for the application. For example, in research on collaborative control or visualization enhancement for teleoperated interventions, the nature of the patient side manipulators or master arms might not be a critical factor. Figures 1 and 2 show two different research platforms that can be used for telesurgery research. The first setup utilizes a da Vinci system with a research interface, whereas the second setup replaces the da Vinci slave arms by a pair of custom research manipulators (Johns Hopkins "snakes" [8]) and replaces the da Vinci master arms by a Northern Digital Polaris tool and a SensAble Omni. The flexibility of the *cisst* libraries make it possible to re-use the same research component (the `CollaborativeControl` task in figures 1 and 2) with different devices as long as they provide the required functionalities. The OROCOS project [2] provides similar capabilities, though it has chosen to focus on real-time control and robotics with real-time operating systems; in contrast, *cisst* also supports conventional operating systems such as Windows.

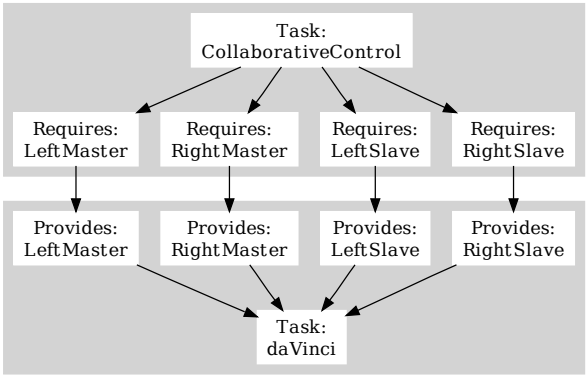

Figure 1: Teleoperation setup using a da Vinci

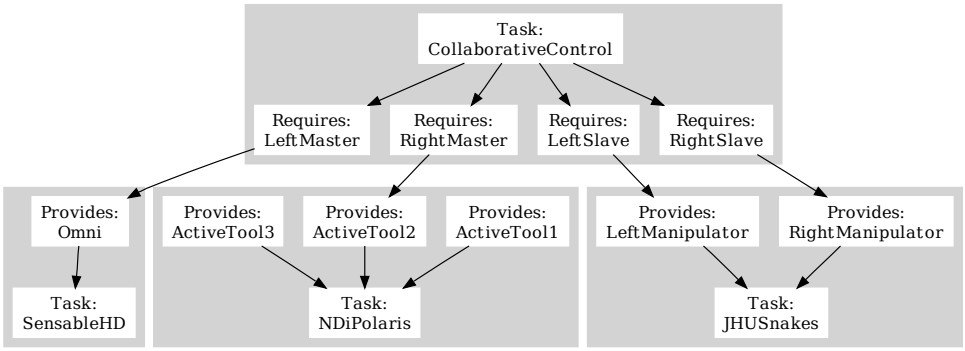

Figure 2: Teleoperation setup using the JHU Snakes, a Northern Digital Polaris tool and a SensAble Omni

2 Architecture

A typical CAI application consists of multiple concurrent tasks (e.g., figures 1 and 2). The *cisstMultitask* library provides a safe and efficient framework for multi-threading (see Section 3 for details); this can be extended to multi-processing via the use of middleware. The design adopts concepts from component-based software engineering: components must be reusable, composable, and encapsulated. Whether the library is truly "component-based" depends on the definition of this term, as discussed in [5].

In our introductory examples, we replaced one of the master arms with a tracking device. These two systems are different by nature but have a common functionality (i.e., provide the 3D position of the surgeon's hand). In a typical Object Oriented design, one would argue that a robot class can be derived from a tracker class. The issue is that a tracker also provides features not available in a robot. Neither one is a subset of the other in terms of functionality, which precludes the use of inheritance. With a component based approach, the user task dynamically queries whether the device provides a *required* functionality and is *provided* a means to use that functionality. In *cisstMultiTask*, functionalities are represented by *commands* as defined by the Command Pattern [4].

2.1 Commands

All communications between tasks and devices in *cisstMultiTask* are performed using commands to ensure thread-safety and component encapsulation. The actual code (actions) of a command is implemented by the "provides" task (or device), the user retrieves a command object and executes it when needed. What happens then varies: on a distributed or multiprocess application, all commands should be serialized, queued, dequeued and finally executed. In a multi-threaded implementation, serialization is not needed and one can avoid queuing in some cases, such as when reading from a thread-safe circular (or ring) buffer. These details of execution are built-in and from a user point of view, all commands have the same API (e.g., trigger with `command.Execute(data)`). Note that *cisstMultiTask* also support events and event callbacks, which also rely on the command pattern.

