Visualisation and Simulation with Object-Oriented Networks

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 26 oktober 2000 om 16.00 uur

door

Alexandru Cristian Telea

geboren te Boekarest, Roemenië

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. R.M.M. Mattheij
en
prof.dr.dipl.ing. D.K. Hammer

Copromotor:
dr.ir. J.J. van Wijk

Contents

Chapter 1

Introduction

1.1 Scientific Simulation and Visualisation

The past decades have seen a revolutionary increase in penetration of computer systems in the scientific research domain. The ever decreasing price/performance ratio of computer systems has made computer-based simulation a compelling alternative to real experimentation in application areas as diverse as hydro and aerodynamics, civil engineering, medical sciences, and architectural design.

Nowadays computer applications can accurately model complex phenomena involving large amounts of time-dependent parameters and multidimensional variables. Weather forecast or computational fluid dynamics simulations may generate hundreds of datasets of gigabyte size during a single run. The decrease of the computer price/performance ratio makes it possible to run simulation programs that produce large datasets on desktop computers, in close to real-time. Consequently, the problem of understanding the produced datasets has gained a large attention in the last ten years.

Scientific visualisation adds an important dimension to the process of acquiring insight in the simulated processes. Post-simulation analysis, the visual examination and interpretation of the results produced by a computer simulation after the simulation process has ended, is the most widespread form of data visualisation. Online visualisation or *tracking* represents the next step towards more interaction. The data produced at each simulation time step is directly sent to the visualisation pipeline for inspection. The researcher can monitor the time evolution of the simulated process, stop the simulation when the observed output is considered invalid, and restart the process with a different input. Still, in the case of processes driven by many input parameters, tracking is not a sufficient option. In order to explore the process evolution in another direction in the parameter space, the simulation must be stopped, reconfigured, and restarted.

Interactive process *steering* gives additional insight in the complex time-dependent behaviour of simulated processes. Interactive steering allows controlling of both the visualisation and the simulation parameters in time and offers immediate visual or numerical feedback on the parameter change.

The increase in sophistication of computer simulation and visualisation software has been naturally parallelled by an increased complexity of the involved software systems. Early simulations consisted mainly of monolithic software applications running on supercomputers which read and wrote their input and output data as batch files. Usually such applications were dedicated to solving a small set of specialised problems. With the growth and diversification of computer software, this application model has become uneconomical and inflexible, leading to the advent of scientific computation and visualisation libraries. The driving force behind the appearance of such libraries was the need to capture the growing amount of application domain knowledge in an efficiently reusable form, and thus to minimise the costs and complexity of producing new applications.

The academic research environment with its unique requirements poses a major challenge to the development of simulation and visualisation software. Development of an interactive simulation requires expertise in various domains such as simulation, visualisation, user interfaces, operating systems, networking, and so on. Research applications often have an experimental nature, as they are built to test new concepts, algorithms, data structures, or simply to examine existing problems and datasets in a different manner. The very nature of the exploratory process means that there is often no fixed application skeleton or pipeline. Instead, insight or new ideas acquired during experimentation with an application can determine the need to change its structure by e.g. adding new visualisation functions or replacing a data structure with a more efficient one.

An academic researcher is often a non programmer expert that needs to invest his time in understanding the problem domain, rather than building complex, fine tuned software architectures. The high level of flexibility needed in research environments when building applications for a growing range of problems determines a strong need for software reuse. In short, an easy to use, highly productive way of programming scientific applications is needed.

However, contrary to the hardware advances, estimated to be a factor of 10^6 (in arbitrary units) in the last 30 years, programmer productivity increased only by a factor of 13 [29]. The difficulty to reuse software is one of the main obstacles the scientific research community faces when new simulation or visualisation applications are to be created. The problem is intrinsically hard, as there is no common denominator in the software industry (and thus even less in research communities which are not specialised in software engineering) in terms of software development, assembly, communication, and evolution.

Interactive simulation and visualisation applications have an intrinsically complex structure, involving issues such as event loops, synchronisation, data communication, and graphical user interfaces. Producing such applications quickly and easily in research environments, by reusing existing software elements, is still an unsolved challenge. The time spent to test an isolated new domain-specific algorithm or method is often much smaller than the time needed to provide the software infrastructure needed to test the algorithm interactively or to add visualisation capabilities to it. Consequently, the exchange of algorithms and data structures between scientists as ready to use, plug-and-play components in interactive applications is limited, as compared to the exchange of datasets or pseudocode.

The difficulty of building custom scientific applications by reusing existing components has been addressed in several ways. One of the most successful solutions comes in the form of visual programming environments or application building frameworks, such as AVS [113] or Iris Explorer [44]. Such environments followed the third generation (procedural) programming languages by providing an application model different from the classical monolithic executable and new forms of packaging software into reusable units, as follows [13, 89].

1. reusable *primitive units* of computation out of which applications are easily created
2. a *control mechanism* that drives the assembled primitive units
3. a *data model* used for data representation and transfer between the primitive units
4. a *user interface* that offers the simulation and visualisation functionality to the end user, and also a visual means of assembling components by using a point-and-click graphics interface.

The term 'visual' in the definition of visual programming environments relates to the last point above, i.e. the capability of building applications by using solely a graphics interface. Combined with a powerful set of reusable domain-specific components, this allows building new applications easily and quickly, without writing a single line of code.

However praised, the success of visual programming environments is limited to a small fraction of researchers. An important problem is that the concrete implementations of the visual environment concept are less flexible than what the general concept promised. The flexibility of an application building environment can be seen as a function of the flexibility of its elements mentioned above and of their interplay. These issues are the subject of the next section.

1.2 Limitations of Current Environments

As mentioned in the previous section, there exist many visual programming systems that address the requirements of the research community in what custom application construction, visualisation, and interactive control are concerned. However complying with the above requirements in principle, most such environments have limitations which make their practical use difficult and time demanding. In this section we shall attempt to relate the most important limitations to the structural elements of the discussed environments. An exhaustive survey of visual programming environments and their limitation is given by Hils in [46].

The *primitive units* are the building bricks for all applications involved in the system. The versatility of such a system can be realized either by openness, i.e. the ability to add new primitive units to the existing ones, or by completeness, i.e. the provision of virtually all needed units by the system manufacturer. The latter path is taken by specialised environments that target a well defined application area, such as Matlab [66] or Mathematica [118]. However, in our case this is not an option, as we would like to have an environment able to cover the open field of computational simulation and visualisation. Many components already exist in these fields, mostly in the form of numerical or graphics libraries [4, 14, 116, 94]. Primitive units, called also components or modules, come in various forms, ranging from procedures and functions to classes, script files, and full executables. The capability of easily reusing such components is thus an essential requirement. However, most application building environments provide quite limited freedom to integrate existing software components, as most come with their own policies, languages, and rules for component construction. Object orientation [91], an important well-proven mechanism for software reuse in classical programming environments, starts being addressed only recently and incompletely by visual programming environments. Object oriented techniques are very poorly supported by such environments, which makes it hard to profit from their known advantages for software manipulation, development, and reuse.

The *control mechanism* ranges usually between control flow [3] and data flow [103]. The data flow model maps well the primitive units and their relationships to a visual graph representation, or dataflow network , that can be easily understood and edited. Constructing programs visually by assembling such graphs is a major advantage for researchers who are not familiar with programming techniques, as it enables them to rapidly construct working applications without writing a single line of code. However, an automatic one to one mapping of third generation, control flow based code to dataflow programs is not possible. In order to be effective, a visual programming environment should provide simple, ideally automatic mechanisms to map code entities and relationships from existing application libraries to the visual dataflow elements. The user should be able to choose the right granularity level when mapping existing code fragments to visual elements, in order to produce expressive, concise visual representations of the modelled problem. Operations such as iterative and conditional constructs and dynamic entity creation and destruction should also be supported in the dataflow model [46].

The above point on relationships between primitive units raises the problem of *data communica-*

tion and representation. Virtually every research code or application library has a different manner to represent its data in terms of structures and relationships between these. To be effective, a generic application building environment should be able to support these different data representations and enable the user to easily define data conversion mechanisms. However, most existing environments do not take this path but rather force all embedded components to adopt a system-specific data representation and data passing policy. This clearly limits their usefulness, as researchers are forced either to abandon or massively recode their algorithms to import them in the respective systems.

Finally, the *user interfaces* serve a dual purpose. On one hand, new applications can be easily built by assembling visual component representations such as icons in a dataflow network. On the other hand, end users can interactively monitor and steer running simulations by using various graphics controls such as 3D cameras, sliders, dials, buttons, or direct manipulation elements. The user interface probably makes up for the most prominent difference between several existing visual programming environments. Besides the ease to integrate new or existing software components mentioned above, the flexibility, genericity, and ease of use of user interfaces is the second key element for the success of a scientific visual programming environment. In this respect, an important problem of visual programming environments is the difficulty to build custom user interfaces for e.g. user defined data types. This operation usually requires complex programming of interface code into the application components. For this reason, many scientists prefer to simply abandon the task of integrating their research code into otherwise powerful visual programming environments.

1.3 Scope of This Thesis

We have outlined above why component development, application construction, and interactive simulation and visualisation are all parts of the experimenting pipeline executed in a typical research session. Among the existing systems, visual programming environments address best these issues. However, producing interactive simulations and visualisations is still a difficult task. This defines the main research objective of this thesis:

The development and implementation of concepts and techniques to combine visualisation, simulation, and application construction in an interactive, easy to use, generic environment.

The aim is to produce an environment in which the above mentioned activities can be learnt and carried out easily by a researcher. Working with such an environment should decrease the amount of time usually spent in redesigning existing software elements such as graphics interfaces, existing computational modules, and general infrastructure code. Writing new computational components or importing existing ones should be simple and automatic enough to make using the envisaged system an attractive option for a non programmer expert. Besides this, all proven successful elements of an interactive simulation and visualisation environment should be provided, such as visual programming, graphics user interfaces, direct manipulation, and so on. Finally, a large palette of existing scientific computation, data processing, and visualisation components should be integrated in the proposed system. On one hand, this should prove our claims of openness and easy code integration. On the other hand, this should provide the concrete set of tools needed for building a range of scientific applications and visualisations.

1.4 Thesis Outline

This thesis is structured as follows. Chapter 2 defines the context of our work. The scientific research environment is presented and partitioned into the three roles of end user, application designer, and component developer. The interactions between these roles and their specific requirements are described and lead to a more precise formulation of our problem statement. Chapter 3 presents the most used architectures for simulation and visualisation systems: the monolithic system, the application library, and the framework. The advantages and disadvantages of these architectural models are then discussed in relation with our problem statement requirements. The main conclusion drawn is that no single existing architectural model suffices, and that what is needed is a combination of the features present in all three models. Chapter 4 introduces the new architectural model we propose, based on the combination of object-orientation in form of the C++ language and dataflow modelling in the new MC++ language. Chapter 5 presents VISSION, an interactive simulation and visualisation environment constructed on the introduced new architectural model, and shows how the usual tasks of application construction, steering, and visualisation are addressed. In chapter 6, the implementation of VISSION's architectural model is described in terms of its component parts. Chapter 7 presents the applications of VISSION to numerical simulation, while chapter 8 focuses on its visualisation and graphics applications. Finally, chapter 9 concludes the thesis and outlines possible direction for future research.

