Development of open source software for computer-assisted intervention systems

Peter Kazanzides, Anton Deguet, Ankur Kapoor, Ofri Sadowsky, Andy LaMora, and Russell Taylor

Department of Computer Science, Johns Hopkins University, USA, `pkaz@jhu.edu` *

Abstract. We are developing open source software for computer assisted intervention systems. Our primary experience has been with medical robots, but the concepts (and software) apply to many physical devices that interact with the real world. The real-time performance requirements permeate all levels of our software, including common tools (such as logging, class and object registers), vectors, matrices and transformations. Our software libraries are written in C++, but are also accessible from Python, which provides a convenient environment for rapid prototyping and interactive testing.The real-time support includes a device (hardware) interface and a task library. Device-specific modules such as robot servo control and trajectory generation can be provided by tasks or by external devices. Ultimately, we intend to provide a framework that supports extension via dynamically loaded plug-in modules. Our development process utilizes a multitude of open source tools, including CVS, CMake, Swig, CppUnit, Dart, CVSTrac, Doxygen and LaTeX. These tools help to ensure compliance with our software development procedure.

1 Introduction

Software development at the Engineering Research Center for Computer Integrated Surgical Systems and Technology (CISST ERC) at JHU initially focused on a library for Computer-Integrated Surgery (*CIS*) application development, a common interface to different tracking systems (*cisTracker*) and a library for Modular Robot Control (*MRC*). Over the past three years, we have undertaken a major redesign of this software, now called the *cisst package*, and plan to make it available under an open source license at www.cisst.org. The redesign effort has focused on portability (different operating systems and compilers), maintainability, real-time compatibility and establishment of a testing framework. In addition, we adopted a more formal development process to improve the quality (and clinical certifiability) of the resulting software.

A prime motivation for the development of the *cisst package* has been our increasing need to implement novel control algorithms for new interventional systems, such as a robot for minimally-invasive throat surgery [1]. This was not feasible with the original MRC library because it relied on intelligent hardware
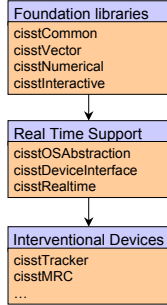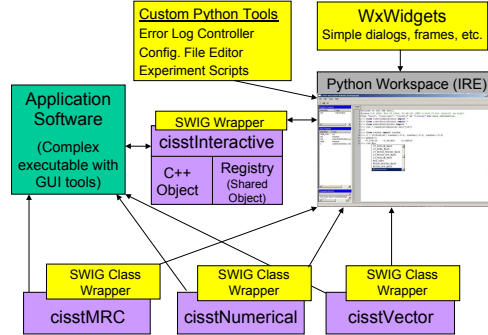

Fig. 1. CISST Libraries

Fig. 2. Interactive Research Environment (IRE)

(a motion control board) to provide the low-level real-time control. Our vision for *cisstMRC* is that it should transparently allow any control function to be provided by intelligent hardware or by a software task coupled with non-intelligent (I/O) hardware. Furthermore, we wish to provide a well-tested core framework that can be dynamically extended by plug-in modules to allow the software to be adapted to different hardware and to allow researchers to add new real-time functions or replace existing ones.

This paper focuses on the open source *foundation libraries* and *real time support* that have been developed for the *interventional device* software. We believe that our software uniquely combines the real-time performance necessary for controlling physical devices, such as robots, with the core functionality needed for computer-integrated surgery applications. The *cisst package* does not include medical image visualization or processing and therefore complements existing open source software such as VTK and ITK and toolkits built on top of them (e.g., 3D Slicer and IGSTK). There are a few open source robot control packages, such as OROCOS (Open RObot COntrol Software)), Orca, Modular Controller Architecture Version 2 and Microb (Module Intégrés de Contrôle de ROBots), but these were either not suitable or not available in time.

2 The CISST Package

The *cisst package* consists of several software libraries that are grouped into the categories of *Foundation Libraries*, *Real Time Suport* and *Interventional Devices* (e.g., trackers and robots). Figure 1 shows a hierarchical view of these categories and the following subsections provide details for selected libraries.

