A Component-Based Dataflow Framework for Simulation and Visualization

Alexandru Telea
Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
alext@win.tue.nl

Abstract

Reuse in the context of scientific simulation applications has mostly taken the form of procedural or object-oriented libraries. End users of such systems are however often non software experts needing very simple, possibly interactive ways to build applications from domain-specific components and to control their parameters. Integrating independently written (existing or new) code as components should ideally be a simple, possibly automated black-box process. We propose a dataflow-based component framework for simulation and visualization in which large existing C++ application libraries were easily integrated as interactively manipulable components by using a meta-layer object-oriented component abstraction. The path from user-level requirements to the framework design and implementation decisions is outlined.

1 Introduction

Designing open systems for scientific visualization and simulation (VisSym) has become crucial for disciplines such as computational fluid dynamics (CFD), medical sciences, or numerical analysis. As simulations' complexity has grown, the amount and variety of software elements involved in their design has increased considerably. In consequence, VisSym software architectures strive to evolve from problem-specific, monolithic applications to general-purpose frameworks offering interactive application monitoring and control and simple mechanisms to plug in various domain specific software.

Component-based frameworks seem ideal for the above task as they decouple component manufacturing from component usage. The latter is reduced to a simple operation of connecting matching components and having the framework enforce their correct cooperation via some standard protocol. Most of VisSym end-users are not programming experts, so frameworks should provide simple, interactive ways to load various components and connect them to produce the desired application without having to program. The components' parameters should be easily controllable interactively in order to steer the running simulations, and results should be available in real-time. Finally, the large amounts of existing scientific software libraries should be easily integrable in such frameworks as components, without having to modify their often unavailable source code.

However, due to several factors, designers have been so far reluctant to adopt component-based designs. Firstly, the above constraint combination is hard to satisfy, so virtually all existing VisSym frameworks mostly use white-box inheritance techniques, and are thus notoriously limited and difficult to use by researchers that need to focus on the modelled problem and not on the often intricate software integration techniques. Secondly, the extra indirections introduced by loosely coupled designs were considered too expensive for the interactive behavior of VisSym applications.

We addressed the above issues by designing VISSION [15], a component-based VisSym framework based on an object-oriented component model which extends the C++ class notion non-intrusively, in a black-box manner. Based on this component notion, we provide an easy way to build components from existing C++ class libraries, an interactive, programming-free manner to load, instantiate and connect them to create applications, and automatically built graphics user interfaces (GUIs) for for component parameter control and visual monitoring.

1.1 Requirements and User Roles

In VisSym, frameworks strive to satisfy the requirements of three user roles, or viewpoints, in the terminology of by Alencar et al[1]. End-users (EU), mostly non-programming experts, need virtual cameras, GUIs, and sometimes scripting languages. Application designers (AD) build the EU application by assembling predefined domain-specific components and work best if this assembly process is supported by a visual, interactive tool. This is usually done by connecting the components in a directed graph called a dataflow network. As the simulation runs, data flows from its source through components that process it up to the visualization components [10, 2]. The component developer (CD) forms the third user category: he builds application libraries by writing or often reusing existing code. The CD requires that existing code should be easily extensible and reusable as application components, and that the system loading the components should not constrain their design. Most frameworks address the CD's requirements by supporting some form of object-oriented components. By *OO component*, we shall further denote an OO software entity directly reusable in the context of a given framework. Often the same person cycles through all the roles (e.g. a researcher who develops his code as a CD, then builds a test experiment as an AD, and steers the final application as an EU). The cycle is repeated, as end-user insight triggers application re-design for the AD, which may induce component changes, a task for the CD. As the same person must quickly and frequently switch roles, frameworks should not only offer freedom for each individual role, but also an ideally automatic way to make the work of a role immediately available to the next role.

1.2 Limitations of Current Frameworks

Matching the above requirements to the most known VisSym frameworks, we identified the following limitations:

1.2.1 Extensibility and Reuse Problems

While OO libraries written in e.g. C++ or Java are easily extensible by subclassing (e.g. vtk [16] and Open Inventor [11] for C++ or Java3D for Java [7]), most frameworks require relinking or recompilation to use new component versions. They also often force components to inherit from a common root class (hierarchies having different roots are not accepted) or use only single inheritance (SI) in e.g. the C++ case [11, 10, 16]. White-box frameworks based on inheritance or other compile-time schemes that check the validity of components by checking their interface conformance are thus the most common.

