Combining Object Orientation and Dataflow Modelling in the VISSION Simulation System

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

Abstract

Scientific visualization and simulation frameworks mostly use data/event flow mechanisms for simulation specification, control, and interactivity. Even though object orientation powerfully and elegantly models many application domains, integration of OO libraries in such systems remains difficult. The elegance and simplicity of OO design gets lost in the integration phase, as most systems do not support combination of OO and dataflow concepts. We propose a general-purpose OO visualization and simulation system which addresses simulation design, control and interactivity by merging OO and dataflow modelling in a single abstraction. The system's advantages over similar ones are illustrated by a comprehensive set of examples.

1: Introduction

Computer simulation and visualization (VisSym) have become well established tools for disciplines as computational fluid dynamics (CFD), medical sciences, and realistic image synthesis. Simulation software has evolved from problem-specific, monolithic applications to general-purpose frameworks offering application interactive monitoring, control, and design, using domain-specific components which are programmed independently on the framework itself. Traditional VisSym framework design is usually centered on a dataflow mechanism: applications are modelled as networks of computational modules exchanging data to perform the required task. Interactive modelling is done by building the network in terms of visual components' representations selected from various domain-specific libraries. Interactive monitoring and control is done by editing the modules' input parameters and visually monitoring their outputs. To be effective, such systems must offer also easy ways to design, reuse, and extend application components and a preferably automatic way to import them into the framework.

Object-oriented (OO) design is the favourite technique for designing, reusing and extending component libraries. Current dataflow-based frameworks are however not built on object-oriented foundations, at least not up to the point where integration of OO component libraries would be a simple, automatic task. Users must often manually redesign the components to the framework's policy, a difficult and time-consuming task. Integrating the many existing, independently developed OO component libraries is even a more formidable task, at which existing frameworks usually fail.

We believe that the above problem is caused by the different views of the framework and the libraries on the (OO) component notion. We addressed this by designing VISSION, a general

purpose environment for VIsualization and Steering of SImulations with Objectual Networks. VISSION merges the data/event flow modelling familiar to visualization scientists with the OO modelling familiar to application library developers in a single component abstraction. Based on this abstraction which extends a C++ class, VISSION provides a better user interface for simulation monitoring and steering than similar systems [5, 7, 15], and a visual programming tool for dataflow network construction directly from C++ classes. Integration of independently developed C++ libraries becomes a simple process.

In section 2 overview the requirements for VisSym frameworks and outline the limitations of existing systems. Section 3 a gives a top-down outline of VISSION, showing how it meets the previously described concepts. Section 4 outlines VISISON's architecture and implementation. Section 5 presents various applications. Section 6 concludes the paper with our current research directions.

2: Simulation and Visualization Frameworks: Requirements and Limitations

2.1: Requirements and User Roles

Modern VisSym tools [11, 17, 15] strive to satisfy the requirements of three user classes. End-users (EU) investigate and steer simulations by virtual cameras, direct manipulation, graphics user interfaces (GUIs), or some scripting language. Application designers (AD) build simulations for the EU by assembling a set of predefined components. They require a preferably visual tool for selecting components from various application domains to interactively build the EU application. 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 modules processing it up to the visualization modules [7, 5, 12]. The component developers (CD) forms the third user category: he builds application libraries by writing or reusing existing code. The CD needs to easily extend and reuse existing code as application components, and that the system loading the components should constrain their design. Most frameworks address the CD's requirements by supporting some form of object-oriented components. We shall use the term *OO component* to denote an OO software entity directly reusable in a give framework's context. Often the same person cycles through all roles (e.g. a researcher who develops his code as a CD, then builds a test experiment as an AD, and monitors 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 role, but also an ideally automatic way to make the work of a role immediately available to the next role.

2.2: Limitations

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

2.2.1: Extensibility and Reuse Problems

While OO libraries written in e.g. C++ or Java are easily extensible by e.g. subclassing (e.g. vtk [6] and Open Inventor [3] for C++ or Java3D for Java [14]), most frameworks require

relinking or recompilation to use new libary 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 [3, 7, 6].

2.2.2: Inflexible I/O Typing

By providing component typed inputs and outputs (also called ports in the dataflow terminology), frameworks assist the AD with run-time type checking to forbid connections between incompatible types. However, most systems' run-time typing has only a few basic types (integer, float, string, and arrays of these), and is 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 [5, 6] or complicated run-time schemes to register conversion functions [3] are used, instead of more elegant schemes present in some programming languages such as conversion operators or copy constructors.

