DiVA : Distributed Visualization Framework
Component Interface Workshop Findings Document

July 8-9, 2003
Eugene, Oregon

Contributors

Polly Baker, Indiana University
Wes Bethel, LBNL
John Clyne, NCAR
Sam Fulcomer, Brown University
Bryce Hathaway, UC Davis
Jim Kohl, ORNL
Pat Moran, NASA-Ames
Steve Parker, University of Utah

Table of Contents

1 Overview

The focus of the July 2003 workshop was on the subject of component interface technology within the context of a framework for remote and distributed, high performance scientific visualization. The workshop was scheduled to occur adjacent to a quarterly meeting of the CCA Forum. The workshop participants included two visualization experts who are part of the CCA Forum. These two experts provided a great deal of background technical information about CCA component interface technology, and led group discussions that focus on component interface issues. We built upon the requirements findings and definitions described in the findings documents from the April 2003 and June 2003 DiVA workshops, and those findings and requirements are not repeated here.

Several commonly used component architectures and interface models were discussed. Of these, only CCA meets two of the requirements for a component-based visualization framework: language independence (and tools supporting the primary languages of interest to our community) and high performance. CCA targets distributed and parallel applications, and permits higher performance *direct connections* (shared memory) for components living in the same address space. Its interfaces may be developed in the component's native language (e.g., C++ or Python), or CCA's Scientific Interface Definition Language (SIDL) may be used to specify a language independent description of the component interfaces. The SIDL language, which provides support for scientific data constructs, is then converted to a target language by a translation compiler (Babel), outputting a component shell to be completed by the component developer.
One of the findings of the July workshop that cannot be overstated is the need for interested parties from the visualization community to "standardize and generalize" interface definitions for many common visualization tasks. Such interface definitions include identifying high-level functional units, interaction patterns, and general descriptions of data objects that are needed by the components. In developing such definitions, one primary objective should be reuse and incorporation of existing tools and technology wherever feasible. While component technology may facilitate interface development, and even encourage well defined interfaces, its use is no guarantee of software reuse. A high degree of reuse will only be obtained through broad community involvement and through provision of resources that facilitate contributions and access to the software.

The document organization roughly parallels the discussion and brainstorming topics from the July 2003 workshop: we first discussed the issue of frameworks in general in order to better understand the relationship of CCA in a broader context. There exist several "industry standard" or more mature and widely used component interfaces. We next discussed the different types of frameworks developed by CCA working groups. Each focuses upon a different area of capability, and none is "all encompassing" nor does one entirely meet the needs for a DiVA. The next section focuses upon some of those needs within the context of identifying what CCA provides, what it doesn't provide (but should), and what it will probably never provide. We next present some discussion on the topic of "CCA-izing" the Visualization Toolkit (VTK). We then go through the process of attempting to define a CCA-compliant interface for an isosurface component using alternative interface presentations and execution models (streaming vs. in-core). Finally, we conclude with some general recommendations for future growth.

2 Language-Neutral Component Interface Specification

Component systems are often specified with the aid of an interface description language (IDL). Existing IDL's typically exhibit trade-offs between performance and the protection and isolation of symbol and address spaces. While most are intended to address platform and language interoperability, their design places some constraints on this flexibility.