Chapter 2

Context Overview

A large variety of scientific simulation and visualisation (SimVis) software is used in the scientific community. As outlined in the previous chapter, the typical requirements for a research SimVis application cover the construction, extension, and use of scientific simulation and visualisation applications. These operations have to be fairly simple for users with a limited programming expertise. Consequently, a method proposed for SimVis application construction and use in research environments should offer enough flexibility and versatility to cover a large class of applications in a simple and generic fashion.

Analysing the SimVis user requirements, we distinguish three user roles involved in the process of construction, extension, and use of SimVis applications. The requirements specific to each role are presented in Section 2.3.

The above requirements have been addressed in practice by many SimVis software systems. These systems are built based on three main software architectural models. Each architecture focuses on covering a part of the above requirements by providing specialised software mechanisms. In section 2.2 these three architectural models are presented. In section 2.3 the presented architectures are compared with the user roles to show that no single architecture can entirely satisfy the requirements of all three roles. This chapter concludes by stating the need to design a new SimVis system architecture. This design is the subject of Chapter 3.

2.1 Definitions

A *software architecture* defines the structure of a software application, or how its software components are connected to each other to form the application [91, 37, 69]. An architecture contains two elements (Fig. 2.1 a): several *components* and the *skeleton*. A component can have in its turn an architecture involving a smaller skeleton and finer grained components. The skeleton provides the data and control communication needed by the components to function as a whole. The components implement specific services (e.g. numerical computations, data transmission, visualisation) which are executed on behalf of the skeleton.

The choice of the architecture for a given software application is important for the success of the final product. To enable an easy construction of applications, a SimVis system architecture should have a generic, reusable, customisable skeleton *and* components. Most used SimVis system architectures are however not complying with all these requirements. Highly generic and reusable architectures often lead to abstract, less easy to customise and use applications. Conversely, architectures favouring customisability and ease of use for the end user tend to be less generic and reusable for different application domains (Fig. 2.1 b). The following section presents the three most used architectures in

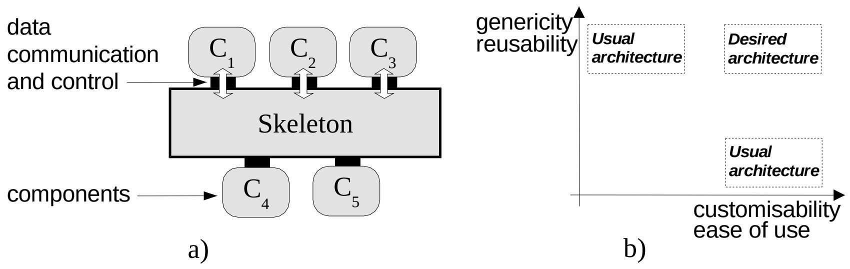


Figure 2.1: Software architecture elements

the SimVis world. The remaining sections detail the requirements of the SimVis user community and present to which extent these are satisfied by the described architectures.

2.2 SimVis Architectural Models

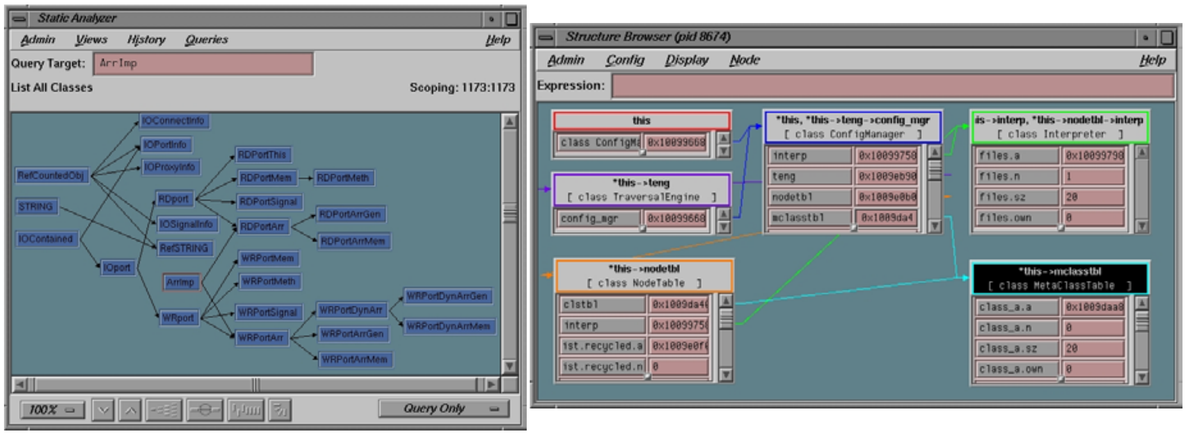
2.2.1 Application Libraries and Development Environments

Software libraries are the most common architectural model for providing services in a reusable form. Libraries consist of a set of related software elements that cooperate to provide functionality for a given application domain. These software elements are accessed via an application programming interface (API). The concrete form of the API depends strongly on the software techniques used to build the library. In this respect, the two main API types employed by libraries are the procedural and the object-oriented one.

Procedural libraries offer their services in terms of functions or procedures, complemented by a set of structured data types, written in a third generation programming language (shortly 3GPL), such as C, Pascal, or FORTRAN. Good examples of computational procedural libraries include the LAPACK library for linear algebra [4, 48]. In the graphics and visualisation area, OpenGL [119] is a well known library, providing an extensive C API to 3D graphics.

Object-oriented (OO) languages introduce more powerful modelling elements such as encapsulation, abstract data types, polymorphism, inheritance, and generic types [91]. These elements allow structuring the library's interface and implementation in terms of the modelled application domain's concepts. This makes OO libraries easier to be understood and more concise in providing the desired functionality. Secondly, object orientation allows decoupling a library's interface from its implementation. This enables providing different (e.g. platform-dependent) implementations of the same interface. Finally, object orientation promotes more flexible couplings between code components than 3GPL techniques. Consequently, OO software is easier reusable and extendable than 3GPL software. Good examples of simulation OO libraries are the Diffpack system for solving partial differential equations [14] or the LAPACK++ library for linear algebra [4]. Visualisation OO libraries include VTK, a visualisation toolkit for multidimensional data representation and visualisation [94], Open Inventor [116] and its similar Java-based Java3D counterpart [104] which perform 3D graphics modelling and rendering, and Vision, a library for realistic rendering implementing various illumination models [99].

Usually, a given library covers a single domain (e.g. only scientific computation or only visualisation). To produce a SimVis application, the user assembles the desired services offered by the involved libraries into a complete program. This is done by writing the application specific skeleton code that calls the libraries' services. Integrated development environments (or IDEs) such as Microsoft's Visual C++ [55] or the CaseVision development system [97] (Fig. 2.2 a,b) help developers in writing such applications by providing various visual browsing, code generation, and graphics user interface (GUI) construction tools.



a) class browser b) object browser

Figure 2.2: CaseVision class browsers (a) and object browser (b)

2.2.2 Turn-key Applications

Turnkey applications [34] are single stand-alone programs that cover a given application domain such as scientific visualisation, computational flow dynamics, or realistic rendering. Turn-key applications perform their operations according to a fixed, hard-coded pipeline, and can be parametrised by different input data sets or batch jobs coming as files. In most of the cases, the results are also delivered by means of data files. Configuration and control mechanisms are provided in the form of a GUI tailored to the specific application needs. For example, the IRIX Showcase [96] computer-aided design application offers a complex GUI to edit and visualise 2D documents and drawings and 3D geometric universes. Many numerical simulation systems come as turn-key applications, either for a dedicated range of problems, such as computational mechanics, or computational fluid dynamics [61, 86]. Similarly, there exist many turn-key visualisation systems, such as the well known applications Vis5D [45] or PLOT3D [114].

The turn-key architecture provides no insight of the application internals to its end users besides the input/output data interface and the eventual run-time control GUI. The operation of a turn-key application is seen as a black box or monolith.

2.2.3 Dataflow Application Builders

Dataflow application builders consist of a set of small to medium sized software *components* that encapsulate various functionality (e.g. graphics, computation), connected to a fixed *dataflow kernel* . The application is actually a network of components that exchange data via a set of well-defined inputs and outputs. When the data inputs of a component change, the dataflow kernel updates the component and transfers the data generated via its outputs to the connected components, which update in their turn, and so on. The components' inputs can be controlled by the end user by means of various GUIs constructed in the application builder. Dataflow application construction is performed by selecting iconic representations of the desired components out of a palette of available components and connecting them by GUI-based point-and-click operations (Fig. 2.3). Existing applications can be easily edited by replacing, re-connecting, or inserting new components. Examples of dataflow application builders are several visualisation systems such as AVS , AVS/Express [113], Khoros [53], IRIS Explorer [44], and Oorange [41] , or simulation environments such as SimuLink [66], SCIRun [81], or VPE [89].

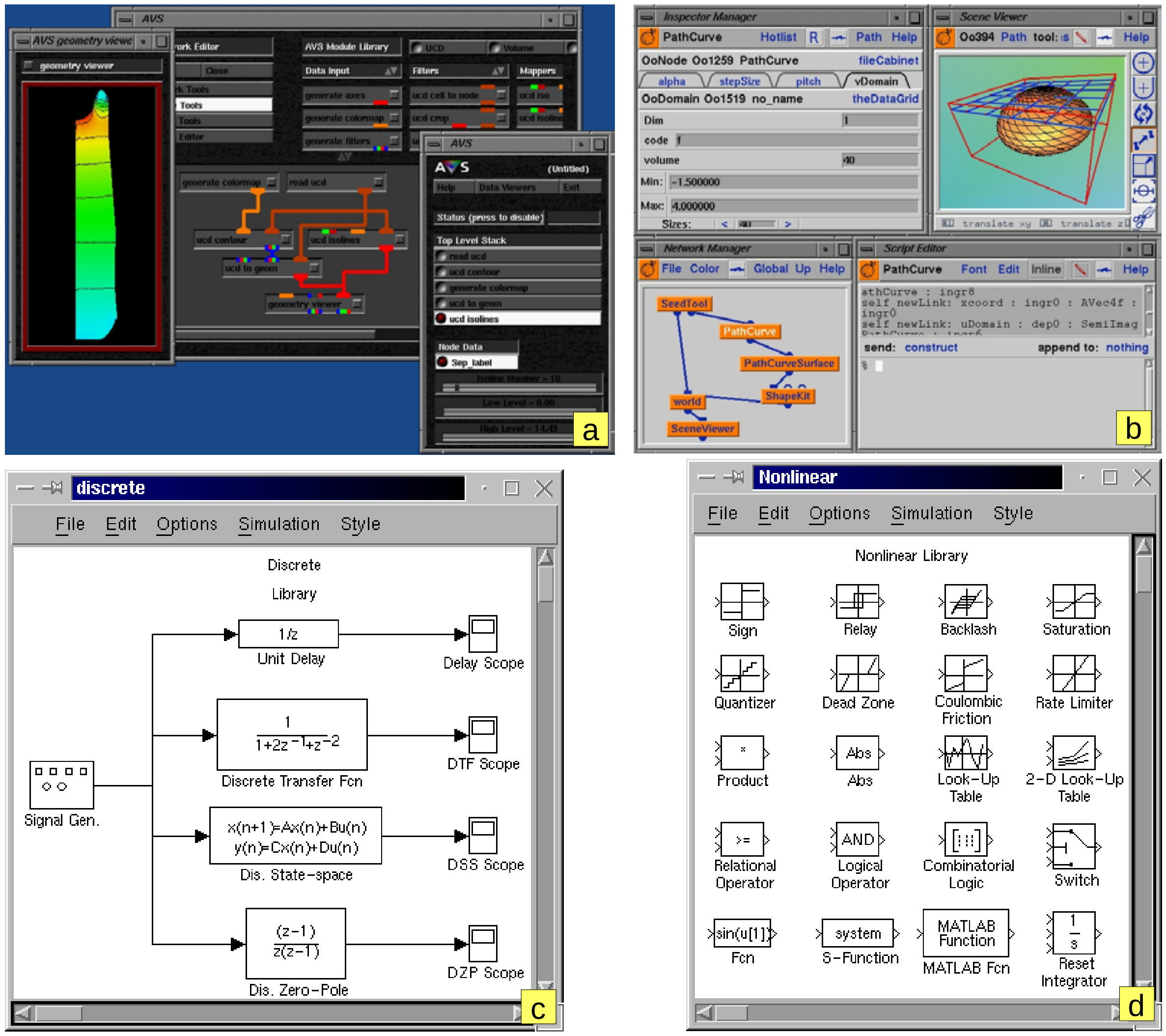


Figure 2.3: AVS visual builder and module library (a). Oorange visual builder and component GUIs (b). SimuLink visual application (c) and component library (d)