1.2.2 Inflexible I/O Typing

By providing typed inputs and outputs for components (also called ports in the dataflow terminology), VisSym frameworks assist the AD with run-time type checking to forbid connections between incompatible types. However, most frameworks' run-time type systems have only a few basic types (integer, float, string, and arrays of these), and are not extensible with user defined types. By value and by reference data passing are also rarely both supported in the same framework. To provide run-time data conversion from one type to another, explicit conversion modules [2],[16] or complicated run-time schemes to register conversion functions [11] are used, instead of more elegant schemes present in some programming languages such as conversion operators or copy constructors, or of other meta-level object protocols that could negotiate data conversions more flexibly.

1.2.3 Intrusive code integration

Frameworks take a limited number of run-time decisions so are often simpler than full-fledged compilers or interpreters. The development language is often richer in concepts than the target framework, so CDs are forced to change their code to match the system's standard (e.g. give up multiple inheritance, pass by value, etc). Some frameworks require the components to be interfaced in a language different from the development one (e.g. tcl used by [16, 10] to interface C++ classes), thus the manual creation of wrapper classes or 'glue' code [1]. Sometimes the CD must add system-specific code to his components to add dataflow and GUI functionality [2, 5, 14], which is yet another form of white-box composition. Many researchers reported that otherwise well-designed OO libraries could not be integrated in VisSym frameworks due to the need to *intrusively* adapt their (sometimes unavailable) source code or need to build complex wrappers.

1.2.4 Different Run-Time and Compile-Time Languages

Most VisSym frameworks implement their components in native executable code for speed reasons, and offer the EUs interpreted languages to quickly set up experiments or issue various commands. Component devel-

opment languages are often different and usually more powerful than the run-time ones (e.g. tcl for [10], V for *avs*, cli for [2]), making some features of the latter unavailable in the former, and forcing the CD to learn two languages.

1.2.5 Manual GUI Construction

Even though GUIs could be automatically created from the components specification, the CD usually has to manually program (or interactively build [2]) the GUIs for the developed components. Moreover, the GUIs of most frameworks support editing only a few basic types (integers, strings, floats). There is no support for editing user-defined types (e.g. a 3D vector) or user defined GUI widgets (e.g. a 3-dimensional virtual spaceball).

2 Overview of VISSION

Analyzing the presented limitations, it seems that most of them are caused by a very limited, ad-hoc notion of components, usually the result of the evolution of simple code integration schemes used in early systems. However, requirements as dynamic, transparent integration of independently developed OO class libraries definitely ask for black-box integration, supported by mechanisms such as dynamic code loading and reflection. The fact that many VisSym frameworks do not promote a clear identity for components, separate from and semantically higher than the computational code they extend, forces white-box composition with all its known problems.

Our solution employs the C++ language in compiled form for the development of component libraries, *and* in interpreted form for the application developer and the end user. VISSION's kernel is a C++ interpreter [4] able to call C++ compiled code from dynamically loadable user-written libraries, execute on-the-fly synthesized C++ code, and offer a reflection API. This allows us to completely merge the OO and dataflow modelling concepts in a new abstraction called a *metaclass*, which extends a C++ class with dataflow semantics to create our framework's component.

Component libraries are loaded in VISSION where the AD interactively builds dataflow networks using a visual programming mouse-based GUI for component instantiation, cloning, destruction, port connections and disconnections (Fig. 3) performed on iconic representations of metaclass instances. The icons for the metaclasses and their instances, as well as the GUIs used for monitoring and modification of port values, are automatically constructed by VISSION from the metaclass specification (Fig. 4).

	C++	**Metaclass Language**
type:	class	metaclass
instance:	object	meta–object
interface:	public part	input/output ports
implementation:	private part	public part of own C++ class and update method