The two main restrictions imposed by the use of commands are: 1) parameters must be derived from a base type, and 2) a finite number of signatures are supported:

- Void commands: These do not have parameters and can be used to send commands such as `Stop`.
- Read commands: These take one parameter (passed by reference). The parameter is a placeholder that is used by the called ("provides") task to return information to the user task.
- Write commands: These take one parameter (passed by const reference). The parameter represents the payload sent to the called task.
- Qualified read commands: These require two parameters (the qualifier or payload is passed by const reference and the placeholder by reference).

The fact that one can only support a limited number of signatures can be perceived as a drawback but it helps to enforce the consistency of interfaces by defining the payload and placeholder as single objects. The *cisstParameterTypes* library contains the most common parameter types (see Section 4.2).

2.2 Interfaces

Another key concept of our design is the possibility to have multiple interfaces. This is used to separate different functionalities (e.g. a da Vinci master and the foot pedals should be seen as two different entities) or to allow the same functionality to be provided multiple times (e.g. an optical tracker with multiple tools). If we consider the example of the Polaris, the proper implementation would have a main interface to perform the setup functions (configure, reset, explore ports, beep) and one interface per tool. If all tools were combined into a single interface, the commands would have to either be named differently (`GetPositionCartesian1`, `GetPositionCartesian2`, ...) or have a different signature to specify the tool index as a parameter. In both cases, this would break the interface consistency and prevent reusability.

2.3 Self-describing components

Each component (task or device) owns a set of interfaces and each interface owns a set of commands. These are populated dynamically and can be searched at runtime. Furthermore, all parameter types must be derived from a base class that provides the class name (string) and a means to create new objects (object factory pattern). Thus, all components are self-describing; if a new component is added to the system in binary form, it is possible after linking (or dynamic loading) to find its list of interfaces and their command signatures. The application can then determine dynamically if the component provides all required functionality.

2.4 Interactive Research Environment (IRE)

Early on we recognized the need for an interpreted language to interact with our C++ libraries [6]. We chose Python and decided to use SWIG to wrap our libraries. The main uses of the scripting interface are:

- Rapid prototyping, i.e. write and test simple functions directly on a device (e.g., robot) without having to stop the device, recompile, restart the device and restore its state.
- Runtime modifications of the application state. This allows users to modify data on the fly without having to restart the application and add new hooks to the code. This is extremely convenient when something unexpected happens in the middle of a lengthy experiment.

For our base libraries, SWIG parses the library header files and extracts the list of available symbols (functions, classes and their methods). In *cisstMultiTask*, task interfaces are populated at run time; hence SWIG can only wrap the base methods, such as `task::GetInterface` and `interface::GetCommand`. It would be tedious for the user to manually invoke these commands (in Python) to retrieve all the interfaces and the lists of commands and events for each interface. But, because all components are self-describing objects, we can augment the Swig-generated proxy with a Python method that automatically populates the Python object's dictionary (i.e. its list of methods and data members). Once all the interfaces and commands have been added to the task, one can trigger the command using an intuitive Python method call:

```
position = prmPositionCartesianGet()
polaris.Tool1.GetPosition(position) # Tool1 is an interface
print position.Status()
```

The main result is that users do not have to do anything to use their classes in Python, i.e. they do not need to create a SWIG interface file, generate the wrappers, compile them and add them to a dynamic library.

Another plus is that commands are thread safe, which allows users to manipulate a task from the Python thread without any problem.

3 Multi-threading

The previous sections discussed the need for concurrent processing. The trend towards multi-core processing architectures motivated us to consider an efficient multi-threading approach, which can easily be extended to multi-processing. This section presents features of the *cisst* libraries that facilitate the development of safe, efficient and flexible multi-threaded applications.

3.1 Thread safety

The first critical aspect of multi-threaded applications is thread safety. This requirement has been somewhat ignored in many existing libraries and toolkits. A prominent example is the free version of LAPACK distributed on netlib.org. This library is often used in open source packages but neither the C version nor the original FORTRAN version is thread-safe. The FORTRAN version maintains global variables in some basic routines such as SVD and the C version also contains static variables introduced by f2c. We use the thread-safe, FORTRAN90 based, LAPACK3e for our numerical routines[1] [1]. We provide a binary distribution of the FORTRAN code (*cisstNetlib*) as well as C++ wrappers (*cisstNumerical*) that can handle memory allocation and verify that all parameters are set correctly. On a side note, *cisstNumerical* takes advantage of the versatility of the *cisstVector* containers by using memory overlay and different storage orders (C++ row major and FORTRAN column major, [7]).