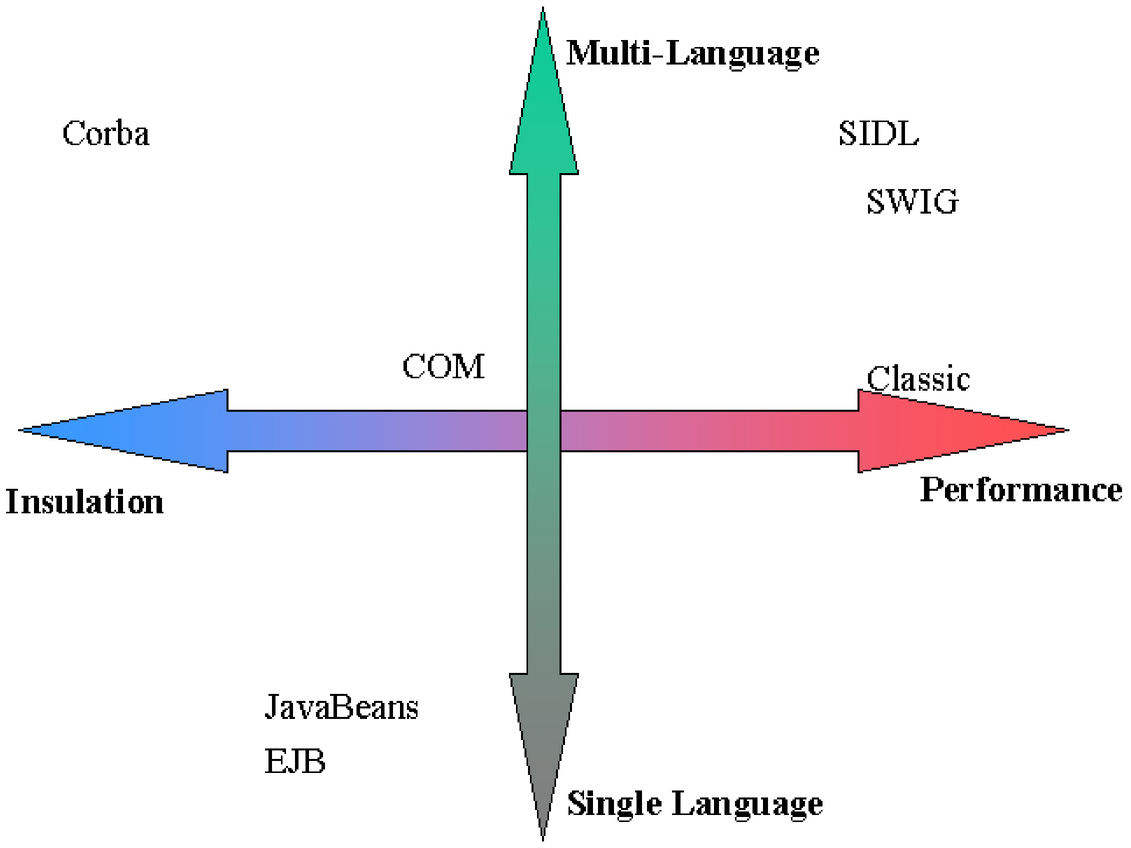
The interface descriptions discussed in the workshop, in descending order of performance, are:

1. SWIG, the Simplified Wrapper and Interface Generator (http://www.swig.org/), provides an interface generator/compiler intended for interfacing C/C++ codes with modules written in other languages. It does not provide an IDL per se, but generates interfaces based on header files. Neither does it provide a component architecture or framework (nor does it impose protection/isolation or semantic restrictions on the interface). SWIG currently supports C/C++ interfaces to Tcl-8.0+, Python-1.5+, Perl-5.003+, JavaJDK-1.1+, Ruby, Mzscheme and Guile.
2. CCA "Classic." CCA "Classic" component interfaces are written in a native language, in contract to those written in some other Interface Definition Language (see SIDL, below). The CCA Classic "IDL" is a set of subroutine calls that create the interface. The primary advantage of a Classic interface over a SIDL interface is the ability to pass objects between components that are built from language-specific features, such as STL-based class libraries written in C++.
3. SIDL, Scientific Interface Description Language – "SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components. SIDL also provides other common features that are generally useful for software engineering, such as enumerated types, symbol versioning, name space management, and an object-oriented inheritance model similar to Java." - from the LLNL/CASC description of the SIDL compiler, BABEL[1]. SIDL provides for direct interaction of components sharing the same namespace (an important distinguishing performance characteristic). SIDL provides mappings to C, C++, JAVA, F77, F90 and Python.
4. CORBA IDL - CORBA (see the CORBA FAQ[2]) provides a system specification and component framework supported by its own IDL. It does not provide for shared address space execution of components; rather, it isolates address space and provides data exchange through the use of buffered copies. This contributes to improved reliability, but at the cost of performance. It is intended primarily for small data transactions and distributed (i.e., networked) client/server systems. CORBA IDL

[1] http://www.llnl.gov/CASC/components/babel.html
[2] http://www.omg.org/gettingstarted/corbafaq.html

supports mappings to C, C++, JAVA, COBOL, Smalltalk, Ada, Lisp, Python and (CORBA) IDLScript.