2.3 User Categories and Their Requirements

In a generic scenario for SimVis research applications, we identify three categories of users. These categories (Fig. 2.4 b) are presented in the following, together with their specific requirements.

2.3.1 End users

In a typical SimVis application scenario, end users set up the desired simulation parameters, monitor its evolution in time (in the case of time-dependent processes), and visualise the results by selecting the desired visualisation method and tuning its parameters to obtain the desired insight.

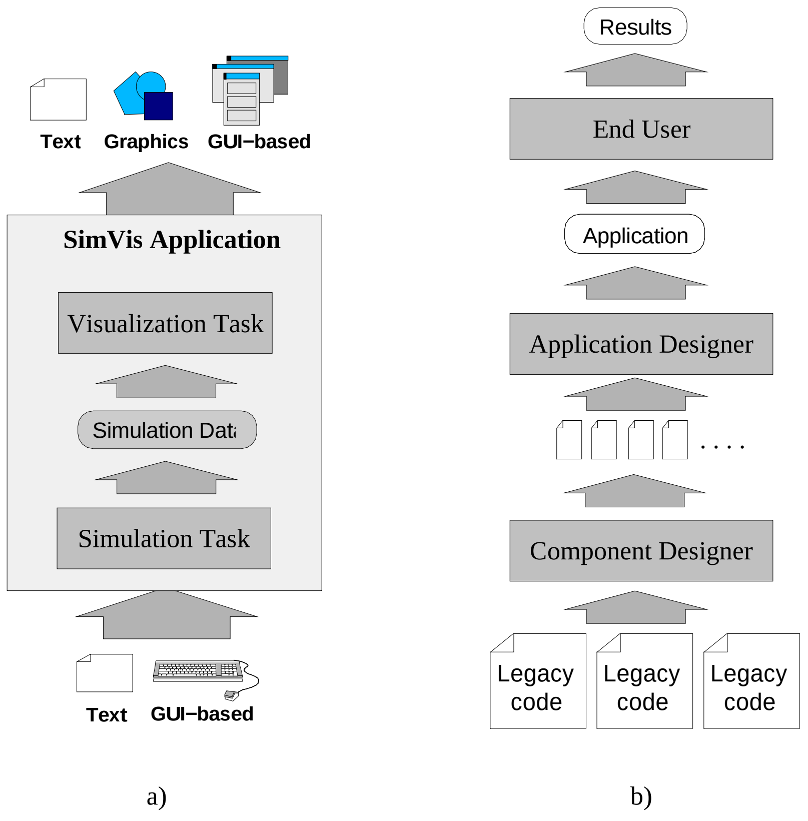


Figure 2.4: SimVis application model (a) and hierarchy of SimVis user roles (b)

In an attempt to unify the simulation and visualisation tasks, we do not make any distinction between simulation and visualisation end users . Both tasks can be treated uniformly as processes that produce textual, numerical, visual, or other forms of outputs, and that are controlled by setting their data inputs to the desired values. The two concepts are naturally unified in a single representation of a functional input/output based unit (Fig. 2.4 a).

End users need several means to control and monitor the parameters of a SimVis task . SimVis applications are classically controlled by means of command-line based consoles, where commands can be input as text fragments and specific queries can be issued. More user-friendly applications provide graphical user interfaces (GUIs) that offer various metaphors for interaction such as sliders, rotary dials, buttons, or other interface elements. GUIs should be available both for data input as well as for output monitoring, similarly to the control panels of hardware devices. Sometimes other interaction paradigms are more convenient for the end user, such as 3D direct manipulation [93], virtual reality techniques, sonification [6], or vision interfaces [18].

Command-line interfaces should coexist with GUIs and direct manipulation interfaces since certain operations are better performed in a text-based dialog rather than by means of graphical controls, or simply because some user categories prefer the former interface to the latter ones.

2.3.2 Application Designers

Chapter 1 stated that a SimVis software environment for the research community should provide a simple way to customise applications to match the end user's needs. This task is done by a new user category: the application designer. Application designers construct the SimVis application required by end users by assembling software components. For example, a numerical simulation based on solving a partial differential equation with the finite element method is constructed by assembling the appropriate mathematical components such as a geometric domain, a discretisation of the domain, a suitable numerical solver and various other linear algebra components. A visualisation application such as the data viewers used in computational fluid dynamics [79] can contain various dataset readers for the data at hand, computational modules for calculating flow gradients or extracting vortices, particle advection or isosurface calculation modules, followed by a 2D or 3D visualisation stage performed by means of an interactive virtual camera.

Similarly to the end user case, we do not distinguish between the construction of a simulation and that of a visualisation. The component assembly process and the definition of a component can take various forms (see [105, 75, 36] for an extensive overview on component-oriented programming and component off the shelf techniques for reusable software). At one extreme, components can be library functions or objects which are joined together by a programmer to produce the desired application, as outlined in section 2.2.1. At the other extreme, visual programming environments enable application designers to build their applications by connecting visual representations of the components, as discussed in section 2.2.3. In the latter case, the application designer may be a non-programmer. A multitude of mechanisms exist between the two extremes that provide a higher level, more flexible definition of a component interface than binary libraries. Well known ones are the Component Object Model (COM) designed by Microsoft [70] and the Common Object Request Broker Architecture (CORBA) [95].

Overall, application designers require an application development environment in which they can construct the simulation, visualisation or a combination of both required by the end-user, and a set of pre-defined components for the desired domains of interest. The ease of finding the right components in the possibly large component repository and of assembling them to yield the desired functionality are two important application designer requirements. A visual application building tool is often the method of choice, as it frees the application designer from the difficult task of writing code, and it provides a simple GUI-based way to browse and combine components.

2.3.3 Component Developers

Component developers are the last SimVis user category. They build the components used by the application designers, either by writing these components in a programming language from scratch or by reusing existing components by specialisation or extension. In our context, component developers are researchers that need to implement custom algorithms or data structures, or modify existing ones. Different component developers may work independently to design component libraries for separate application domains. Component developers are mostly programmers, in contrast to the previous two user categories which mostly focus on non-programming tasks.

Currently a vast amount of component libraries exists that cover various SimVis-related application domains such as 3D graphics, realistic rendering and modelling, scientific visualisation, data and image processing, linear algebra, finite elements and differences, etc. An important requirement of component developers is the ability to easily (re)use these so-called legacy libraries, to produce components for the application developers. This task is made difficult by a couple of factors. The most important one is the lack of a common denominator in the structure of legacy software. This makes otherwise well structured and self-consistent libraries incompatible [106, 85].

2.3.4 Requirements Overview

Similar hierarchical user models to the one presented above are encountered in various fields of computer science [69, 27, 6, 88, 85]. In our case it is better to speak about different *roles* rather than different *users*. Indeed, the same person can go through all three situations, e.g. a researcher who develops his own code as a component developer, then builds test scenarios for his algorithms as an application designer, and finally monitors and/or steers the final SimVis application as an end user. The process can cycle, e.g. in the case that the end user's insight leads to a redesign of the application, which may trigger the need for new or modified components [69].

Easily switching between the three user roles is an often neglected requirement of SimVis software environments. Such environments often focus on providing functionality for a single user role. Neglecting the other two roles or providing insufficient support for an easy role transition is a serious problem for the scientific researcher, who must often perform all the tasks involved in a SimVis application (coding algorithms, assembling the application, and running it) by himself.

A role in the hierarchy relies on the services provided by the lower role and provides services to the next role in its turn. The usability requirement of a SimVis software can thus be translated to the fact that the role coupling, seen as a data flow process between the roles, is highly (preferably completely) automatic:

- the component developer should be enabled to easily reuse legacy code to create components, and also to extend and specialise existing components to create new ones.

- the components created by the component developer should be immediately available to the application designer who should easily be able to produce the VisSim application required by the end user.

We can now summarise our SimVis user requirements. These consist of intra-role requirements, i.e. the specific requirements of the three roles presented above (interaction facilities for end users, application construction facilities for application designers, and component development facilities for component developers), and the inter-role requirement, stating that the role transition should be a transparent, highly automated procedure.

The problem statement introduced in chapter 1 is reformulated in terms of user requirements as follows:

Design a software system for simulation and visualisation (SimVis) that:

- addresses **all** the intra-role requirements of the end user (Sec. 2.3.1), application designer (Sec. 2.3.2), and component developer (Sec. 2.3.3) user categories;

- treats all requirements of all user roles with **equal** priority - that is, does not favour the satisfaction of one role's requirements in the detriment of another role's requirements;

- provides a simple, ideally automatic role transition mechanism such that users can **easily** change roles at any time;

- offers powerful and generic software **tools** for every role. These tools should smoothly complement each other, such that the resulting SimVis system offers the union of their separate advantages and minimises the union of their disadvantages.

2.4 Architecture Comparison

The above sections have presented the three main architectural models used for SimVis systems: the library and IDE model, the turnkey model, and the visual application builder model. Next, the specific requirements of the three SimVis user categories (end users, application designers, component developers) have been presented. This section will discuss the extent up to which the discussed architectures support the above requirements. The purpose of this comparison is to establish the limitations of the presented architectures and to outline their strong aspects. Throughout the discussion, the labels **CD**, **AD**, **EU** mark whether the presented advantages and disadvantages affect the *component developer*, *application designer*, or *end user* respectively. Based on this analysis, we shall develop a new SimVis architectural model which complies with the requirements presented in the previous section.

2.4.1 Applications Libraries and IDEs

Advantages

1. The component developer has total freedom to model any application domain in any manner, up to the inherent limits of the underlying development programming language. The same is true for the application designer (**CD**,**AD**).

2. Decoupling functionality from context when building libraries makes them highly reusable and also portable in a variety of contexts. Libraries, especially in their OO variant, have been the most effective form of software reuse and tailorability [26] (**AD**).