Figure 1: Comparison of C++ class and metaclass concepts

2.1 The Metaclass Concept

A metaclass is a programming construct written in a simple object-oriented declarative language. It adds a dataflow interface to a C++ class: a description of the inputs,outputs, and update method, i.e. the code to be executed by the dataflow engine when the inputs have changed. The metaclass inputs, outputs and update are forwarded to the public interface of the C++ class it extends: when an input is written into, a C++ class method is called to perform the write or a public member is written; when an output is read from, a method is called and the return value is used or a public member is read. Inputs and outputs are typed by the C++ types of their underlying methods or members. Metaclasses are object-oriented since they can inherit inputs, outputs and the update methods from other metaclasses (single, multiple and virtual inheritance are supported) similarly to their underlying C++ classes, so a metaclass hierarchy is isomorphic to the C++ hierarchy it extends. We added however some OO features present in C++ to the metaclass language, e.g. the selective hiding or renaming of inherited features, similar to the approach described by Meyer in [9]. This proved very useful when managing complex metaclass hierarchies. Metaclasses of non abstract C++ classes with public constructors can be instantiated to create *meta-objects* which are connected to form the dataflow network. A metaclass is ultimately an object-oriented type for the dataflow mechanism, implemented in terms of the C++ class type (Fig. 1). Fig. 2 shows two C++ classes of a larger hierarchy and their metaclasses: the IVSoLight metaclass has three inputs for a light's color, intensity, and on/off value, modelled by the corresponding C++ class's methods with similar names, and of types IVSbColor (a RGB color triplet), float, and respectively boolean. Meta-

Metaclasses:	*C++ classes:*

module **IVSoLight**
{ input:
WRPort "intensity" (setIntensity,getIntensity)
editor: Slider
WRport "color" (setColor,getColor)
WRport "light on" (on)
}
	class **IVSoLight**
	{ public:
	BOOL on;
	void setIntensity(float);
	float getIntensity();
	void setColor(IVSbColor&);
	IVSbColor getColor();
	};

module **IVSoDirectionalLight**: **IVSoLight**
{ input:
WRPort "direction" (setDirection,getDirection)
}
	class **IVSoDirectionalLight:** public **IVSoLight**
	{ public:
	void setDirection(IVSbVec3f&);
	IVSbVec3f getDirection();
	};

Figure 2: Example of C++ class hierarchy and corresponding metaclass hierarchy

class IVSoDirectionalLight extends IVSoLight with an input for the light's direction, of type IVSbVec3f (a 3-space vector). Besides the port to C++ member mapping, metaclasses specify other non-functional requirements such as the labels to be used in their automatically constructed GUIs (Fig. 4), optional widget preferences (for the intensity, a slider is preferred to a typein in the above example), and documentation data that can be accessed on-line. Appropriate widgets are automatically created based on the ports' C++ types (3 float typeins for IVSbColor and IVSbVec3f, a toggle for the boolean, and a slider, as the user option specified, for the float). The requirements and limitations listed in Sect. 1.2 are addressed as follows:

2.1.1 Extensibility and Reuse

The CD develops his application C++ classes with no restriction imposed by VISSION (no common root class required, multiple and virtual inheritance can be used, etc) and organizes them in application libraries. Extra information is next added by writing the metaclass descriptions for the C++ classes in a straightforward fashion (the metaclass language has only a few keywords and very simple declarative constructs). Metaclasses are organized in libraries using the C++ application libraries as implementation only via their public interfaces, introducing a first level of black-box reuse. Metaclass libraries can include other metaclass libraries and metaclasses from one library can inherit from metaclasses in other libraries, similarly to Java packages. When VISSION dynamically loads a metaclass library, metaclasses from directly and indirectly included libraries are transparently loaded, together with their corresponding C++ classes. This is a second level of black-box reuse between the framework and the metaclass libraries.

2.1.2 Flexible I/O Typing

Data flow between ports is based on the full OO typing of C++: it can be passed by value, by reference, and can be of any type (fundamental or class). For class types, constructors and destructors are automatically called when data flows from an output to an input. Port connection type checking obey all C++ typing rules: a port of C++ type A can be connected to a port of type B if A conforms to B by trivial conversion, subclass to baseclass conversion, user-defined constructor and conversion operator [12]. This generalizes the dataflow typing policies of other systems: The Oorange system [10], based on *Objective C*, offers by-reference but no by-value data passing. AVS/Express [2] limits the run-time data types to the ones provided by its OO *V language* which lacks constructors, destructors, and multiple inheritance. Compiled toolkits as Open Inventor [11] and vtk [16] are only statically extendable, as all types have to be known at compile time.