5. Microsoft's Common Object Model (COM) (see the COM Technical Overview[3]) provides a component architecture in some ways similar in structure and performance to CORBA. It provides for complete isolation and protection between components, with buffered copy of data for data exchange. It uses the Microsoft Interface Definition Language (MIDL), which is an extension of the Open Software Foundation (OSF) IDL, and provides a translation compiler for C++ and Java.
6. Enterprise JavaBeans (EJB) (see EJB Overview[4]), is the server side component architecture for the Java2 Enterprise Edition (J2EE) platform. J2EE provides for CORBA compliance and interoperability, but does not provide for translation to languages other than Java.
7. Java Beans (see Java Beans overview[5]) provides a component architecture for integration primarily with application containers (e.g., Internet Explorer, Visual Basic, Microsoft Word, Lotus Notes, etc.). It does not provide for translation to languages other than Java.



[3] http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_comppr.asp
[4] http://java.sun.com/projects/ejb/index.html
[5] http://java.sun.com/products/javabeans/

3 CCA Frameworks

The concepts of “component” and “framework” are distinct, yet sometimes confusing. The term “component” refers to a functional unit of software with well-defined interfaces. A CCA component is one that implements a CCA-compliant interface. In contrast, a “framework” is the vehicle for executing components.

From the various groups within the CCA, four different frameworks have emerged. Each focuses on a different aspect of component deployment. One focuses on execution of distributed components, while another provides the capability to execute parallel components. Yet another is strictly SIDL/Babel-compliant, yet is still in a very early stage of development. The four frameworks from the CCA groups are:

Ccaffeine, from Sandia-California, is the primary CCA implementation that is used in the CCA tutorials as well as several applications. It is SPMD parallel, and implements “Classic” (native language interface implementations) and Babel-style (interfaces specified in language independent SIDL code) components. Ccaffeine is readily available from the cca-forum website.

SCIRun2, from Utah, is a CCA implementation that implements SPMD, multithreaded components. However, some portions of it are still experimental and it is not yet broadly distributed or supported. It implements Babel-style components as well as a custom SIDL-based version that supports distributed computing.

XCAT, from Indiana, specializes in Grid-based distributed computing. It does not yet utilize Babel for inter-language interoperability.

Decaf, a part of the Babel distribution, is intended as a reference implementation of a CCA framework using Babel. It does not explicitly support parallelism.

At the present time, there does not exist any CCA framework that meets the needs of the visualization community, where an exact definition of “needs” is still somewhat amorphous. Generally speaking, desireable characteristics of a suitable visualization framework would include support for multithreaded component execution (so you can have interactive rendering, for example), support for distributed and parallel components that are executed in an interactive fashion in response to changes to one or more component parameters (an isocontouring level, for example). In the meantime, Ccaffeine and/or SCIRun2 are probably the most appropriate for visualization interface development. Ccaffeine’s advantage is that it is more mature than the others. SCIRun2’s advantage is that it supports a threaded components and a threaded execution model. It may very well be the case that a new, visualization-specific framework may be required if one of the existing ones does not evolve to meet the needs of the visualization community.

4 Appropriateness of CCA as DiVA Basis

One of the primary objectives of the July workshop in Eugene was to evaluate whether or not CCA is "the right basis" for component interfaces in visualization tools. Some of the questions we considered are listed below.

4.1 *What does CCA Provide?*

- CCA is the only framework/component architecture that can provide desired performance. Both SIDL/Babel-based components and "CCA Classic" components provide the means for direct (i.e. shared-memory) component-component connections. In contrast, other forms of component interfaces (COM, Corba) require data marshaling between components. Data marshaling is a tractable solution for transaction processing systems, but is entirely inappropriate for high performance scientific applications where data may be quite large.
- Related to the previous point is the fact that the CCA specification and sample implementations provide for component use in both distributed- and shared-address space architectures. In other words, a given CCA component can be used without modification on a variety of architectures. The complement to the component is the framework, which is the environment used to execute components. Not all CCA frameworks provide the same type of functionality: some provide only for component use in a single address space, while others are oriented towards component deployment on distributed platforms. The degree to which a component may be used on different platforms depends upon the features of the framework, and not all frameworks provide the same set of features.
- CCA provides language portability/interoperability. The SIDL/Babel combination provides the means to have components based upon many languages all interoperate. SIDL/Babel components may be written in C, C++, F90/F77, Python and Java.