3.2 Efficiency

Another aspect of multi-threading is ease of use and efficiency. At the low level, *cisstOSAbstraction* provides a simple class with basic thread manipulation (creation, priority, yield, unique ID, etc.) for different operating systems (Linux, RTAI-Linux, Windows, Mac OS X, Solaris). At a higher level, *cisstMultiTask* provides non-blocking mechanisms to communicate between tasks (i.e. threads), as illustrated in figure 3.

When a task needs to retrieve some information from another ("read"), it is possible to use double-buffering, i.e. the working task maintains two (or more) states, using the current one for its computation and providing a previous one for any other thread to read. Instead of using a mutex when the task needs to copy the current state to the old one, we can use a circular buffer with at least 3 elements (the *State Table*) and increase the "current" index before the "previous" one[2].

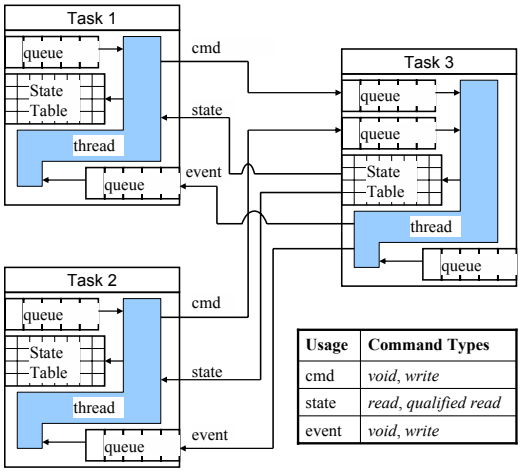

Figure 3: Inter-task communication

[1] Also freely available from netlib.org

[2] This relies on the fact that pointer increments are atomic.

When a task needs to send a message to another ("write"), the best solution is to use a message queue. In this case, *cisstMultiTask* relies on thread IDs to dedicate one queue per thread. Thus, all queues are single reader and single writer, thereby avoiding the need for a mutex to control write access to the queue. Thread safety between the reader and the writer is also based on atomic pointer increments.

3.3 Memory

Finally, it is important to control how memory is handled to prevent excessive allocations and de-allocations (paging can be a problem in real-time operating systems). The *cisst* libraries are designed to avoid behind-the-scene memory allocation and provide mechanisms to use memory in place (e.g. method `a.SumOf(b, c)` can be used instead of operator `+` to avoid allocation of a temporary variable). In *cisstMultiTask*, all classes perform their memory allocations (e.g., allocating queues) in the configuration phase.

4 API for Computer Assisted Intervention Devices

cisstMultiTask defines a framework for devices but the API itself still has to be defined within this framework. The main characteristics of the API are:

- Interfaces don't have to be exactly the same between devices.
- Commands and events must have the same signatures (name AND parameters) to make devices interchangeable.
- Because a limited number of signatures are supported ("read", "write", ...), the payload is likely to be a compound object (i.e. a C++ class with multiple data members).

These traits are typical of any component-based software. For the type of message (payload) passed between devices and tasks in CAI applications, we need to give special attention to Cartesian coordinates.

4.1 Transformation Manager

Most of the devices used in CAI provide and use 3D coordinates. When these devices are integrated within an application, the first step is to register the different reference frames with respect to each other. In our second example (fig 2), the positions provided by the Polaris will be defined with respect to the camera while the Omni will provide positions with respect to its base. These two will likely be meters away from each other and don't represent the relative positions of the surgeon's hands (which themselves should very likely be defined with respect to the camera frame). The first step in most applications is to calibrate and register the different reference frames used by the devices and application. This usually leads to a tree where the different nodes represent the key frames used by the devices and the edges represent the current transformations. A similar feature can be found in *AL* [3].

The *cisst* libraries provide a "transformation manager" (singleton) that manages two types of transformations, *fixed* and *dynamic*. The first type corresponds to a constant transformation between frames (e.g. set once after registration) while the second corresponds to a moving object (e.g., tracking device, robot end effector). Dynamic transformations contain a *cisstMultiTask* read command. When a user requests a transformation between two frames, the transformation manager searches for the path in the tree and multiplies

the relative transformations along the path. If a node is static, the transformation is directly available. If a node is dynamic, the read command is used to retrieve the current transformation from the device.