2.1.3 Non-intrusive code integration

All information needed to promote a C++ class to a metaclass directly usable by VISSION resides in the metaclass. The metaclass-C++ class pair is roughly similar to the handle-body idiom [3], the Adapter pattern [6] or the customizable adapters presented by Kucuk et al [8], but is much easier to do than e.g. manual Adapter coding as the parallel hierarchies are managed automatically by the system, not the user. Separating the dataflow information in the metaclass allows adding dataflow semantics to existing class libraries, even when they are not available in source form. This separation between the pure, framework-independent code and the 'adaptation' layer including framework specific elements such as non-functional requirements is advocated by many [6],[13], as it allows code to be

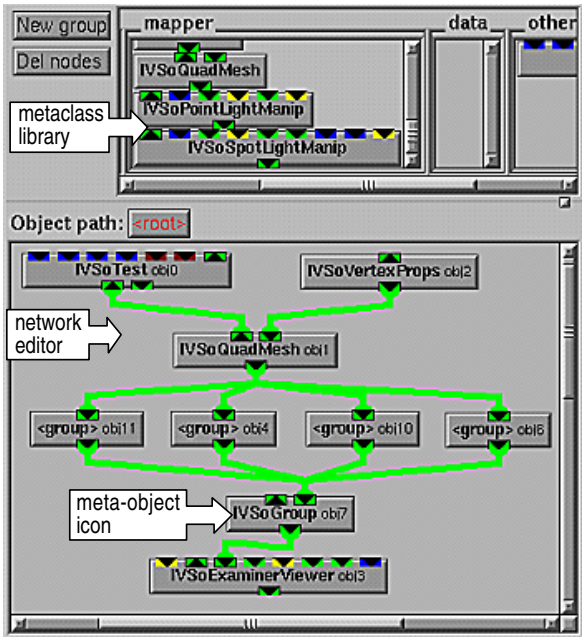

Figure 3: The dataflow network is build in the visual editor by instantiating the loaded metaclasses

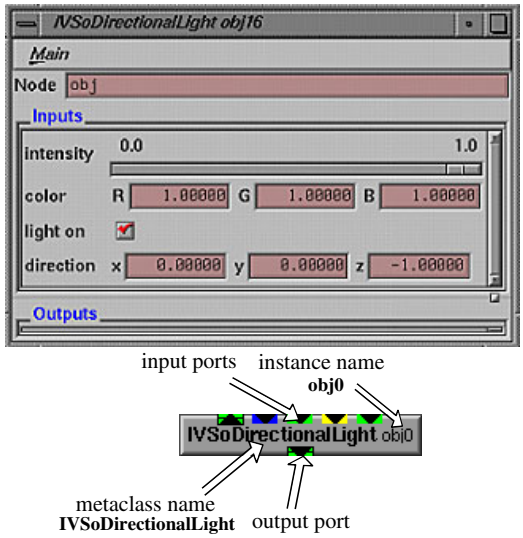

Figure 4: Metaclass icon with ports (left) and its automatically constructed GUI (right)

easily reused in various contexts as its design is not influenced by the target environment.

Moreover, since the metaclass specification code is simply parsed by VISSION, it is very easy and fast to change it to e.g. adapt different C++ classes (e.g. having different interfaces) or adapt the same classes differently (e.g. define ports or the update to call back on other C++ methods). In this sense, our adaptation method is different from other solutions. The metaclass code *might* be seen as partially white-box (since it has explicit dependencies on the C++ classes' interfaces), but it has practically none of the white-box drawbacks, since it comes in a very easy to edit/change form and not as binaries.

2.1.4 Single language solution

C++ is VISSION's single language: application libraries are written in C++, the metaclass ports are typed also in C++, C++ commands can be typed in a console to be dynamically interpreted (obviating the need for a scripting language). We implemented also a generic persistence mechanism which saves all meta-object input port values and connections as C++ source code commands and restores the state by simply interpreting the saved code. The fact that our metaclass state is completely intrinsic supports once more the idea that components should be designed independently on the context in which they are used.

2.1.5 Automatic GUI Construction

VISSION automatically builds *GUI interaction panels* (shortly interactors) to examine and modify the values of any metaclass's ports. Interactors create the system's third object, isomorphic with the C++ class and the metaclass hierarchies: an interactor inherits the widgets from the interactors of its metaclass' bases. The three hierarchies correspond to the three user classes: EUs are concerned with the interactors, ADs with the metaclass interfaces, and CDs with the C++ classes. The interactor widgets reflect directly the C++ types of the edited ports. For example, a *float* port can be edited by a slider, a *char** port by a text type-in (Fig. 4), a three-dimensional *VECTOR* port by a 3D widget allowing direct manipulation of a 3D vector icon (Fig. 5 c), a *boolean* by a toggle button, a complex Material class encoding over 15 attributes by a Material widget (Fig. 5 c), and so on. VISSION's widget set for the fundamental C++ types is extendable by the AD with widgets for new, application-specific C++ types. We used this mechanism to provide GUI widgets for C++ types used by various libraries we included in VISSION, such as 3D vectors, colors, rotation matrices, and light values. Type-specific widgets are loaded and unloaded with the metaclass libraries defining their types, so they can also be seen as black-box components from the framework's point of view.