4.2 *What does CCA not provide that is out of the scope of CCA:*

- Data models for visualization. Data models/methods/accessors are needed for: scientific data types as well as visualization-centric results (geometry, images, normal maps, vectors, etc.). CCA provides a way to define component interfaces, not data models. As a community, we need to address the fundamental problem of disparate data models and types regardless of the specific component interface implementation that is used.
- A production quality framework that support remote, distributed and parallel components. The needs of the high performance scientific visualization community are diverse and demanding. In some cases, we will want a framework that supports rapid prototyping on single platforms for the purposes of development and debugging. In other cases, we will want a framework that supports development of large capacity, parallel visualization tools. In yet other cases, we may want a framework that provides for use of components that exist on multiple machines and at multiple sites. Some applications may be highly interactive, while others may be logically batch-style. The frameworks created by the working groups within the CCA are focused on meeting specific deployment environments, and none is all-encompassing or will provide all the features we may want. Creating such a

framework is well outside the scope of the CCA itself, and may become the responsibility of the visualization community.

4.3 What does CCA/CCTTSS[6] not yet provide, but should:

- Complete support for distributed computing. Includes basic data marshaling (serialization and deserialization) and data transport facilities.
- CCA frameworks should be Babel compliant. E.g., a component interface described using SIDL should be usable/runnable in all Babel-compliant frameworks.
- Provide mechanisms for remote component staging, startup, monitoring and management. There was not ample opportunity at the July workshop to more fully explore the capabilities of existing CCA frameworks, or to begin the process of providing more detailed requirements for remote component execution management.

4.4 What does CCA not do (and likely never do) that is a disadvantage:

- Use of language-neutral interface specification (SIDL) can preclude use of powerful language-specific features, e.g., C++ templates. (Some CCA people would say this statement should go into the "should do" category). In other words, a data model library (or component) that makes use of STL in C++ would not be directly usable in a SIDL/Babel framework: the OO capabilities provided by the underlying language would not survive traversal through the SIDL/Babel component interface. This particular issue is the primary reason for using the CCA "Classic" interface rather than the SIDL/Babel interface. Since most visualization components are written in C++, choosing the Classic interface appears to be the most attractive solution at this time.

4.5 Costs of Using CCA

- Complexity of specifying component interfaces. As the examples later in this document illustrate, the code necessary to add a CCA interface to a well-defined algorithm can appear to be complex. We would hope that as CCA evolves, that CASE tools would similarly evolve to relieve the developer of some of this burden.
- Immaturity of technology. The CCA specification is relatively new and evolving. As such, it presents a "moving target" for developers.
- Complexity of building, installing and using CCA tools. Some workshop participants shared their experiences of attempting to build and use CCA components within one of the frameworks. These anecdotes painted a picture of difficulty and frustration: there exist many software dependencies that are not always documents; some third-party software is not distributed in RPMs and there was no documentation for where such software was to be installed (if it wasn't installed in the correct location, some portion of the component build or run process would fail). To a large extent, these difficulties are typical of new and evolving software. At the workshop, we learned

[6] Center for Component Technology for Terascale Simulation Software. The subset of CCA funded by DOE to create software.

that the CCA project is in the process of assembling RPM distributions, which will hopefully alleviate most of these problems, at least for the Linux environment[7].

4.6 If not CCA, then what?

- Use a pure C++ templatized/class approach. To some extent, this approach is analogous to the CCA "Classic" strategy, sans the ability to use one of the CCA-compliant frameworks. In this approach, all modules/components would use compile time bindings, although interfaces via Python are possible.
- An alternative to a purely class library approach is to use SWIG to generate "interfaces" from header files. Such an approach does not address the fundamental issue of component interoperability.
- The "grid" approach – all the world is a socket. Lose language-specific features.
- Use alternative component architecture (COM, Corba, etc.) – all are known have significant disadvantages and shortcomings insofar as high-performance scientific visualization is concerned.