3. Libraries usually have a simple control structure, as this is part of the context in which they operate and thus has to be provided by the skeleton of the application in which they are assembled. This makes them rather simple to design and extend, as the component designer can focus strictly on modelling the application domain at hand (**CD**).

4. A huge number of libraries exist that covers practically any desired application domain. It is thus relatively easy to produce a custom library for a specific domain by specialising and/or extending existing legacy code. This can reduce the component development and testing costs considerably as compared to building a library from scratch (**CD**).

Disadvantages

1. The main disadvantage of the library architectural model is that the application designer must explicitly build from scratch the application skeleton code. Since libraries usually provide context-free services, all the end application's specific behaviour (e.g. control structure) has to be manually written by the application developer. This requires usually extensive programming knowledge and development time. A classical example is the construction of an application that has to combine numerical computations, graphics, and user interfacing, all provided by different libraries. Producing an end-application is thus not a simple, automatic task as our problem statement demanded (**AD**).

2. The APIs of many libraries are often collections of low-level programming language constructs (e.g. functions and structured types for procedural libraries). The application designer needs extensive skills and time to understand them. Moreover, every library has a different structure for its API and many such APIs are not clearly and/or uniformly documented [85]. Understanding how to couple components from several independently developed libraries requires usually an effort proportional with the sum of the libraries' sizes (**AD**).

3. Independently developed libraries often have incompatible interfaces and usage policies. If the integration task becomes too hard or time-consuming at the application design level, component developers may have to restructure an existing library, adapt it by wrapping or delegation, or even rewrite it from scratch (**CD**).

2.4.2 Turn-key Systems

Advantages

1. Turn-key systems are built with the goal of covering a specific application domain. Since the application domain is precisely known and fixed, turn-key systems can offer the most convenient and easy to use mechanisms for the end user. For example, the problem-specific graphics interfaces and dialog scenarios of turn-key applications (e.g. CAD editors) are more convenient to use than the generic ones of e.g. dataflow visual systems, which are sometimes too abstract. This is an essential advantage of turn-key systems from the point of view of the end user. Consequently, the vast majority of software applications, including the SimVis domain, are turn-key applications (**EU**).

2. Turn-key applications hide their implementation completely from the users, so their developers are free to implement the provided features in the most convenient manner, both from the point of view of the development cost and of the speed of the resulting product. For example, the turn-key model has no imposed system modularity constraint as the library and application builder models exhibit.

Disadvantages

1. The main disadvantage of the turn-key model is closely related to its main advantages. Since turn-key systems offer highly specialised mechanisms for modelling a given application area, they are strongly domain-dependent. This means that it is very hard, often impossible to extend a turn-key system to cover another application domain than the one it has been designed for. Similarly, their closed structure makes it often impossible to be extended even within the same application domain. For example, it is relatively easy to extend a visualisation system such as AVS to perform also some numerical computation tasks. In comparison with this, it is almost impossible to extend a turn-key application such as the PovRay raytracing system [31] to perform radiosity computations without a massive overhaul of its source code. This conflicts with our requirements, since, for example, a researcher often needs to customise his experimental applications by changing the involved algorithms or data structures or interconnecting them in different ways.

2.4.3 Application Builders

Advantages

1. Factoring the application into context-independent components and an application domain dependent kernel means that the domain dependent code (the kernel) is constructed just once. The context independence advantage offered by the library architectural model is preserved. The main disadvantage of the library model, i.e. that the context has to be manually written every time a new application is designed, is largely removed (**AD**).

2. Building an application visually from existing components is relatively easy as compared to the library model. Programming a new application is reduced to simple point-and-click operations. Such systems come close to our initial requirement that the application designer should be able to easily build an application by assembling already existing components (**AD**).

3. A dataflow kernel offers a component interface at a higher level than the one offered by the library model in terms of the constructions of a programming language (functions, classes, etc). Since the kernel is designed to model a certain application domain, it can expect domain-specific information in the component interfaces. By using this information, the kernel can infer more about the components and thus provide several features automatically to the application designer or even to the end user. For example, if the component interface uses a special naming of component services, an automatic help, documentation, or component service browsing facility can be implemented. This technique is used by e.g. the ROOT data management system [15] to browse the services of its C++ components. The Oorange and AVS visualisation systems use a similar mechanism to automatically construct the iconic component representations from the components' input and output specifications. The Visualisation Studio [84] works similarly to offer a visual representation to its VTK-based components (**AD,EU**).

Disadvantages

1. The main disadvantage of the application builder architectural model is strongly correlated to its main advantage. Assuming some knowledge about the component interfaces limits the freedom of modelling applications outside that domain considerably as compared to libraries, which are by definition context-independent. This is especially unpleasant if we need to cross the borders of a traditional application domain as we do when we address the integration of simulation and visualisation. In this case, it is hard to use e.g. an existing visualisation framework to perform simulations, or conversely, since that means forcing a given system to cover an application area outside its original scope. Although this can be done by choosing to model both application domains using the framework mechanisms best suiting one domain, this will force the component developers to recode or re-interface large component libraries in an unnatural way (**CD**).

2. As explained above, the more specialised the component-kernel interface is, the easier it is for the framework to provide higher-level component-related services. On the developer side however, more specialised interfaces mean more design constraints one must comply with. This makes the development of new components a difficult process. Moreover, this provides an undesired two-way component-framework coupling: not only will the framework assume some knowledge on the components, but the components will also assume some knowledge on the framework. Consequently, components developed for a specific framework will loose their main advantage, i.e. the context independence. Such components will be unusable in another context than the framework for which they were designed. Similarly, independently developed components will need to be modified to comply with the restrictive framework interface. We call this *intrusive component integration*, since the design of the framework intrudes explicitly in the design of the components [106, 85].

Intrusive component integration is the most severe problem of application building environments, as pointed out by a number of authors [91, 69, 101]. Frameworks based on object-oriented inheritance for component introduction (also called white box frameworks [35, 80, 101]) are a clear example, as the developed components have to explicitly inherit from the framework's base classes. Intrusive integration is also present in other systems which do not require components to inherit from their structures (also called black-box systems) . For example, some systems require a strict class or procedure naming or code annotation policy in order to perform the component integration [15, 14, 94]. Other systems require the insertion of specific system calls in the code of the developed components [113, 117, 86, 50, 79] (**CD**).

2.4.4 Conclusion

The preceding comparison of the three SimVis architectures is summarised in Table 2.5. At this point, it is obvious that the turnkey and library architectural models have fundamental limitations which preclude us from using them for satisfying the targeted combination of user requirements. Both architectures offer insufficient freedom for application design. Indeed, the turnkey model has practically no application design process, as it models a single, immutable application structure. The library model allows extensive freedom in designing applications, but requires a too large amount of manual programming to be done.

	Libraries and IDEs	**Application Builders**	**Turnkey Systems**
CD effectivity	+ many libraries available + context independence	+ system is open for extension − intrusive integration is frequently only option	− no component development
efficiency	+ OO helps reuse, modularity, extensibility	− hard to integrate/extend components, especially OO − intrusive integration	
AD effectivity	+ freedom to write application − writing new applications requires programming skills	+ dataflow model covers most application scenarios + automatic documentation	− no application design
efficiency	− writing new applications is laborious − learning libraries is complex	+ visual programming is easy to use and learn + visual libraries easy to learn	
EU effectivity	− hard to get custom−designed applications with GUIs, interactivity, visualisation	+ most SimVis applications can be easily provided − hard to get new GUI widgets	− specialised GUIs and custom applications for all domains
efficiency	− custom systems are less performant than turnkey commerical systems	+ GUIs, steering,command−line interfaces well supported	+ easiest to use applications + optimised for memory and speed

Figure 2.5: SimVis architectures vs user roles

In reality, there are no precise boundaries between the SimVis architectural models described in this chapter. This can be visualised as a software continuum with application libraries at one extreme and turn-key applications at the other extreme (Fig. 2.6). Libraries are less specialised and context-dependent, therefore more easily reusable and customisable, while turn-key applications fall into the other extreme. Consequently, component developers will prefer the library model, while end users will be more comfortable with ready-to-use turn-key systems.

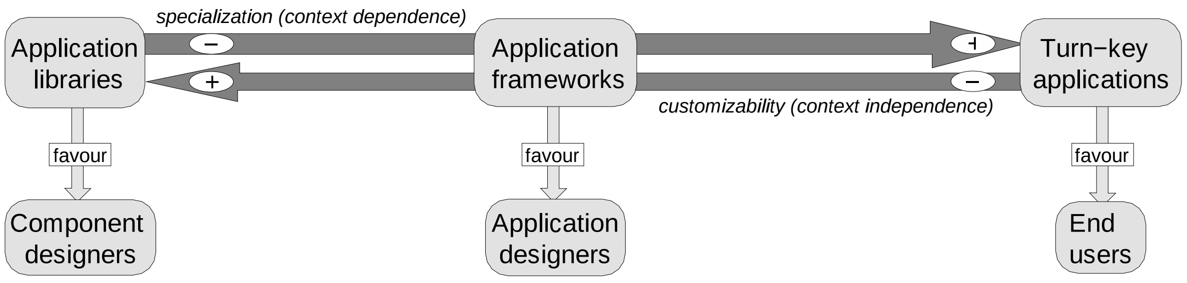


Figure 2.6: SimVis software continuum

Application builders are an interesting middlepoint which combine the advantages of both extremes, leading to the appearance of the separate user class of application designers. As this architectural model is the only one addressing the easy (visual) application construction requirement, we shall focus our analysis on this class of systems in the next section.

2.5 Visual Application Builders

Visual application builders seem an appropriate architectural model for our SimVis user requirements. However, as discussed in section 2.4.3, these systems exhibit some limitations, mainly related to the component developer role. In order to better analyse the cause of such limitations, the elements of the application builder architecture are discussed in further detail.

As introduced in chapter 1, a visual application builder consists of reusable primitive units or components, a control mechanism, a data model, and a user interface. In the following section, these four elements are discussed in relation with the requirements of the three user categories introduced in section 2.3.4.

2.5.1 Requirements

Primitive Units

The primitive units are the basic building bricks of any SimVis application. These units come mainly in the form of components in 3GPL or OO libraries, such as procedures and classes. The requirements with respect to the primitive units are:

- ability to easily integrate components from existing SimVis libraries in the application builder. Many of these libraries, such as Diffpack, VTK, or Open Inventor come in an object-oriented form. The application builder should be able to integrate these OO components.
- non intrusive integration. As explained in section 2.4.3, intrusive component integration is a major problem of many application builders. In order for such an environment to be effective, one should be able to integrate computational or visualisation components into it without having to change their code.
- ability to easily develop, specialise, or modify in any way the existing components.
- components should execute efficiently in terms of speed. For optimal efficiency, components should be compiled to executable binary code.

- hierarchical component creation should be supported, i.e. one should be able to easily build coarse-grained components out of fine-grained components. Hierarchical component construction is necessary to allow arbitrarily complex applications to be easily constructed by manipulating components on several levels of granularity.