2.2.3: Intrusive code integration

As the development language is often richer in concepts than the target framework, the CD must change his code to match the system's standard (e.g. give up multiple inheritance, pass by value, etc). Other frameworks require the components to be interfaced in a language different from the development one (e.g. the tcl language used by [6, 7, 16] to interface C++ components), thus manual creation of wrapper classes. Sometimes the CD must add system-specific code to his components in order to add dataflow and GUI functionality [5, 11, 15]. Many researchers reported that otherwise well-designed OO libraries could not be integrated in VisSym frameworks due to the need to *intrusively* adapt their source code (sometimes libraries were not available in source form).

2.2.4: Different Run-Time and Compile-Time Languages

Most VisSym frameworks implement their components in directly executable (compiled) code for speed, and offer the EU interpreted languages to quickly set up experiments or issue commands. The component development language is different and usually more powerful than the run-time one for most frameworks (e.g. tcl for [7], V for *avs*, cli for [5]), making some features of the latter unavailable in the former, and forcing users to learn two languages.

2.2.5: Manual GUI Construction

Even though GUIs could be automatically created from the components specification, the CD usually must manually program (or interactively build [5]) them. Moreover, most frameworks' GUIs can edit 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).

3: Overview of the VISSION System

From the presented limitations, we infer that the inability of current VisSym frameworks to cope with their requirements is caused by their inability to *directly accept* the OO component development language.

Metaclasses:	*C++ classes:*

node **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();
	};

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

Figure 1. Example of C++ class hierarchy and corresponding metaclass hierarchy

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 able to call C++ compiled code from dynamically loadable user-written libraries, execute on-the-fly synthesized C++ code, and offer a reflection API. Next, we 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 GUI based on simple mouse operations for component instantiation, cloning, destruction, port connections and disconnections (Fig. 2) performed on a visual representation of the metaclass instance. The visual 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. 3).

3.1: 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 delegated to its C++ class's public interface: 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 as they can inherit inputs, outputs and 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 not present in C++ to the metaclasses, e.g. the possibility to selectively hide or rename inherited features, similar to the approach described by Meyer in [9]. This proved useful when managing complex metaclass hierarchies. Metaclasses having instantiable C++ classes can be instantiated to create *nodes* which are connected in 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 shows an example of two C++ classes of a larger hierarchy and their corresponding metaclasses: the IVSoLight metaclass has three inputs for a light's colour, intensity, and on/off value, modelled by its C++ class's methods with similar names, and of

types IVSbColor (a RGB colour triplet), float, and respectively boolean. Metaclass IVSoDirectionalLight extends IVSoLight with an input for the light's direction, of type IVSbVec3f (a 3-space vector). Besides the C++ class member to metaclass port mapping, metaclasses specify other informations such as the labels for the metaclass's automatically constructed GUI (Fig. 3) and optional widget preferences (for the intensity, a slider is preferred to a typein in the above example). 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). We address the requirements and limitations from Section 2 as follows:

3.1.1: Extensibility and Reuse

The CD develops application C++ classes with no restriction imposed by VISSION (no common root class required, multiple and virtual inheritance supported, etc) and organizes them in application libraries. Dataflow semantics 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 grouped in libraries using the C++ application libraries as implementation. 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 loaded, with their corresponding C++ classes.

3.1.2: Flexible I/O Typing

Data flow between ports is based on the full OO typing offered by 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 [1]. This generalizes the dataflow typing used by other systems: The Oorange system [7], based on *Objective C*, offers by-reference but no by-value data passing. AVS/Express [5] limits the data types to the ones provided by its own OO *V language* which lacks concepts as constructors, destructors and multiple inheritance. Compiled toolkits as Open Inventor [3] and vtk [6] are only statically extendable (all types must be known at compile time), so are unsuitable for a dynamic, interactive environment.

3.1.3: Non-intrusive code integration

Metaclasses contain all information needed to render a C++ class directly usable by VISSION. The metaclass-C++ class pair resembles the handle-body idiom [2] or the Adapter pattern [13], but is simpler than e.g. manual Adapter coding as the management of the parallel hierarchies is done 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 as source. The CD can focus on his application code without caring that it will be later integrated in a dynamic system, managed by a dataflow mechanism, and interactively steered by a GUI.