5 CCA and VTK

In considering the use of CCA as a supporting technology for a reusable visualization framework, we must consider the possibilities for incorporating existing visualization libraries and tool environments. Certainly we want to leverage the substantial investments that have already been made in developing visualization data structures, algorithms, interface techniques, and rendering capabilities. For example, the Visualization ToolKit (VTK) is a large library with substantial functionality, is widely used, and has been developed (in part) with DOE support. In the workshop, it was noted that VTK already follows the practice of wrapping the definitions of each class (implemented in C++) so as to provide API's that are callable from a variety of other languages, including Java, Tcl, and /or Python. Perhaps this same wrapping mechanism could be used to generate CCA-compliant SIDL interfaces, which would then make VTK capability available for use in CCA applications. The VTK-to-CCA mapping could be done at different levels of granularity. One alternative (not necessarily desirable) would be wrap the entire VTK library up as a single CCA component. At the other extreme, perhaps every VTK class would be wrapped as an individual component. At an intermediate level, existing VTK-based applications, constructed from a collection of VTK classes, could also be wrapped and made available as CCA components. Further discussion and some experimentation would be necessary to determine the level of granularity that is most useful, and this might be application specific.

At least one other person (Lori Freitag) has created a CCA component using VTK classes that was used as a viewer for some unstructured mesh data. The work was performed as part of the TSTT CCA working group, and demonstrated at SC2002. Email was sent to Lori on 7/9/2003 requesting more information, but she has not yet responded (as of 7/16/03).

[7] There is increasing support for RPM-based package distribution for non-Linux platforms.

6 Recommendations

- Regardless of whether we use CCA or some other component technology, or an entirely different software model (e.g. conventional libraries), we as a community must address the issue of "standard" data models. No component interface model provides a solution in this space. Without a "standard" for defining interfaces, there is no hope that our individual efforts will ever interoperate.
- As a community, we need to refine a list of requirements for visualization component interfaces. While requirements have been generally discussed in each of the workshops, they have not been fully codified. Such codification would form the basis for making decisions about reuse of any existing technology, such as CCA, or for clearly defining the need for new technology development.
- Further evaluation of CCA as the basis for visualization component interfaces by this particular group is warranted. While the CCA interface architecture appears to hold much promise, we are not entirely convinced that it is appropriate for our needs. Having a clear set of requirements would help to measure our needs versus the capabilities provided (and anticipated) by CCA.
- The "Mary Kay" problem. We all agree that we would like to be able to share our work and to leverage others' work. The mechanism for doing so is not clear. In other words, how do we distribute our technology and ensure it is widely adopted by others? We would like a centralized location where components can be placed, located/searched, and downloaded. At a higher level, we need to define mechanism for sharing our work and broadening community involvement. A simple SourceForge repository probably won't provide enough functionality for our purposes. An example vehicle for sharing work is the LLNL CASCV Alexandria project[8]. (a web based machine./human browsable component repository). This is an issue that is essential for success and still needs further investigation.
- Define the roadmap for framework and component standardization within the Visualization Community so that we realize interoperability. (In the text below, the term "class" is used in the broad English semantic sense, rather than any strict semantic of a programming language)
    - Define broad functional classes of visualization components.
    - Define classes and minimal requirements for data models and formats.
    - Define minimal interfaces for component classes (and which may be extended by implemented components).
    - Identify classes of framework users, and ensure representatives from these groups participate in the visualization component and framework definition and development effort at some level:
        - Pure research programmers (may not have an a priori goal of producing reusable code – just want to try new things; not necessarily interested in producing re-usable code; want maximum flexibility with minimum restrictions; potential (sparse) high value in "prototype quality" specialized components providing new utility)

[8] See https://www.llnl-casc.gov/alexandria. You will have to create a user login. Once the login is created, you won't see much. It is not clear what, exactly, Alexandria provides without having an example collection to view.