2.1 cisstVector (Foundation Library)

The *cisstVector* design was motivated by the desire for an efficient implementation of fixed-size vectors, matrices and transformations that is suitable for real-time use. Most other vector libraries use dynamically allocated memory to store

the vector elements, which is not ideal for real-time computing. An even greater number use loops as the underlying computational engine, which is not efficient for small vectors. Our goal in the development of *cisstVector* was to achieve high computational efficiency by using stack-allocated storage, and by replacing loop mechanisms by templated engines, defined using recursive template metaprogramming [2]. The templated definition enables us to define vectors of different sizes and types and to apply the same operations to the vectors in a consistent form. We identified a small number of recursive engines that would provide all the operations that we would want to perform on and between vectors. For completeness, *cisstVector* also provides dynamically allocated vectors and matrices.

A special feature of *cisstVector* is that it allows matrices to be stored in either row- or column-major order, to accommodate both C-style and Fortran-style two dimensional arrays. This provides convenience to C/C++ programmers, and at the same time integration with existing numerical packages based on Fortran, such as C LAPACK. In addition, *cisstVector* supports direct operations on subregions (slices) of vectors and matrices.

2.2   cisstInteractive (Foundation Library)

The *cisstInteractive* library provides the structure for embedding a Python-based interactive shell, the Interactive Research Environment (IRE), into our C++ programs. The IRE uses *wxWidgets* for Python to provide the GUI features and relies on SWIG to automatically wrap the C++ libraries for Python (see Fig. 2). It also provides an object registry that enables the Python and C++ software to share objects. This is especially useful when embedding the Python interpreter in an application because it allows the user to modify C++ objects from the Python interpreter.

2.3   cisstDeviceInterface (Real Time Support)

This library defines the *ddiDeviceInterface* class, which provides the hardware interface. A novel feature of our design is that we do not use class inheritance to specialize the device interface to different types of devices. Conventionally, one would define a "tracker interface class" with methods such as *GetPosition*. A "robot interface class" would be derived from it, adding methods such as *MoveToPosition*. The rationale here is that a robot provides all the features of tracking systems and adds others. Our approach addresses the same rationale with a different answer. We developed a mechanism for dynamic interface query, where a caller can query an object for the methods it "Provides", where the method name is represented by a string. To preserve runtime efficiency, we implemented the *command pattern* [3]. This approach is more flexible than a class hierarchy, but requires careful maintenance of, and adherence to, a "dictionary" of command names and their associated data types. The dynamic nature of this design implies that some errors, such as attempting to use a device interface that

does not implement a required command, can only be detected at runtime. Unlike the inheritance paradigm, however, our approach does not require multiple inheritance to provide multiple interfaces for a single object.

2.4 cisstRealTime (Real Time Support)

The *cisstRealTime* library provides the features needed by software that must interact with the physical world in a real-time manner. We decided to use RTAI (Real Time Application Interface)) with Linux as our real-time operating system because it is open source, appears to be well supported and provides useful features such as sharing of data between real-time and non-real-time code. We created a *cisstOSAbstraction* library so that our software can be portable to other operating systems. We also support non-real-time operating systems, such as Windows and Linux, though without the real-time guarantees.

We desired a software architecture that allows any control function to be transparently provided by a real-time software thread or by an external (intelligent) device. We achieved this by defining the classes shown in Fig. 3. The *rtsTask* class provides the basic real-time task. It is derived from *ddiDeviceInterface* and also contains a *ddiDeviceInterface* object. This allows us to define identical interfaces to both a real-time task, coupled with a non-intelligent I/O device, and an actual intelligent device; i.e., they are both instances of the *ddiDeviceInterface* class) and are both accessed using the *command pattern* described earlier. This concept is illustrated in Fig. 4, which shows two potential implementations of a robot force controller. The top line shows the implementation using non-intelligent hardware, where three software tasks provide the application thread, force controller and servo control. The bottom line shows the implementation using an intelligent motion controller, where only the first two software tasks are required. Note that our architecture allows exactly the same two application and force controller tasks to be applied in both cases – the specific implementation of the servo control, whether by a task or by an intelligent board, is hidden from the higher layers.

The *rtsTask* class also includes an *rtsStateDataTable* object. The state data table is a two-dimensional array that is indexed by "data id" and "time" and contains all persistent data in the real-time task. The "time" indexing provides a snapshot of the history of the real-time system and can be used for data collection as well as for debugging (i.e., to provide a "flight data recorder" functionality). It also solves the mutual exclusion problem between the real-time and non-real-time parts of the system (similar to a double-buffering technique).