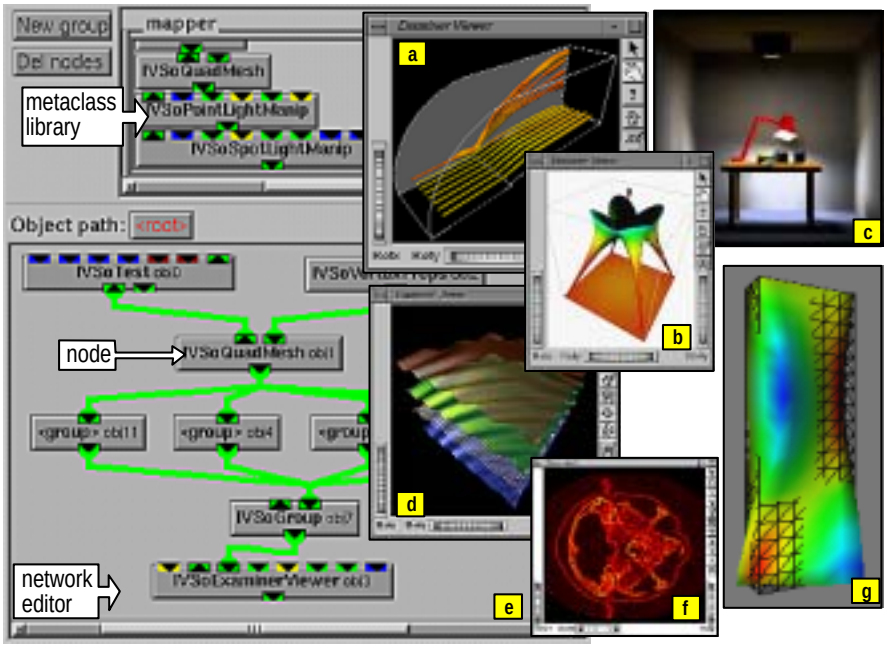

Figure 2. VISSION **network editor (a) and visualization examples (b-f)**

Figure 3. Left: Metaclass icon. The ports' graphical signs encode C++ type, by value/by reference and other attributes. Right: metaclass GUIs

3.1.4: Single language solution

C++ is VISSION's single language: application libraries are written in C++, the metaclass ports are typed also in C++. The EU can type C++ code in a VISSION console to be dynamically executed by the C++ interpreter (thus obviating the need of an extra scripting language). We implemented a generic persistence mechanism which saves the values of all node inputs and the network topology as C++ source code. The simulation state is fully described by its inputs and connections since the internal values of the C++ objects are not visible to the dataflow model, so it is enough to load and interprete the saved C++ source to restore the simulation. Finally, animations based on arbitrarily complex control sequences are easily produced by writing C or C++ script-like files which is executed by VISSION using metaclasses from the application libraries. Users don't have to learn special-purpose animation scripting languages, as it is the case in most animation systems.

3.1.5: Automatic GUI Construction

VISSION automatically builds *GUI interaction panels* or shortly interactors to examine and modify any metaclass's port values. Interactors create the third object hierarchy in the system, isomorphic with the C++ class and the metaclass hierarchies: an interactor inherits the widgets of the interactors of its metaclass' bases. The three hierarchies match 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 directly reflect the C++ types of their ports. For example, a *float* port can be edited by a slider, a *char** port by a text type-in, a three-dimensional *VECTOR* port by a 3D widget directly manipulating a vector icon in 3D, a *boolean* by a toggle button, and so on (Fig. 3). VISSION's widget set for the fundamental C++ types is extendable by the AD with widgets for application-specific C++ types. In this way, we provided GUI widgets for C++ types used by a visualization library we included in VISSION, such as 3D vectors, colours, rotation matrices, and light values. VISSION associates widgets to port types automatically at run-time by picking out of 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, similar to C++'s type conformance rules. Users can customize an automatically built GUI by supplying new GUI widgets or by specifying preferred widgets for certain ports (prefer a float type-in to a slider for a float port, for example). Users can also change a port's widget at run-time by a mouse-posted menu with all widgets capable of editing that port. 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 again to the three user layers (CD,AD,EU) and give VISSION a very flexible but simple way of automating the GUI creation process. Finally, VISSION offers a GUI for inspection and modification of the C++ objects used by the metaclasses, similar to the visual object browsers offered by several debugger environments. This GUI shows all public class members (in a format suitable to their type) and methods, and allows the user to modify the non *const* members or call the methods with interactively supplied parameters. This interface is useful for the CDs who want to directly monitor the C++ classes bypassing the metaclass abstraction level.