        - Research application developers (may have “customers” - domain scientists and analysts; generally want a functionally rich, re-usable, portable code base to develop, maintain and extend persistent applications and PSE’s; potential high value in stable components derived from existing code and new development.
        - Commercial software (product) and contract (e.g., Kitware) developers (want re-use, but may have long-established/entrenched framework or component system; may have low incentive for interoperation with “community framework”; little incentive to develop/port to new framework except from outside pressure (customer requests, funding pressure from sponsors, etc…); potential high value in components derived from commercial or open source packages (e.g., VTK).
    - Identify minimal set of components providing value (and incentive for adoption) to classes of users
    - Define interfaces for minimal set of visualization and framework infrastructure components. Isosurface, renderers (surface, volume), colormap editor, hedgehogs, streamlines, icon/glyph-based techniques, 2D plots, data interchange and transport components, etc …
- Define development plan for forward progress:
    - Reference implementations for a minimal set of visualization components.
    - Reference implementation for GUI-based application assembly.
    - Reference implementation for execution environment, including support for different types of display devices and event propogation/management as well as support for logging and replay and events.
    - Implementation/integration of the minimal set of components providing incentive for adoption.
- Learn more about CCA-based analysis tools for integration with visualization tools.
- Form a Visualization working group within CCA.

7 Appendix: Example Isosurface Component Interface Specification in SIDL

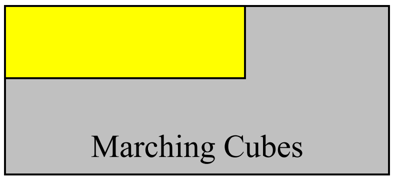
The following sections present example component interfaces for an isosurface component using SIDL. The three different examples illustrated the flexibility of SIDL in exposing interfaces to a visualization tool. The first example shows an isosurface component that presents a single port to the outside world. The single port encompasses input data, output results and parameters. The second example uses a different port for each of input data, output data and parameters. Both the first and second example interfaces are for components that process the entire dataset with one invocation. In contrast, the third example shows a set of ports and interfaces that would be used within the context of a streaming isosurface application.

7.1 Single-Port Component Example

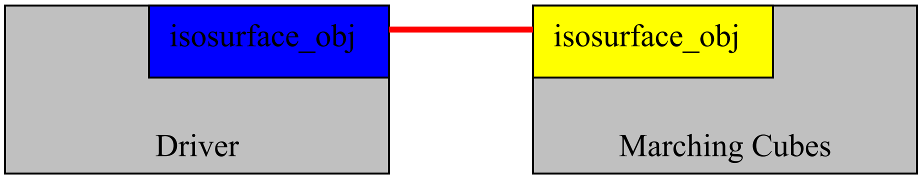
We create a new package "viz" that contains a single abstract interface "`isosurface_obj`". This interface, or "port", like all CCA interfaces extends "`cca.Port,`" which is an empty interface specification, useful for later dynamic casting of port handles obtained from the underlying framework. The "`isosurface_obj`" interface includes four method definitions: one for setting and getting the isosurface contouring level, one for setting the input data set, and one for actually generating the desired isosurface output for the given input data.

```
package viz {
      interface isosurface_obj extends cca.Port {

            void setContouringLevel( in float level );
            void getContouringLevel( out float level );

            void setInputData( in Data data );
            void generateSurface( out IsosurfaceData surface );
      }
}
```

Note that this port definition is but one of many potential sets of functionality that a given "component" might export; a single component can implement multiple interfaces/ports. For example, the above port might be implemented as one of many ports in a generic multi-purpose visualization component "Viz", or could be the **only** port exported from a special-purpose isosurface component, such as for a particular algorithm, say a "Marching Cubes" component. The user must select and instantiate the specific component by its name, and then further specify the port connections based on the port name. This arrangement is illustrated below:



The yellow "isosurface_obj" box corresponds to a "provides" port, i.e. a port that actually implements the given interface. Another component, say a "Driver" component, might support a matching "uses" port of the same type (depicted by a blue box below), which can be connected to the "provides" port to invoke the implemented methods:



Note that for this interface example, there are no additional ports required for the component implementation in "Marching Cubes" – everything occurs through this one interface, which defines the input data and parameters of the isosurfacing as well as directly producing the desired surface output. The input and output data are referenced as independent objects, not using additional explicit port connections among the components (see the complementary "isosurface_port" example below).