Control Mechanism

The control mechanism of most SimVis application builders uses a dataflow mechanism. Such a dataflow mechanism can be implemented in two ways:

1. **Demand driven:** Demand-driven implementations use a lazy evaluation procedure. Whenever a component input changes, the component is marked as modified without updating its outputs. When the end user requests to examine, visually or numerically, an output of a modified component, the kernel recursively updates all modified components whose outputs are connected to this component. This is done in a two-pass network traversal. The first pass starts from the component interrogated by the end user (the 'demand') backwards along the output-input links until no more modified components are found. The second pass traverses the network back to the demand origin, updating all components found modified in the first pass. Demand driven kernels are efficient since they update only those components that produce the output demanded by the end user. However, they execute synchronously and thus can not include asynchronous actors such as external signal sources.

2. **Event driven:** Event-driven kernels update the whole dataflow network immediately after a component's inputs are changed, by traversing all components whose inputs depend on the modified component and updating them. Event-driven kernels might be slower than demand driven ones, but have two important advantages. First, the dataflow network is always up to date, which is desirable when managing highly dynamic, interactively changing networks. Second, the incorporation of asynchronous external event sources is immediate.

For the above reasons, we prefer an event-driven dataflow implementation.

Another important point is the support for cyclic dataflow networks. One can show that virtually any imperative control structure can be mapped to a dataflow network if cycles in the network are allowed. Since iterative processes frequently occur in simulation applications, support for cyclic dataflow networks is required.

Data Model

The data model encompasses the way data is represented and communicated in a system. Virtually every application library or research code has its own data representation in terms of data types. The same holds for various data passing mechanisms, such as by value, by reference, by counted reference, by pointer, etc. The non intrusive integration requirement implies that component libraries *and* their data representation and passing mechanisms between components should be integrated as such in the application builder. In other words, the data types flowing in the dataflow network should be exactly the ones created by the component developer, not a restricted set of 'system' types. This is however a rarely addressed requirement by most existing SimVis application builders, as discussed further.

User interfaces

Visual programming application builders have two sorts of user interfaces. First, a *network editor interface* is provided to the application designer to interactively assemble a dataflow network out of component icons. The requirements for this interface are as follows:

- the components provided by the component developer should automatically be available in a visual (e.g. iconic) form in the network editor's component browser. No extra work should be performed to provide a software component with a graphical representation. It should be possible to dynamically load and unload component libraries in the system with the same ease as one loads or unloads data files in some visualisation program.

- building a dataflow network should involve no programming knowledge regarding the internal component structure. Application design should be done entirely in the visual conceptual space. All required information for component finding, creating, destruction, and connection in a network should be available visually.

Besides the application design interface, an end user interface should be available. Such an interface should provide various methods of controlling the parameters of a SimVis application and monitoring its results. In practice, four main types of end user interfaces occur:

1. **Batch interaction:** In this mode, the actions desired by the end users are listed in a batch or script file which is input to the SimVis application that executes it. Batch interaction usually does not allow the end user to intervene and steer the process further during the batch execution. However, this interaction mode should be supported as it is very common in the computational simulation world.

2. **Command-line interaction:** In this mode, users can type commands which are interpreted on the fly, during the application execution. Such commands can be gathered into a file, in which case one reverts to the batch technique. Command-line interaction should be provided for all user categories. End users may need to set or query the inputs and outputs of dataflow components. Application designers may want to build applications by typing in the construction commands interactively. Component developers might adjust or even write components from scratch using the scripting language.

3. **Graphical user interfaces:** By their very nature, SimVis applications use visualisation of 2D or higher dimensional objects to convey insight in the data at hand. The data are transformed into geometric objects, which are shown on the screen. Colour and texture are often used as cues for data values. Simulation steering is done by GUIs providing sliders, buttons, dials, and other controls to monitor and change the simulation parameters. Providing GUIs for the application components should be simple, ideally automatic operation, similarly to the provision of component icons for the application design. Moreover, such GUIs should be easily extensible with new widget types, in order to cover custom application needs.

4. **Direct manipulation:** In order to overcome the expressivity problem of the classical GUIs, one needs a simple way to design more specific, direct ways of manipulating and representing graphic information [93]. In standard GUIs, the input is visually separated from the output. In direct manipulation interfaces, these are integrated. The user can directly interact with the data, presented as 2D or 3D objects. Direct manipulation is supported for instance by the Inventor graphics library [116], which offers 3D interactive manipulators with predefined functionality,

and by the Computational Steering Environment (CSE) [117] which allows the user to construct parametrisable geometric objects (PGOs) by assembling simple geometries.

Direct manipulation is indispensable for SimVis applications. However, there is still no consensus about the best way in which to structure and provide multi-dimensional direct manipulation services. The cause is that 2D, 3D or nD manipulation scenarios are too different to be easily reduced to a closed set of operations. Consequently, a generic SimVis environment should provide a few of the most usual 2D and 3D manipulation tools and also a simple way to design new, application specific ones.

In conclusion, there is no best interaction technique to be preferred above the others. Ideally, a SimVis application should provide all of them, since they perform the best for different users or for different phases of building and using a simulation or visualisation.

2.5.2 Single vs Dual Language Frameworks

The previous two sections mentioned intrusive code integration as one of the most severe limitations of SimVis application builders. Since this issue is going to play a crucial role when analysing existing SimVis systems, we shall elaborate more on the origins of this problem.

Two technical issues are of importance in the process of component integration in SimVis application frameworks. These issues concern the component programming language, as follows:

- **Execution model:** is the implementation language compiled or interpreted?
- **Uniformity:** is there a unique implementation language or several languages?

The first issue is related to execution efficiency. In virtually all cases, SimVis components have to come as compiled code for maximal efficiency. This is especially critical in applications such as computational steering where computations on large datasets have to be performed in real time. However, the visual instantiation and assembly of components into a dataflow network has to be done dynamically, at run-time. The above requirements are both satisfied by having two different languages for the compiled, respectively interpreted parts (Fig. 2.7). Components are implemented off-line, in a compiled language, and can be dynamically loaded by the (already running) kernel. Applications are interpreted by the kernel which calls back for services on the compiled components. All run-time actions issued by the application designers to the kernel are expressed and executed in the dynamically interpreted language. Sometimes the end user can also issue commands to be interpreted by the kernel, either by typing statements in the interpreted language, or by performing GUI-based operations which are translated to interpreted statements.

The compiled and interpreted parts of such systems are usually done in different languages. The rationale of this is that components are best developed or already exist in a ’classical’ compiled language (e.g. C,C++,FORTRAN), whereas it is either difficult or impractical to support the same language in interpreted form for the usually simpler run-time actions. Moreover, the framework kernel encodes domain-specific knowledge, for which a custom-designed interpreted language may be more suitable than a general-purpose one.

However widespread, dual language frameworks have two fundamental drawbacks [110]. First, the transition between the component developer and the application designer roles involves learning two languages. Secondly, this transition is seldom automated, as mapping software abstractions between two different languages is a complex task. Often the compiled language has more powerful concepts than the interpreted one, which forces restriction or recoding of components. The most encountered language incompatibilities that make dual language solutions hard to use are:

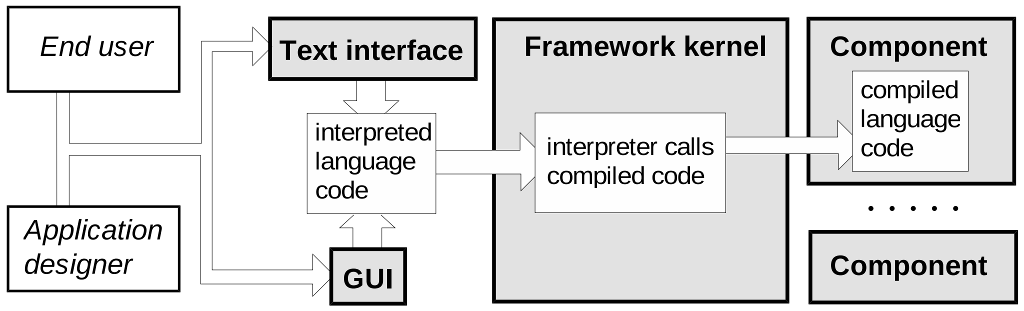


Figure 2.7: Dual language framework implementation

- Interpreted languages have often a simpler typing system than compiled one. The former support often only a few basic types such as integer, float, boolean, pointer, and arrays of these [113, 66, 94], and are not extensible with user-defined types.
- By value and by reference data passing are rarely supported both by the interpreted languages, although compiled ones do.
- To provide run-time conversion between data types, explicit conversion modules [113, 41] or complicated run-time schemes to register conversion functions [116] are used. Compiled languages have more elegant schemes such as conversion operators or copy constructors in C++ [102].
- Most OO dual language frameworks do not support multiple and virtual inheritance in the interpreted language, even though the compiled one (e.g. C++) does. Mapping interfaces of class hierarchies between the two languages is often done by manually designing a class hierarchy in the interpreted language which parallels or 'wraps' the compiled one. Various design patterns have been proven useful here, such as Bridge, Adapter, or Decorator [37]. Their manual coding, however, is very difficult when mapping complex class hierarchies.
- Many dual (but also single) language frameworks force the components to inherit from some common kernel base-class, therefore forbidding class libraries with multiple roots [82, 15]. This form of white-box integration forces massive restructuring, wrapping, or even rewriting components for integration. White box integration is a serious problem for commercial libraries whose sources are not available or not permitted to be changed.

The structural limitations of dual language frameworks propagate often to the end user as well. Many such frameworks offer GUIs to monitor and/or steer a SimVis application by inspecting or changing various component parameters. Such GUIs can be visually constructed by assembling various widgets offered by the system, such as sliders, buttons, text type-ins, and so on (Fig. 2.3 a,b). However, due to limited support for user-defined types of the underlying interpreted language, it is very difficult to build GUIs with custom widgets for user-defined types, e.g. a three-dimensional trackball widget to edit a 3D vector data type. The end user has to content himself with GUIs that map the domain-specific data types to the restrained set of system supported types (integer, float, string, etc).

2.5.3 Existing Application Builders

Several dataflow application builders are available today. In this section, several such systems will be reviewed and compared against the requirements set presented in the preceding sections.

AVS

The Advanced Visualization System (shortly AVS) [113] is probably the most known SimVis application building environment. AVS evolved from its initial release to the AVS/Express system which offers several enhancements in what code integration is regarded. Both AVS and AVS/Express focus mainly on data processing and scientific visualisation, although they can be used to build and steer simulation applications as well. In the following both AVS and AVS/Express systems will be discussed, as they both have separate advantages and disadvantages.

Primitive Units. The basic building blocks of AVS applications are called *modules*. An AVS module is implemented as a compiled FORTRAN subroutine or C function. AVS/Express allows also using a restricted subset of the C++ class language construct to create modules. Basic modules can be aggregated into macro modules in order to describe more complex operations. Macro modules are described in a proprietary interpreted language, called `cli` for AVS and `V` for AVS/Express. Besides hierarchical module construction, these languages permit programming various control sequences such as iterations, and conditional and selection statements.

AVS uses intrusive component integration. The application code must be modified in order to insert AVS library calls for module initialisation, finalisation, port declaration, data passing, and so on. For small and medium sized modules, the module code becomes usually a mix of 50% application code and 50% AVS system code. First, this makes understanding and extending module code very difficult. Secondly, integrating existing code in AVS means practically a complete rewrite of that code.