4: Architecture

VISSION consists of three main parts: the object manager, the dataflow manager, and the interaction manager (Fig. 4). Their operation is based on two secondary components: the C++ interpreter and the library manager. All components communicate by sharing the dataflow graph that describes the simulation. The key element is the *C++ interpreter*. Operations throughout VISSION, such as connection or disconnection of ports, data transfer between ports, node updates, GUI port editing, and command-line C++ interpretation are all implemented as small C++ fragments dynamically sent to the interpreter. The 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 the metaclass declarations and create and destroy nodes, and with the *interaction manager* to manage the GUIs. Almost all code is executed from compiled C++ classes, leaving only a tiny amount of interpreted C++. Performance loss as compared to a 100% compiled system was estimated to be below 2%, even for complex networks intensively communicating with the interpreter.

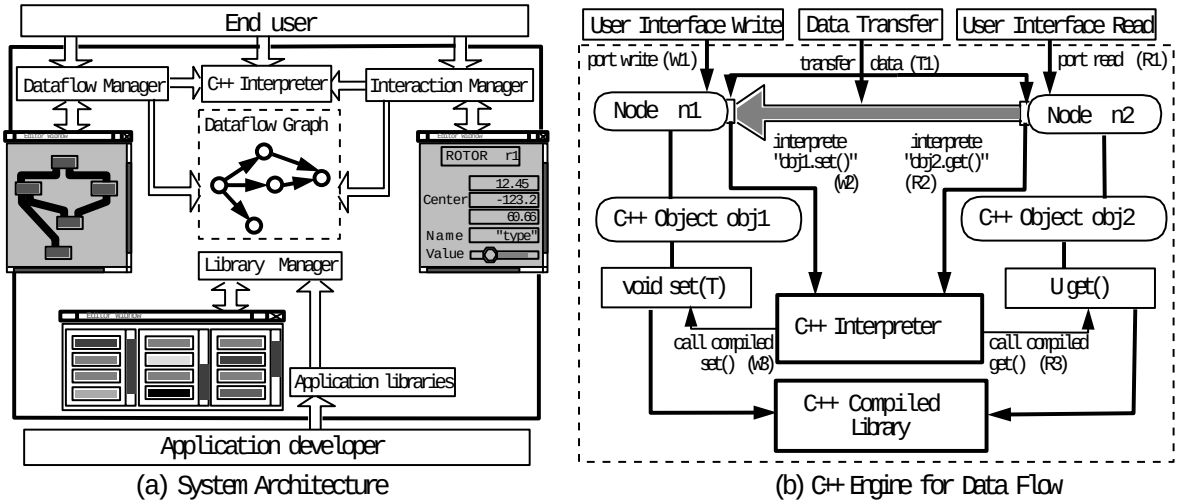

Figure 4. Architecture of the Vission system

Figure 4 b 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 traverse networks automatically, calling node updates whenever an input changes. A less common feature of VISSION is the support of networks containing *loops*, a very natural way to describe iterative processes.

As networks often contain too many nodes to be handled in the editor, we introduced the concept of *node groups*, which can contain arbitrarily nested subnetworks. Node groups can be interactively constructed by adding nodes and ports to them, thus generalizing Oorange's nodes, AVS/Express's macros, or Inventor's node kits. Moreover, the EU can decide to promote the structure of a node group to a 'type level', i.e. automatically generate a metaclass for it. This (the 'group' metaclass) differs from the metaclasses presented so far (the 'C++-based' metaclasses) as its implementation is a whole metaclass subnetwork and not a C++ class. Group metaclasses can be stored in a metaclass library as usually and used later exactly as C++-based metaclasses. They are very convenient for users who notice recurrent subnetwork patterns in their applications and would like to manipulate them as named entities in a single move, instead of building them from scratch every time.

5: Applications

The following presents some of the applications we have built with the VISSION framework. We have chosen the Visualization Toolkit (shortly vtk) [6], a large, comprehensive C++ toolkit for scientific data representation, processing, and manipulation, and integrated it in VISSION. in the VISSION system. For a rendering back-end we integrated Open Inventor [3],

a powerful 3D rendering and interaction toolkit. EUs can pick any vtk or Inventor class (of the total of approximately 250, respectively 70) in the visual browser, instantiate, and connect it with others, without knowing C++ or even knowing they are written in C++. We had to write a single Adapter class [13] of around 120 C++ lines to connect all the Inventor rendering and direct manipulation facilities to the vtk pipeline. Scalar, vector, tensor, and medical visualizations were created with the vtk-Inventor metaclasses (Fig. 2 a,b,d,f) easier than when using similar systems. The integration required writing around 320 metaclasses, of an average length of 6 lines, and absolutely no change to the two libraries (of which, Inventor was not even available as source code).