The usage of the above interface as seen from the connected "Driver" component would be as follows (some details simplified or omitted for brevity):

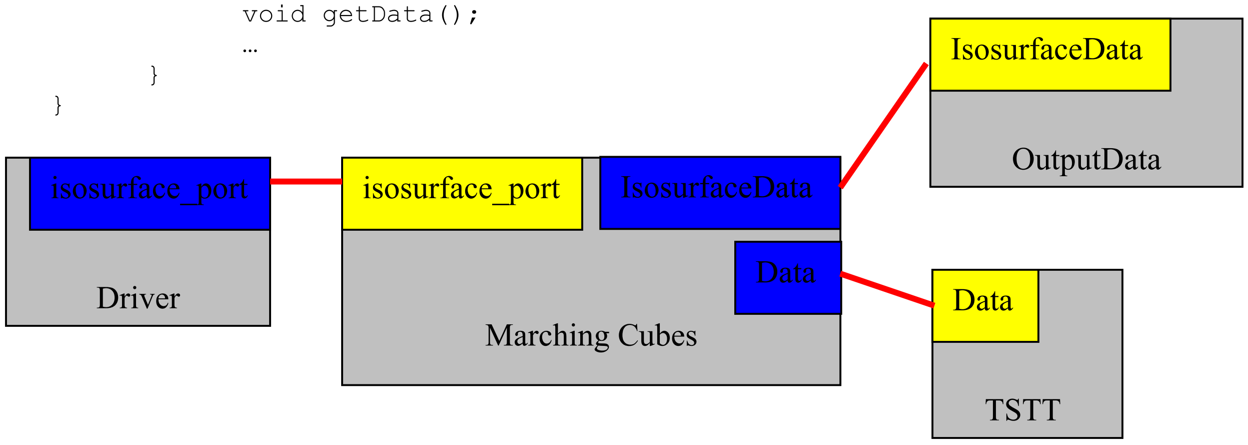
```
I = getPort( "isosurface_obj", … );

I->setContouringLevel( 1.0 );

I->setInputData( mydata );

I->generateSurface( mysurface );
```

The getPort() function returns a handle ("I") to the connected port implementation, so that the driver component can "use" its implementation. Next, the contouring level is set, as is the input data, and finally the isosurface is generated. This is all done via programmatic control using the given port handle. The "mydata" object must have been obtained in some way by the driver component before passing it to the setInputData() method, and the "mysurface" object is generated internally by the providing component and is returned in the invocation of the generateSurface() method.

7.2 Multi-Port Component

The next example differs in approach from the first example, and eliminates the use of dynamically instantiated objects for the input data and output isosurface in favor of explicit port connections to external component implementations. The "isosurface_port" interface therefore omits the setInputData() method, and instead implicitly assumes that the necessary input data is made available through a "tstt.Data" interface for which the encompassing component will register and a "uses" port. Similarly, the generateSurface() method no longer specifies an output argument for the resulting isosurface data; the method still triggers computation of the desired surface, but the actual triangle data is instead instantiated using the "`IsosurfaceData`" interface, which will also be accessed via a registered "uses" port. The component that implements the "`isosurface_port`" port will obtain its input data by invoking functions on the "`tstt.Data`" port, and will generate the desired surface data by invoking functions on the "`IsosurfaceData`" port (e.g. the `createTriangle()` method).

```
package viz {
      interface isosurface_port extends cca.Port {
            void setContouringLevel( in float level );
            void getContouringLevel( out float level );

            // uses a Data port

            void generateSurface( );

            // uses a IsosurfaceData port
      }
      interface IsosurfaceData extends cca.Port {
            void createTriangle( … );
            void gimmeTriangle( out Triangle tri );
            void gimmeAllTriangles( out Triangle tris[],
                        out int ntri );
      }
}
package tstt {
      interface Data extends cca.Port {
            void getData();
            …
      }
}
```