AVS and AVS/Express are dual language frameworks . However, the `V` interpreted language used by AVS/Express is less powerful than compiled OO development languages such as C++. Consequently, integrating a C++ class hierarchy into AVS/Express would require major restructuration of the library, such as, for example, restricting it to single inheritance only.

Control mechanism. Both AVS and AVS/Express use an event-driven dataflow mechanism. These systems make a distinction between downstream and upstream dataflows, i.e. dataflows going to, respectively coming from the viewing component. The distinction reduces the freedom of building arbitrary networks such as networks containing loops. This distinction is purely artificial and is motivated only by internal implementation limitations of the systems.

Data model. AVS supports a fixed set of system data types such as int, float, string, fields (n-dimensional data arrays), and unstructured datasets. AVS/Express extends this by complicated mechanisms which allow the inclusion of some user defined data types, such as C structs or, up to some extent, C++ classes. Since support of user data types is very limited and complicated to achieve, is is almost never used in practice, especially in the case of object-oriented types. Data passing is basically by value for AVS system types and by pointer for user defined types. Genuine OO by-value passing is not supported, as it implies notions such as nested constructor and destructor calling [102].

User interfaces. AVS and AVS/Express offer a comprehensive network editing interface. Per-module GUIs are constructed based on the component code annotations. Alternatively, these GUIs can be built visually in a GUI editor by assembling various widgets provided by AVS, such as sliders, lists, buttons, etc. However, extending the AVS widget set with user-defined widgets is not an option provided by these systems.

Iris Explorer

The Iris Explorer system was originally built by Silicon Graphics, and was further extended by NAG [44]. Iris Explorer is very similar with AVS/Express is most respects, as outlined next.

Primitive Units. The basic building blocks in Iris Explorer are called modules. Modules are assembled in a dataflow network, called a program map. Modules are implemented as C or FORTRAN functions. In contrast with AVS/Express, no support is provided for object-orientation, so it is very difficult to build modules out of C++ code. As in AVS, modules can be aggregated to create compound modules.

Iris Explorer uses a partially intrusive code integration strategy. User code doesn't have to be manually modified by inserting system calls to create modules. Instead, modules are built via a GUI-based module builder tool. In this tool, the module ports are declared and associated to the computational function's arguments. The module builder generates automatically a wrapping code which adds Iris Explorer API calls to interface the user code with the system kernel. However, several implicit limitations are placed on the argument types of the computational function, as explained further.

Control mechanism. Iris Explorer uses an event-driven dataflow mechanism. Networks with loops can be constructed. In this respect, Explorer offers more freedom than AVS.

Data model. Explorer offers a set of system types, such as int, float, string, and lattice (a n-dimensional data array). The arguments of the user-written computation function are restricted to the fundamental C types and to opaque pointers, which are mapped in the module builder to fields of the lattice type. User defined types are supported in a restricted sense. These have to be declared in the proprietary language ETL, which supports a limited version of C-like structure data type. Explorer is thus a dual language framework, similarly to AVS. There is no support for passing data by value between modules or for object-oriented (e.g. C++) types. Similarly to AVS, the complexity of defining and using new types forces in practice component developers to restrict themselves to the fixed set of system types. Consequently, integration of existing SimVis code in Explorer is a difficult task.

User interfaces. A comprehensive network editing interface and module GUI building tool are provided, similarly to AVS. However, introducing new GUI widgets is very complex as it requires intimate knowledge of the Motif widget library [33].

Oorange

Oorange is a visual programming environment which focuses on experimental mathematics [41]. However, any simulation or visualisation application can be built in Oorange, provided that the desired modules are created. Contrary to AVS and Explorer, Oorange has an open structure, as it is built on a shareware software basis. This makes it more easily extendable in some respects, as discussed next.

Primitive units. An Oorange application is built from modules, just as in Express and AVS. An Oorange module is an Objective C class, which contains a C data structure and several associated methods [83]. Modules can extend or specialise other modules, by using Objective C's single inheritance mechanism. This offers a flexible way to customise existing Oorange modules or create new ones from C code. Modules are compiled into libraries which can be dynamically loaded by the system. However, Objective C is not a fully interpreted language. In order to achieve more run-time freedom, Oorange modules are wrapped into a `Tcl` interface. Oorange is thus also a dual language framework. Writing new Oorange modules involves advanced knowledge of the interplay between `Tcl` and Objective C.

Control mechanism. Oorange uses an event-driven dataflow model. However, no loops are allowed in the network. An useful feature of Oorange is the fact that the module computation function (written in a mix of `Tcl` and Objective C) can be easily edited at run-time to customise the module behaviour.

Data model. There is actually no data flow between Oorange modules. Similarly, there are no system data types, like in AVS or Explorer. The network modules are connected via references. These are actually C pointers which refer to each others' Objective C classes. It is the module's responsibility to extract data from the upstream modules and to write the results in the downstream modules' data structures. On one hand, this makes the module code more complex. On the other hand, this allows any data passing protocol between modules, up to the limits of the Objective C language.

User interfaces. Oorange has a visual network editor and per-module GUIs similar to those of AVS or Explorer. However, the underlying `Tcl`-Objective C language combination is present in the end-user interface, as some operations are provided via GUIs, while others involve editing `Tcl` scripts or issuing `Tcl` commands in a console window. Moreover, providing a GUI for a user-written module involves manually writing it from scratch in a `Tcl-Tk` language combination. Overall, using Oorange involves knowledge of two languages: Objective C, and the `Tcl-Tk` combination.

The Visualization Studio

The Visualization Studio [84] is a visual programming environment built atop of the Visualization Toolkit , or shortly VTK [94]. VTK is a C++ class library for scientific visualisation and data processing. VTK offers over 400 classes covering charting, graphing, 2D and 3D data presentation, interactive data manipulation, flow visualisation, medical imaging and image processing, etc. The provided functionality exceeds the one supplied by the standard AVS or Explorer modules.

Primitive units. VTK's primitive units are C++ classes which inherit from some system classes defining data passing, synchronisation, and other management mechanisms. The VTK class library is well structured to cover concepts such as data readers, filters, viewers, mappers, actors, etc [91]. Developing a specialised component such as a new filter or mapper requires less coding than in AVS, Explorer, or Oorange. This is done by subclassing the desired VTK component and overriding one or more virtual functions. In this respect, VTK is a white-box, single language framework.

However powerful, extending VTK by subclassing is not a simple operation, as it requires deep understanding of the VTK class hierarchy and of advanced OO modelling concepts such as design patterns [37] . Writing a module with several inputs and outputs, for example, is a complex procedure, involving the overriding of a variety of methods.

Control mechanism. VTK uses a demand driven dataflow mechanism, coded in the library's base classes. Combined with automatic multithreading support, this leads to a very efficient pipeline execution. However, the provided demand driven implementation makes it very difficult to incorporate external data sensors, e.g. programs or modules which do not inherit from VTK's base classes.

Data model. VTK provides a set of object-oriented data types such as regular, irregular, curvilinear, and unstructured grids. User defined data types, OO or not, can be relatively easily incorporated by subclassing the appropriate VTK data set class. Data passing is essentially by pointer, with a reference counting scheme built in the base classes. By value passing of datasets is not supported.

User interfaces. The Visualization Studio provides a network editor interface to the VTK library, as well as per-module GUIs. Since all VTK modules inherit their data inputs and outputs from a known, fixed set of base classes, the Visualization Studio can easily construct module icons and GUIs automatically from the C++ module declarations. The modules are compiled into C++ dynamic link libraries and manipulated at run-time via a `Tcl`-based interface. The tcl interface is automatically constructed by parsing the modules' C++ declarations. In this respect, the Visualization Studio-VTK combination offers more automation in integrating user-written modules than all the other systems discussed before. However, development of new modules must obey a complex set of rules, such as single inheritance only, strict naming and module port typing conventions, method overriding requirements, and so on. Moreover, the automatic GUI construction provided by the Visualization Studio is very basic. Only a few widgets are supported, and it is not possible to introduce custom widgets at this moment.

Matlab SimuLink

Matlab is probably the most popular programming environment in the numerical research world [66]. Matlab consists of a kernel interpretor engine for the proprietary language `M`. The `M` language is designed with special provisions for mathematical computations, such as matrix algebra operations, an extensive set of built-in operations and mathematical functions, etc. SimuLink is a graphical environment built over the Matlab kernel in which simulations can be visually assembled, steered, and monitored. In the following, we shall focus on the functionality of the visual programming builder SimuLink.

Primitive units. The primitive units are procedures or functions written in Matlab's proprietary language. Compiled C or FORTRAN code can be imported into Matlab, subject to certain interface limitations (functions must restrict their parameter types to a few system supported types, etc). In this respect, Matlab is a dual language framework similar to Oorange. Matlab components are organised in toolboxes that cover various application domains, such as signal processing, statistics, symbolic mathematics, optimisation, etc. Primitive units can be nested hierarchically to form more complex units, similarly to the other systems previously described in this section. No support for object-orientation, or OO code integration, is provided.

Control mechanism. SimuLink offers a simple event-driven dataflow mechanism, similar to Oorange. Module input and output ports are created by declaring `M` global variables. Modules are connected by specifying how these variables are shared between modules. Loops in networks are supported to build iterative processes.

Data model. Similarly to AVS and Explorer, Matlab offers a few 'system' data types, such as string, matrix, vector, etc. There is no support for user-defined C or C++ types outside this space. Data passing is basically done only by reference, in a similar manner to the Oorange system.

User interfaces. Matlab offers a command-line interface where commands in the `M` language can be issued. SimuLink offers a very primitive network editor. Per-module GUIs have to be built manually by writing `M` GUI scripts over the computational code. Module icons are also created manually, by specifying the drawing commands to display the icon. Only a few widgets are available for GUI construction. It is not possible to add custom widgets to this set. Overall, the SimuLink visual programming environment is very restrained, as compared to all the other environments presented before. Both in execution speed and application domain coverage, Matlab and SimuLink are designed for building and testing small research programs, and not for complex, computational-demanding simulation or visualisation applications.

2.6 Conclusion

In this chapter, we have identified three user roles involved in the SimVis application domain: the end user, the application designer, and the component developer. On the SimVis software system side, three main architectural models were discussed: the application library, the turn-key system, and the dataflow application builder model. From the presented architectures, dataflow application builders satisfy the best our user requirements. A more detailed analysis of instances of this class of systems reveals that no existing system satisfies our user requirements entirely.

We rephrase here our problem statement in design terms: Design a SimVis system that combines the library, application builder, and turnkey architectural models in such a way that the result unites their advantages and minimises their disadvantages. This statement implies that we should not prefer a single architectural model, but try to merge all of them in a single new one. Since the application builder model proved to satisfy the closest the initial requirements, the new architectural model will resemble it the most.

We have seen that each user role answers the question 'what is a SimVis application?' in a different way. For component developers, an application is a set of code fragments and APIs. For application designers, an application is a set of component instances communicating via a kernel that models a specific domain. For end users, an application is the set of user interfaces with which they interact. A SimVis environment may ensure a smooth role transition if it uses *the same application model*, or very similar models, *for all user roles*, rather than different ones. None of the reviewed architectural models provides however such a cross-role application model.