4.2 Parameter types

In our framework, parameters are an integral part of the API and must be carefully defined. They depend on the devices in the sense that they should carry the information provided by the device but they should be independent of the device implementation to guarantee the component encapsulation (specification based). For example, typical tracking devices provide not just a position, but also a metric that expresses the quality (or confidence) of the provided position. The range of values for this metric can vary between devices, however, and many devices that provide a position (such as robots) do not provide such a metric.

Based on our experience, we defined the following guidelines to define parameter types for 3D related commands:

- All parameters must contain a time stamp (time of acquisition or request).
- Parameter types for read and write commands (e.g. `PositionGet` and `PositionSet`) are likely to be different. A "read" command should use the `PositionGet` parameter type which will only contain the 3D position. Meanwhile, a "write" command should use `PositionSet` which will contain the goal position as well as the motion parameters (velocities, accelerations, synchronization flag) used by the device (e.g. robot).
- All data types related to 3D positions must provide two frames as defined in the transformation manager. The first is the reference frame and the second represents the frame that should be moved ("write" commands) or tracked ("read" commands).

Basic types are provided in the *cisstParameterTypes* library. This includes "Set" and "Get" for positions, velocities and accelerations of both Cartesian and joint commands.

It is important to note that the payload of events also needs to be standardized across devices (event payloads are command parameters in *cisstMultiTask*). These parameter types follow the previous guidelines. For example, a button press will need a time stamp.

The *cisstParameterTypes* library is populated as new types of devices are added to our framework and refined as needed.

4.3 Command names

For the command names, we are populating a dictionary of valid signatures. Our goal is to put in place a tool to dynamically check that all newly added devices use only authorized signatures. The *cisst* CppUnit based test suite runs nightly and uses CDash to report errors. This should detect rogue interfaces in a timely manner.

5 Conclusions

We have presented elements of the *cisst* library that facilitate the development of safe and efficient multi-tasking (multi-threaded) software, using concepts from component-based software engineering to achieve

loose coupling between tasks. Each task contains one or more interfaces that are self-describing, in that they can dynamically provide a list of supported commands and associated parameters. This library can facilitate the development of computer assisted intervention (CAI) systems, with the ability to dynamically configure the application software to use available hardware, such as diverse robots and other devices. In particular, it is enabling the concurrent development of the Surgical Assistant Workstation (SAW), an application framework that supports research with information-enhanced telesurgical robot systems [9].

The *cisst* software is available under an open source license. Currently, a subset of the software can be downloaded from www.cisst.org/cisst. Additional elements will become available when the implementation and documentation are sufficiently mature to support widespread dissemination.

Acknowledgments

The *cisst* libraries have been created by many individuals; primary contributors include Ankur Kapoor, Ofri Sadowsky, Balazs Vagvolgyi, and Daniel Li. This work is supported in part by National Science Foundation EEC 9731748, EEC 0646678, and MRI 0722943.

References

[1] E. Anderson. LAPACK3E – a Fortran 90-enhanced version of LAPACK. Technical report, Science Applications International Corporation, 2002. 3.1

[2] 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)*, pages 2776–2771, Taipei, Taiwan, 2003. 1

[3] R. A. Finkel, R. H. Taylor, R. C. Bolles, R. P. Paul, and J. A. Feldman. AL, a programming system for automation. Technical Report CS-74-456, Stanford University, Stanford, CA, USA, 1974. 4.1

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, Jan 1995. 2

[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, 2006. 2

[6] P. Kazanzides, A. Deguet, A. Kapoor, O. Sadowsky, A. LaMora, and R. Taylor. Development of open source software for computer-assisted intervention systems. In *MICCAI Workshop on Open-Source Software*, Insight Journal: http://hdl.handle.net/1926/46, Oct 2005. 2.4

[7] O. Sadowsky, D. Li, A. Deguet, and P. Kazanzides. Multidimensional arrays and the nArray package. In *MICCAI Workshop on Open Science*, Insight Journal: http://hdl.handle.net/1926/553, Oct 2007. 3.1

[8] N. Simaan, R. Taylor, and P. Flint. High dexterity snake-like robotic slaves for minimally invasive telesurgery of the upper airway. In *MICCAI*, pages 17–24, Sep 2004. 1

[9] 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*, Insight Journal: http://hdl.handle.net/1926/???, Sep 2008. 5