Here, the corresponding usage of the "`isosurface_port`" port omits the explicit invocation of any methods to set the input data or generate the isosurface output. These data and output instead are referenced internally by the port implementation, given additional `getPort()` calls to access the necessary "uses" ports. The selection of input data and the destination for the resulting isosurface is then done "compositionally" by connecting ports among the "Marching Cubes", "OutputData" and "TSTT" components in the framework at run-time, and does not require programmatic control from the driver component. The code implementing "`isosurface_port`" would then execute something like this (again simplified for brevity):

```
      I = getPort( isosurface_port, … );

      I->setContouringLevel( 1.0 );

      // Data is provided compositionally from a port…
      I->generateSurface();
```

```
        // Isosurface data is generated compositionally to a port…
```

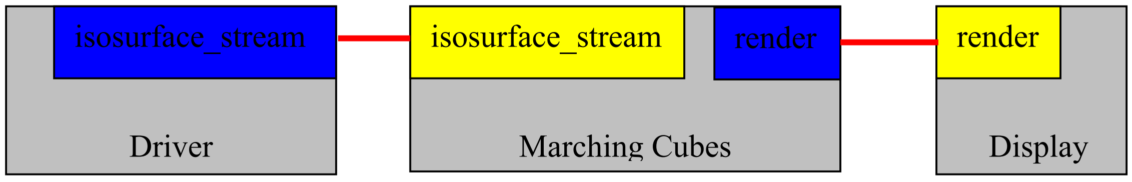
Internally, then the `generateSurface()` method would need to make its *own* port calls to get the data and create the output surface data:

```
        D = getPort( Data, … );

        mydata = D->getData();

        J = getPort( IsosurfaceData, … );

        …

        J->createTriangle( … );
```

7.3 Streaming Example

Another sample interface, "`isosurface_stream`", hypothesizes the nature of expected stream-like invocations, where the input data is incrementally indicated in the form of distinct sub-cells, with individual isosurfaces generated one-by-one for each cell by a single integrated `generateSurface()` method.

```
package viz {
        interface isosurface_stream extends cca.Port {

                void setContouringLevel( in float level );
                void getContouringLevel( out float level );

                void generateSurface( in Data data );
        }
        interface render extends cca.Port {
                void renderData( in IsosurfaceData surface );
        }
}
```



7.4 Creating Components and Connecting Ports in a Framework

To further explore the details of the underlying component implementations, excerpts of the associated `setServices()` methods, as provided by the driver component and a sample isosurface implementation component "`mcfoo`", are shown for the "`isosurface_port`" interface (the example which uses additional "uses" ports for obtaining the input data and creating the isosurface output data):

Inside the driver component:

```
void driver::setServices( in Services svc )
{
      svc->registerUsesPort(    "iso",    "isosurface_port",    …    );
}

Inside the component mcfoo:

void mcfoo::setServices( in Services svc )
{
      svc->addProvidesPort( "Isosurface", "isosurface_port", … );

      svc->registerUsesPort( "InputData", "Data", … );
      svc->registerUsesPort( "IsosurfaceOutput", IsosurfaceData", … );
}
```

The driver component registers a "uses" port for the "`isosurface_port`" interface (with the name "`iso`"), and the "`mcfoo`" component adds a "provides" port for the same interface (called "`Isosurface`"), as "`mcfoo`" provides an implementation of the port. The "`mcfoo`" component also registers "uses" ports for the "`Data`" input data and the "`IsosurfaceData`" output data.

Appropriate components could then be instantiated, and their ports connected, to compose the desired application program. This could be done using a framework GUI with "drag & drop" component creation and drawing graphical port connections, or could be done via a command line script for controlling the framework. This could also be done via programmatic control using a special "BuilderServices" library in the application program. A sample script is shown below:

In the framework script (for example):

```
create component driver
create component mcfoo
create component data
create component isoout

connect driver.isosurface_port to mcfoo.isosurface_port
connect mcfoo.Data to data.Data
connect mcfoo.IsosurfaceData to isoout.IsosurfaceData

driver.go()
```

Matching ports are connected between the "`driver`" and the "`mcfoo`" components, and between the "`mcfoo`" component and its required "`data`" and "`isoout`" components. The final `driver.go()` command invokes the special "`go`" port on the driver, actually executing the desired sequence of operations in the connected components.