Radiosity-based illumination simulation software [10] often requires delicate tuning of many input parameters, and thus can not be used as black box pipelines. Testing new algorithms requires also the configurability of the radiosity pipeline. These options are however rarely available to non-programmers in current radiosity software. We addressed this by including a radiosity system written in C/C++ by us before VISSION was conceived, into VISSION. The simulation's output was easily made available for visualization in the Inventor library by the creation of an Adapter module. Users can now change all the parameters along the radiosity pipeline, easily insert new algorithms by e.g. subclassing, and visually monitor the rendered output (Fig. 2 c).

Finite element (FE) applications mostly come as packages limiting the user's interaction to file input/output. By integrating our FE C++ library [8] in VISSION, we enable researchers to specify and solve FE problems interactively, experiment with different numerical techniques (some were written and loaded in VISSION in less than two hours), and monitor error and convergence rates, without quitting the environment to redefine input files or recompile (Fig. 2 g).

6: Conclusion

VISSION is a general-purpose visualization and simulation framework built on an object-oriented foundation. It offers specification, monitoring, and steering of generic simulations and removes many limitations of similar systems by merging the powerful, yet so far independently used OO and dataflow modelling concepts, in the context of the C++ language.

We enhanced the traditional simulation system dataflow mechanism to an object-oriented one by the metaclass concept, which extends C++ classes non-intrusively with dataflowsemantics. Adding application code to VISSION is simpler as compared to similar systems, and clearly separates application library design from system-specific dataflow information present in the metaclasses. We have provided a mechanism for automatic GUI construction for the OO metaclasses, and a way to add type-specific, user-defined widgets, based on OO typing. VISSION's key design issue, the C++ interpreter/compiler combination, shows that one can combine the advantages of interpreted environments (run-time flexibility, ease of use, reflection APIs) with the speed and extensive set of features of compiled C++ (multiple inheritance, pass by value for user types, etc). We could have implemented VISSION based on the Java Beans component model as well, but we preferred C++ for the sake of direct integration of existing C++ legacy code, speed, and also for the extra design freedom offered e.g. by multiple inheritance and by value data passing.

Several applications illustrate the advantages a fully object-oriented computational steering architecture provides. Component designers could include libraries for scientific visualization and rendering (420 classes), radiosity (18 classes) and finite element analysis (25 classes) in

VISSION in a short time (approximately 2 months, 5 days, respectively 10 days). Application designers and end users could effectively use VISSION in a matter of minutes. The strong separation of pure application code (written by the component designer) from infrastructure as dataflow mechanisms, GUIs, persistence (provided by VISSION) makes the code to be written by the former clearer and also very concise. This is noteworthy since most existing VisSym toolkits [4, 3, 6] (OO or not) dedicate up to 50% of their code to backbone services implementation as the ones we mentioned. Library designers may thus save 50% of their time if the infrastructural services are automatically provided. Moreover, once the backbone can be reused, it is coded just once (in VISSION) and not replicated in endless flavours among the open set of application libraries.

We are extending VISSION with features such as metaclass hierarchy browsing, automatic documentation, and a generalization of the dataflow model to include also *code flow*, that is to have modules exchange fragments of dynamically created C++ code to be interpreted, which should create multiple new possibilities for modelling simulations. Parallel work is targeted at the inclusion of other application domains as numerical iterative solvers and their coupling with the already available libraries.

References

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

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

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

[4] A. M. Bruaset, H. P. Langtangen, *A Comprehensive Set of Tools for Solving Partial Differential Equations: Diffpack*, Numerical Methods and Software Tools in Industrial Mathematics, (M. Daehlen and A.-Tveito, eds.), 1996.

[5] 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.

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

[7] 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

[8] A.C. Telea, C.W.A.M. van Overveld, *An Object-Oriented Interactive System for Scientific Simulations: Design and Applications*, int *Mathematical Visualization*, H.-C. Hege and K. Polthier (eds.), Springer Verlag 1998

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

[10] M. F. Cohen and J. Wallace, *Radiosity and Realistic Image Synthesis*, Academic Press, 1993

[11] 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

[12] S. Rathmayer and M. Lenke, *A tool for on-line visualization and interactive steering of parallel hpc applications*, in *Proceedings of the 11th International Parallel Processing Symposium*, IPPS 97, 1997

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

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

[15] 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

[16] J. Lemordant, *Linear Inductive Reductive Dataflow System for ViSC*, in K. Polthier, H.C. Hege, editors, *Visualization and Mathematics*, Springer Verlag, 1997.

[17] G. A. Geist, J. A. Kohl, P. M. Papadopoulos, *CUMULVS: Providing fault tolerance, visualization, and steering of parallel applications*, in *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3): 224-235, 1997