The above observations lead to the conclusion that a new SimVis system must be designed to satisfy our user requirements. This new system would offer the customisability and easy extensibility of application libraries, dynamic application construction from visual components of application builders, and the ease of use via command-line, GUIs, and direct manipulation of turn-key systems. Next, applications in this system should be represented by the same few concepts for all user categories, in order to ensure an easy role transition. We shall achieve this structural unity by using the same object and functional (dataflow) models to describe our application components, application, and end user interfaces. Applications in this new system should be as efficient as compiled applications, but as easy to edit dynamically as interpreted applications. The new system should satisfy our initial problem statement for real-size applications taken from various simulation and visualisation application domains. The design and use of this system is covered in the rest of this thesis.

Chapter 3

Architecture of VISSION

In chapter 2 we have presented the requirements of SimVis users, and outlined several existing SimVis architectural models. We identified the need for a better SimVis system, based on a combination of the existing architectural models and software techniques.

The main problem the designer of such a system faces is the combination of several existing architectures in a single, coherent system. For this, several elements of the library, turn-key system, and dataflow architectures have to be connected in a new architectural model. This chapter presents this new model on which we have built the general-purpose visualisation and simulation environment VISSION. The architectural model presented here leads to a design, implementation, and use of VISSION that complies with the problem statement presented in chapter 1.

3.1 Overview

The acronym VISSION stands for VISualisation and SImulation with Object-oriented Networks. VISSION is a general-purpose SimVis environment that provides a consistent framework for the construction and use of simulation and visualisation applications. VISSION does not target a single, specific SimVis domain such as medical imaging, flow visualisation, or finite element simulations. Instead, VISSION is built to address the requirements of component development, application design and use for a broad class of simulations and visualisations, including, but not limited to the ones mentioned above. In conclusion, VISSION is built to address the problem statement presented in the beginning of chapter 1.

We have shown in chapter 2 that most existing SimVis systems are built on architectural models that do not satisfy entirely our user requirements. In order to remove these limitations, VISSION is built on a different architectural model that combines many aspects of the library, framework, and turn-key models. An overview of the main advantages and disadvantages of the abovementioned architectures will help to deduce the main ingredients of the new architectural model (Fig. 3.1). Out of this overview we shall extract a concrete specification for the new architecture we need to develop, as well as for the software techniques needed to implement it.

As already mentioned, we aim to build a SimVis environment that offers a generic SimVis application model. For this, we choose the *dataflow model*, which describes a simulation or visualisation as a dataflow network of cooperating components. Any network topology, such as acyclic or cyclic, should be supported. We shall use an event-driven kernel model, since this has several advantages over the alternative demand-driven model, as shown in Section 2.5.1. Overall, the proposed SimVis architecture will resemble an application building framework with a dataflow kernel and application-specific

Application Libraries	
Advantages	**To provide them, we need:**
freedom for component development	powerful development OO language(s), preferrably a single one
no concern about context of component use	−
large SimVis library legacy	must be able to reuse binary, not just source code
high performance	compiled libraries
Disadvantages	**To remove them, we need:**
need context to use libraries	combine library and framework architectures
libraries' APIs often low−level and/or incompatible	higher−level component and component library abstraction
Frameworks	
Advantages	**To provide them, we need:**
provide context for components to operate	implement such a context, e.g. by a dataflow model
mechanisms to build applications by assembling components	implement such a mechanism, e.g. a visual builder
provide component−related services automatically	reflection capability on the component interfaces and implement the services (GUIs, etc) upon it
Disadvantages	**To remove them, we need:**
framework context may be too specific	implement generic SimVis application model
two−way component−framework coupling may cause intrusive integration	decouple component interfaces from their implementations
Turn−Key Applications	
Advantages	**To provide them, we need:**
specialized functionality for a narrow domain	ways to customize the generic application model
Disadvantages	**To remove them, we need:**
closed for customization and extension	combine turn−key and framework architectures

Figure 3.1: Advantages, disadvantages, and requirements of SimVis architectural models

components.

Components are grouped in domain-specific libraries. Component libraries are written in a single object-oriented language. This language should be compilable to produce efficient executable code and should also allow reuse of existing SimVis legacy code in source and binary form. Integration of component libraries in the SimVis environment should be non-intrusive. Communication between the SimVis framework and the components should be done via a high-level component interface. This interface should be decoupled from the component implementation, to ensure a non-intrusive, easy integration of components in the framework.

The targeted SimVis environment should provide several services for all user groups, such as transparent integration of new components, easy assembly of components into a final application, and steering and monitoring of the final application. These services should be accessible via the most convenient interfaces, such as visual programming tools and scripting for application design, GUIs and direct manipulation for application steering and monitoring, and command-line tools for all tasks.

Finally, it should be possible to provide the look-and-feel of a turn-key application to the general-purpose SimVis application model implemented by the proposed system. This should enable the production of specialised applications for well-defined, focused application domains and user groups.

This requirement should not be achieved by modifying the general-purpose SimVis kernel, but by providing an open mechanism to customise its user interfaces to support various policies and interaction modes.

The VISSION system was designed and implemented to follow the above specifications. The architecture of VISSION is based on two main elements:

1. the combination of object-orientation and dataflow modelling
2. the visual representation and manipulation of the above combination

This chapter covers the first element. The second element is the objective of chapter 4.

3.2 Combination of Dataflow and OO Modelling

The VISSION architectural model outlined above contains two main elements:

1. the component libraries
2. the framework kernel

We have chosen for *object-oriented component libraries* written in a compilable OO language. This language should allow an easy reuse of legacy SimVis code in source or binary form. One language that satisfies very well the above requirements is the C++ language [102]. Indeed, C++ offers better language mechanisms for OO modelling than other object-oriented languages [21]. Among these, we note:

- a type system supporting multiple and virtual inheritance
- method overloading
- extensive generic programming by use of templates
- a rich choice of data passing mechanisms, i.e. by reference, by value, and by pointer.

Second, numerous studies have shown that well-written C++ compiled code can be almost as fast as e.g. C or FORTRAN compiled code[58, 14]. Third, C++ can easily integrate C and FORTRAN code by means of its external linkage mechanism, and can support practically all the programming constructs of these two languages. This strongly favours the reuse of SimVis legacy code, which comes mainly as C++, C, or FORTRAN sources or binaries.

Secondly, we have chosen a *dataflow application model*. This means that applications are described as directed, possibly cyclic graphs in which the nodes represent processes, actors, and stores, and the arcs are the data and event flows [91]. On one hand, all these elements are dataflow entities, which means they should provide dataflow-related attributes, such as data inputs and outputs, and an update operation. On the other hand, these elements model domain-specific notions, such as a numerical integration process, a visualisation actor, or a 3D vector dataset store. Since the VISSION framework kernel is open and domain independent, these abstractions can not be coded into the kernel itself. Instead, they all have to be specified in the component libraries, i.e. programmed as C++ components that provide the desired simulation or visualisation services.

At this point we can see that the 'component' notion used in the above has to provide two types of services (Fig. 3.2). On one hand, components are dataflow entities, so they have to provide a *dataflow* *interface* that supports services such as data inputs, outputs, and an update operation. On the other hand, components model domain-specific concepts, via *domain-specific interfaces*. These two interfaces have to be combined in a single *kernel-component* (KC) interface that should support all interactions between the kernel and the components.

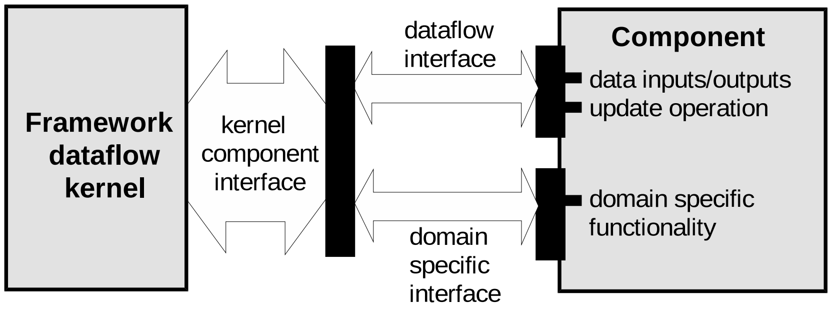

Framework dataflow kernel
kernel component interface
dataflow interface
domain specific interface
Component
data inputs/outputs
update operation
domain specific functionality

Figure 3.2: Kernel-component interface

The combination of the two above interfaces can be realised in practice in two ways. The first method, white-box interfacing , is the most widespread and is described next. The second method, based on meta-programming, is the one we have adopted in the architecture VISSION, and is presented in Section 3.2.2.

3.2.1 White-Box Interfacing

The first possibility to merge the two interfaces is to do it at the development language level. In an OO case, this is usually done by the creation of abstract base classes that declare and partially define the dataflow interface and some other infrastructural mechanisms, but implement no concrete services. Next, concrete subclasses are derived from these base classes. These subclasses implement the dataflow-related operations declared by their bases in specific ways. Subclasses can be refined in turn to provide more specialised operations.

An example of the above is given in Fig. 3.3. A baseclass `Process` implements a minimal dataflow fundament that consists of the declarations of an input and output and the declaration of an update operation. Both the input and output are of type `DataSet`, which is an abstract baseclass for all possible data stores. The `Process` interface consists thus of a method `setInput()` which sets the given input to a given store `DataSet` and of a method `getOutput()` which returns the `DataSet` store into which the process computes its result. All `Process` methods are abstract, since we do not know the concrete store types accessed by `Process`, nor its update operation at this level (Fig. 3.3 a).

The dataflow kernel communicates with the components only via the `Process` interface, as follows. First, the kernel represents a dataflow network as a graph of `Process` objects which are connected by their inputs and outputs. When the application designer desires to connect a component `c1` to the output of a component `c2`, the kernel executes the equivalent of the C++ statement `c1.setInput(c2.getOutput())`. Secondly, components can be created and destroyed by calling the constructors, respectively the destructors of `Process` subclasses. Thirdly, the `Process` interface is used by the dataflow kernel during the traversal of a dataflow network to transfer data from outputs to inputs and to call the update operations of the traversed components (Fig. 3.3 b).