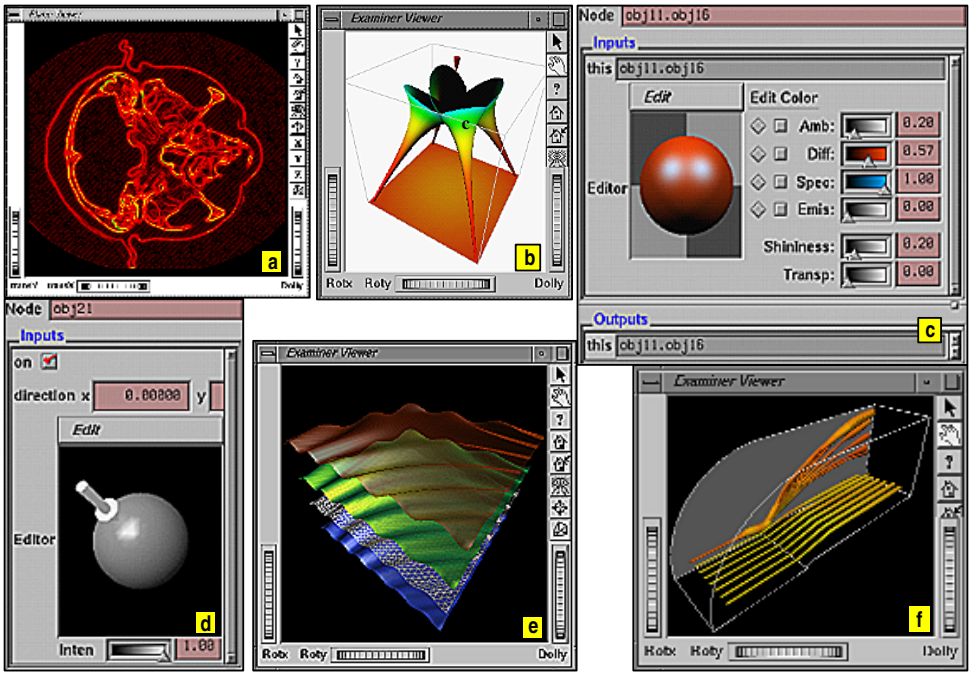

Figure 5: Automatically constructed GUIs and visualization examples in the VISSION framework

The association of a widget to a port type is done automatically at run-time by VISSION, which picks from the available widgets the one whose C++ type *best matches* the type of the port to edit. The best match rules are based on a distance metric in type space between the type to edit and the type editable by a widget, roughly similar to C++'s type conformance rules. The GUI building process can be however customized by supplying new GUI widgets or by specifying preferred widgets (a float type-in can be preferred to a slider for a float port, for example) in the metaclass specification. The loose coupling between OO widgets and OO ports via the run-time best match rule, the user-specifiable hints and the interactive widget switching correspond to the three user layers (CD,AD,EU). They form a meta-object protocol between VISSION and the component to negotiate the GUI creation.