3 Development Process

Our development process uses the following open source tools (see Fig. 5):

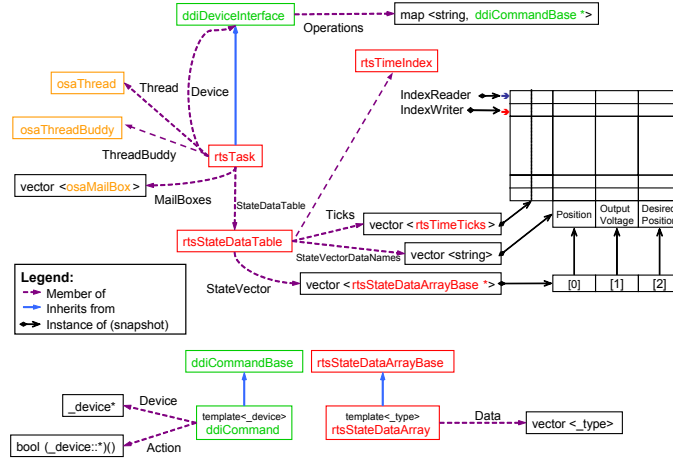1. CVS (Concurrent Versions System) [4] for source code and document control.

Fig. 3. Devices and Tasks: Class Collaboration

2. CMake for cross-platform builds. CMake generates the appropriate compiler-specific makefiles/projects/workspaces/solutions from compiler-independent configuration files.
3. CppUnit and PyUnit to provide a unit testing framework for our C++ and Python software, respectively.
4. Dart for automated, distributed tests. We routinely use "Experimental" builds and plan to add multiple "Nightly" builds.
5. Doxygen to automatically extract (specially-formatted) documentation from the source code and create class diagrams, dependency graphs and other design documentation in HTML and LATEX formats.
6. SWIG to generate wrappers for interpreted languages such as Python.
7. CVSTrac to manage bug tracking and feature requests.

We rely on manual creation of requirements documents, high-level design documents and user guides/tutorials. We chose to prepare these documents using LATEX[5] because it is text-based and therefore more amenable to change control. Our build process includes "compilation" of the LATEX documents into PDF and HTML formats (see Fig. 5).

4   Summary

Despite the existence of some computer assisted intervention systems, research is hindered by the lack of open source software that can be certified for clinical use. We endeavor to address this need by making the *cisst package* available as open source software in the next few months. At the time of this report, the *foundation libraries* (*cisstCommon*, *cisstInteractive*, *cisstVector* and *cisstNumerical*)
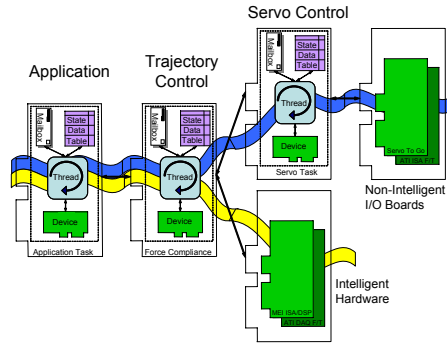
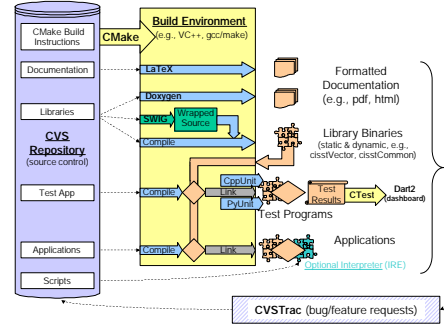**Fig. 4.** Robot Task Example   **Fig. 5.** Software Development Tools

are mature and ready for general use. The *real-time support* (*cisstOSAbstraction*, *cisstDeviceInterface* and *cisstRealTime*) are in beta testing, and the *interventional devices* (*cisstTracker* and *cisstMRC*) are in active development.

References

1. Simaan, N., Taylor, R., Flint, P.: High dexterity snake-like robotic slaves for minimally invasive telesurgery of the upper airway. In: MICCAI. (2004)
2. Veldhuizen, T.: Using C++ template metaprograms. C++ Report **7** (1995) 36–43 Reprinted in C++ Gems, ed. Stanley Lippman.
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
4. Fogel, K.: Open Source Development with CVS. Coriolis Open Press (1999)
5. Lamport, L.: LaTeX — A Document Preparation System. Addison-Wesley (1985)