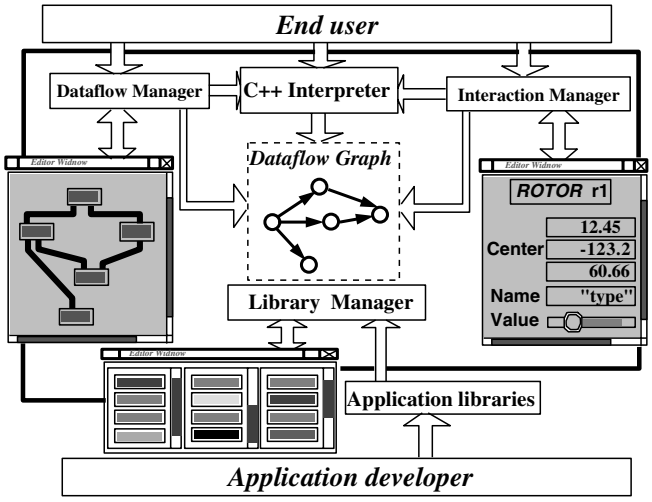

Figure 6: Architecture of the VISSION component framework

3 Architecture

VISSION consists of three main parts: the object manager, the dataflow manager, and the interaction manager (Fig. 2) that use two subsystems: the C++ interpreter and the library manager. All subsystems communicate by sharing the simulation dataflow graph. The key element is the *C++ interpreter* [4]. Operations throughout VISSION, such as connection or disconnection of ports, data transfer between ports, update methods' call, GUI editing of ports, are all implemented as small C++ fragments dynamically sent to the interpreter. The automatic GUI construction and the port connection type checking use the interpreter's reflection API. The interpreter cooperates with the *library manager* to dynamically load and unload

metaclass libraries and their underlying compiled C++ classes, with the *object manager* to parse metaclass declarations and instantiate metaclasses, and with the *interaction manager* to build and control the GUIs. Almost all code is executed from the compiled C++ classes, leaving only a tiny amount of C++ code to be interpreted. Performance loss as compared to a 100% compiled system was estimated to be below 2%, even for complex networks requiring an intensive communication with the interpreter, so the extra indirection level due to the loose coupling was definitely negligible. Loading and unloading metaclass libraries was however much slower: this implies, for medium-sized libraries having hundreds of metaclasses with 10-20 attributes each, the introduction in VISSION's type and function tables of thousands of new names and other information. We noticed however that loading a few large libraries is times faster than managing a fine-grained network of many small libraries referring each other, a fact similar to the performance issues of compilers vs header files.

Figure 7 presents the relationship between the port read and write operations, the interpreted and compiled C++ code, and the high-level tasks (data transfer, GUI-based inspection and modification of ports). A write operation in a GUI widget triggers a write to a port of the GUI's metaclass (step W1), which sends a C++ fragment of the form "obj1.set()" to the interpreter (step W2), the argument of *set*() being the data written to the GUI and the target of the *set*() message being the metaclass's C++ object *obj*1. The interpreter executes the C++ fragment calling the *set*() method from the compiled application library (step W3). A similar process occurs when reading a C++ value to refresh the GUI (steps R1,R2,R3). To transfer data between two ports (step T1), a port read (steps R1,R2,R3) followed by a port write (steps W1,W2,W3) is executed. The *dataflow manager* uses the above mechanism to perform automatically a network traversal calling node updates whenever an input changes.

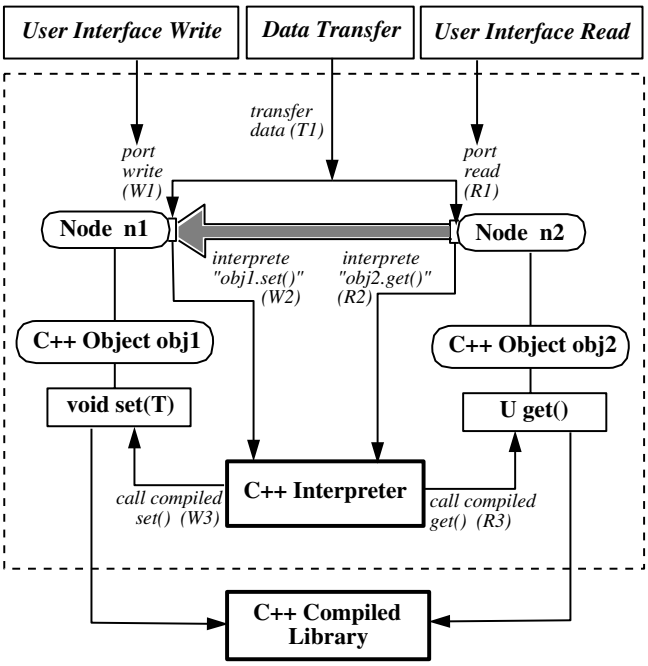

Figure 7: User interface read/write operations, data transfers between ports and the underlying C++ mechanism

4 Conclusion

VISSION is a general-purpose visualization and simulation component framework built on a black-box component foundation. It provides simple ways to specification, monitoring, steering of simulations, and component integration, by merging the powerful, yet so far independently used OO and dataflow modelling concepts in the *metaclass* component notion.

Metaclasses extend C++ classes with dataflow semantics, GUI and documentation information in a black-box fashion, making them easily reusable in other contexts and also making it easy to plug-in existing C++ libraries as components. Metaclass libraries (or their underlying implementation C++ libraries) can be seen as application frameworks modelling some particular domain, while VISSION is a component framework, since it a) coordinates the interaction of *independently* designed components according to its domain-specific (VisSym) rules implemented using the dataflow model, and b) it communicates with these components *only* via the metaclass interface.

We have provided a mechanism for automatic GUI construction from type-specific, user-defined widgets, based on a meta-object protocol merging OO typing with higher level information such as user preferences. VISSION's implementation key issue was the choice for a single (OO) language solution based on C++. VISSION's C++ interpreter/compiler design shows that one can combine speed and design freedom of compiled C++ (multiple inheritance, pass by value for user types, etc) with the advantages of interpreted environments (run-time flexibility, ease of use, reflection APIs) like Java/JavaBeans also in the context of the C++. Had we chosen a JavaBeans-based implementation, it would have been tedious and very difficult for our tar-

get users to integrate several large existing C++ class libraries which extensively use pass by value and multiple inheritance.

Several applications illustrate the advantages a fully object-oriented computational steering architecture provides. Component designers plugged-in existing well-known libraries such as vtk [16] for scientific visualization and Open Inventor [11] for rendering (420 classes,over 200000 C++ lines), or other libraries for radiosity (18 classes, over 6000 C++ lines) and finite element analysis (25 classes, over 7500 C++ lines) in VISSION in a short time (approximately 2 months, 5 days, respectively 10 days). Integration required writing a comparable number of metaclasses of an average length of 6 lines, and absolutely no change of the libraries (of which, Inventor was only available as binaries).

Application designers and non-programmer end users could effectively use VISISON in a matter of minutes (Fig. 5 a,b,e,f show snapshots from scalar, vector, tensor, and medical visualizations).

The strong separation of pure application code (written by the component designer) from infrastructure as dataflow mechanisms, GUIs, persistence schemes (provided by VISSION) makes the code to be written by the former clear and also very concise. This is noteworthy since most large application toolkits [11, 16] dedicate up to 50% of their code to implement backbone services as the ones mentioned before. Library designers may save 50% time if infrastructural services are automatically provided. Moreover, if a single backbone is applicable, it is coded just once (in VISSION) and not replicated in endless flavors among the open set of application libraries.

References

[1] P. Alencar, D. Cowan, C. Lucena, L. Nova, *A Model for Gluing Components*, Proc. of WCOP' 98, Turku Centre for Computer Science, 1998.

[2] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, *The Application Visualization System: A Computational Environment for Scientific Visualization.*, IEEE Computer Graphics and Applications, July 1989, 30–42.

[3] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992

[4] M. Goto, *The CINT C/C++ Interpreter and Dictionary Generator*, The ROOT System URL http://root.cern.ch/root/Cint.html

[5] J. J. van Wijk and R. van Liere, *An environment for computational steering*, in G. M. Nielson, H. Mueller and H. Hagen, eds, *Scientific Visualization: Overviews, Methodologies and Techniques*, Computer Society Press, 1997

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[7] *The Java 3D Application Programming Interface*, http://java.sun.com/products/java-media/3D/

[8] B. Kucuk, M. N. Alpdemir, R. N. Zobel, *Customizable Adapters for Black-Box Components*, in *Proceedings of WCOP '98, Turku Centre for Computer Science*

[9] B. Meyer, *Object-oriented software construction*, Prentice Hall, 1997

[10] C. Gunn, A. Ortmann, U. Pinkall, K. Polthier, U. Schwarz, *Oorange: A Virtual Laboratory for Experimental Mathematics*, Sonderforschungsbereich 288, Technical University Berlin. URL http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html

[11] J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1993.

[12] B. Stroustrup, *The C++ Programming Manual*, Addison-Wesley,1993.

[13] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[14] D. Jablonowski, J. D. Bruner, B. Bliss, and R. B. Haber, *VASE: The visualization and application steering environment*, in *Proceedings of Supercomputing '93*, pages 560-569, 1993

[15] A. C. Telea, J. J. van Wijk *Design of an Object-Oriented Computational Steering System*, to be presented at the IEEE-Eurographics Workshop on Scientific Visualization and Simulation VisSym'99, Vienna, Austria

[16] W. Schroeder, K. Martin, B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1995