Domain-specific components are derived from the `Process` interface. For example, one could create a `VectorDataFilter` component to model a process that reads a vector dataset, performs

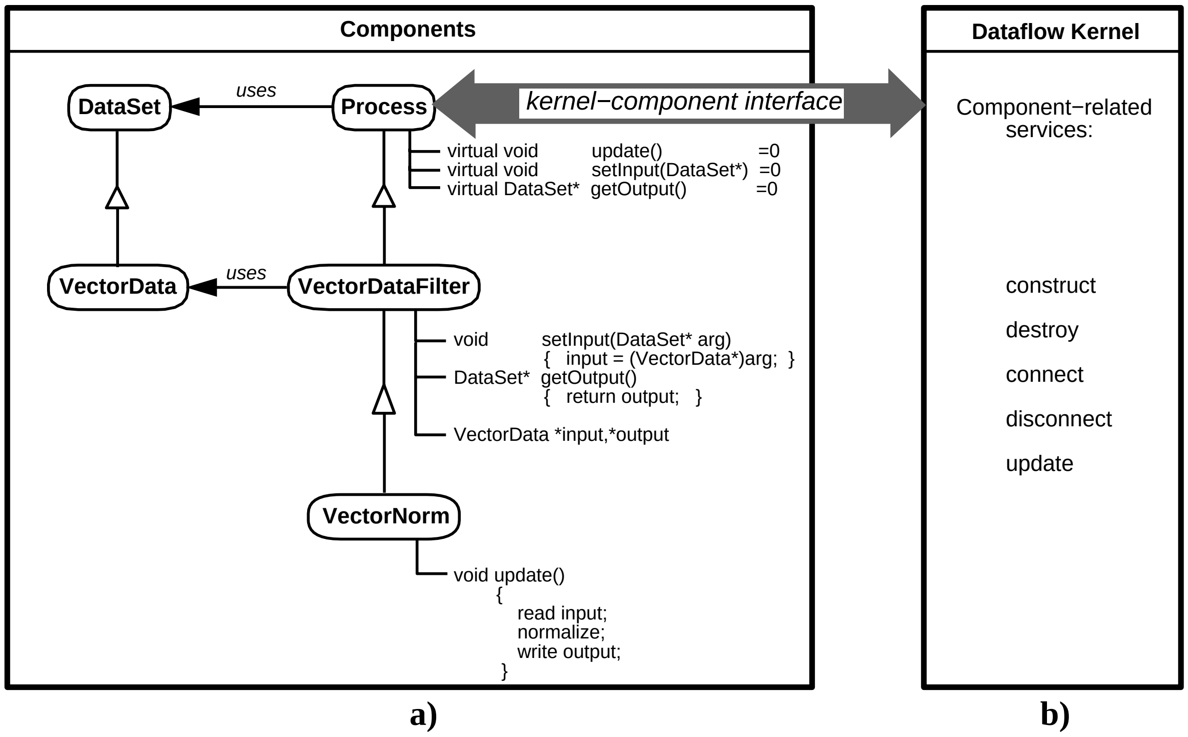


Figure 3.3: White-box dataflow class hierarchy (a) and dataflow kernel (b)

an operation on it, and outputs the resulting dataset. `VectorDataFilter` provides an input and output of the type `VectorData` by implementing the inherited `setInput()` and `getOutput()` methods by using its `VectorData` members `input` and `output`. `VectorData` is thus a concrete subclass of `DataSet` which models a vector dataset, e.g. as a discrete grid with vector values defined in each of its cells. There is a *uses* relationship between `Process` and `DataSet`, as well as between `VectorDataFilter` and `VectorData`.

Finally, a concrete `VectorNorm` could be derived from `VectorDataFilter` to implement the normalisation of a vector field. `VectorDataFilter` implements the inherited update operation as 'read the `VectorData` input, normalise the vector field read, and write the normalised field to the `VectorData` output'. Both the `VectorData` input and output are defined at the `Vector-DataFilter` level, but effectively used at the `VectorNorm` level.

This inheritance-based approach is used by most SimVis frameworks, especially object-oriented ones, such as Open Inventor [116], VTK [94], Diffpack [14], and GDP [82].

There are two fundamental problems with this approach: the intrusive component integration and the restricted dataflow interface. These problems are presented next.

Intrusive Component Design

The main problem is that the above approach of attaching a dataflow interface to domain-specific components actually represents a white-box framework solution. As outlined in Section 2.4.3, white-box frameworks use a two-way, language-level coupling between the components and the framework kernel. For example, in order to create a new component, one has to inherit from the abstract dataflow interface known by the kernel. This technique is convenient when both the framework and its application libraries are designed from scratch.

However, if existing application libraries have to be integrated in an existing dataflow framework, the components of these libraries must be forced to inherit from the framework's dataflow interface. If the libraries cannot be modified e.g. due to copyright issues or if they are not available in source form, the above technique can not be applied. Moreover, even in the case one can and is willing to modify existing components to integrate them in a given dataflow framework, this modification can be impracticable. For example, to integrate a library that contains the class hierarchy in Fig. 3.4 a, one should modify its roots to obtain the hierarchy in Fig. 3.4 b. This implies the modification of all classes in the hierarchy to adapt them to the dataflow interface, as well as the appearance of virtual inheritance at the top of the hierarchy. The above technique, known as the Class Adapter design pattern [37], is appropriate only when adapting single classes or very simple class hierarchies. When adapting large, multiple-inheritance based hierarchies, the Class Adapter generates extremely tightly coupled inheritance lattices which soon become unmanageable [63].

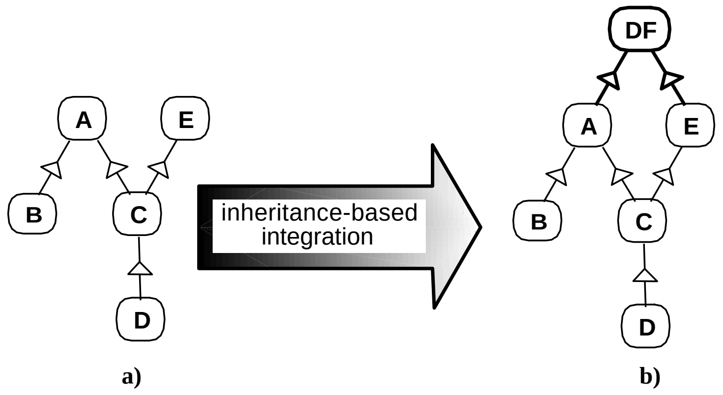


Figure 3.4: Inheritance-based integration. Classes A and E have to inherit from dataflow interface DF, which causes virtual inheritance

A more flexible way of attaching a dataflow interface to a set of existing components is provided by the Object Adapter design pattern [37]. This implies practically the creation of an 'adapter' or 'wrapper' class for every component to be adapted, or adaptee. The adapter-adaptee communication is done via a *uses*, and not via an *is a* relationship, as in the case of Class Adapters (Fig. 3.5). Adapters inherit from the dataflow interface and delegate all the requests they receive to their adaptees. Adaptees are created and destroyed by their adapters at their creation, respectively destruction time. Object Adapters are better than Class Adapters since they do not impose the modification of the adaptee code. However, the appearance of complex class lattices still remains. Moreover, writing Class Adapter hierarchies for large component libraries is a time-consuming task for the component developer. For every adapted component, an adapter class has to be written which calls back on the adapted component's interface.

Restricted Dataflow Interface

While the first problem of inheritance-based component integration relates to component development, the second problem regards the framework kernel itself. As described above, the inheritance-based integration is based on a KC interface which is implemented by the components and required by the kernel. This interface is hard-coded in the kernel and represents the gateway through which this one communicates with the components. The main elements of this interface are the components' data inputs and outputs.

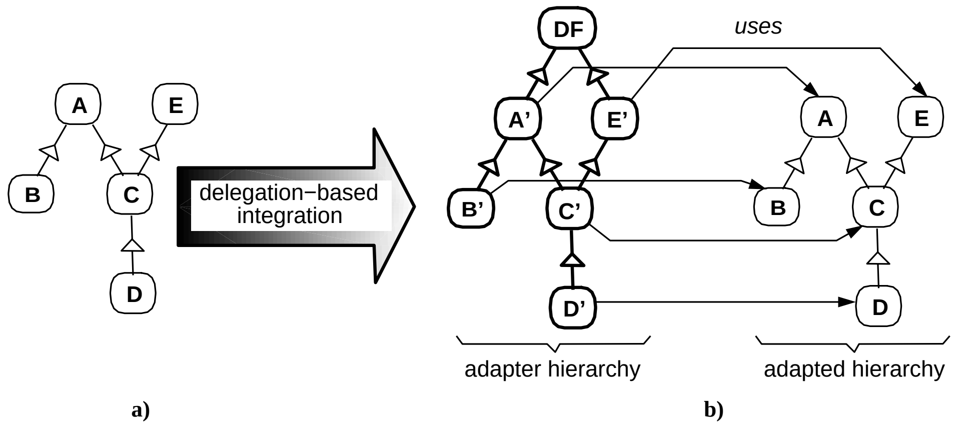


Figure 3.5: Delegation-based integration. Class hierarchy in (a) is adapted in (b) in order to conform to the dataflow interface `DF`

In the vast majority of cases, dataflow interfaces use typed data inputs and outputs. This has several advantages. First, the dataflow kernel can check whether compatible inputs and outputs are connected by the application designer by checking their type compatibility and forbid incompatible connections. This ensures that constructed applications are always correct from a typing point of view. This mechanism is similar to the type checking phase of a compiler, but is usually done at run-time, when the application designer assembles the components together. For this, the KC interface should implement some form of run-time type information (RTTI) [102, 21].

Since the KC interface is hard-coded in the kernel, the choice of input and output data types is obviously limited to the types which are hard-coded in the kernel together with this interface. This restricts the development to components that communicate via the fixed set of data types supported by the kernel. This set usually contains fundamental types such as integer, float, double, boolean, and character string. Usually SimVis systems based on the above architecture provide also some structured data types such as vectors, matrices, and some data sets such as regular, irregular, and unstructured grids, with several cell types. This is the case for systems such as AVS/Express, Iris Explorer, or VTK.

A possible option is to leave the dataflow kernel open for modification, such that component developers could extend the KC interface with new types. However, this is a dangerous option, as developers could modify the existing interface, besides extending it. This could render previously functional components inoperational. Moreover, modifying an interface which is hard-coded in the kernel implies modifying all kernel subsystems that use that interface. Overall, this option is not viable and actually not used in any of the SimVis systems we know of. For an extensive overview of the negative implications of modifying a base interface, see the work of see Meyer [69] and Martin [64] on the Open-Closed Principle, and Martin [65] and Liskov [59] on the Substitution Principle.

The development of new components and integration of existing ones has either to directly use the data types supported by the kernel or to map the data inputs and outputs to and from these data types. In both cases, this leads to a complex component development and integration path. The resulting code is either system-dependent, as it contains explicit references to the kernel-specific structures, or heavily adapted by massive adapters that perform the data translation. In the latter case, data has to be converted by value between the component's and the kernel's representation, which may result in severe performance loss. If the components communicate via encapsulated data types, such as C++ classes, it might not be possible to directly access their internal representation in order to convert these to the kernel data representation.

3.2.2 Metalanguage-Level Integration

The second method of integrating the dataflow and domain-specific functionality is based on the observation that inheritance from an interface is too rigid to cover the communication with an open set of components from an open set of application domains. Consequently, a more flexible KC interface is needed.

The KC interface is the only mediator between the dataflow kernel and the object-oriented components. This communication consists primarily of data typed inputs and outputs which should, on one hand, map directly to the component data interface, and on the other hand, be directly accessible to the kernel. Via this interface, the kernel should be able to read and write any input or output data value. In order to accomplish this, the kernel should be able to:

1. have a direct representation of all possible value types supported by any component interface.

2. ask the component to update, or recompute its state, as a response to a change of its inputs. On the component side, the update operation should be able to check which inputs have been modified, then perform the desired actions on the component, and finally inform the kernel about which outputs have been changed as a result of the update.

3. instantiate and destroy the components at any time. As components are C++ classes, this operation should cause the usual C++ construction and destruction actions on the component side.

4. load and unload several component libraries. By loading a library, the component types of that library should become accessible to the kernel. Unloading represents the inverse operation through which the kernel relinquishes the component types of a given library.

In order to accomplish all the above operations, the KC interface needs more freedom than a C++ class interface hard-coded in the kernel. Services such as dynamic component type loading and unloading, dynamic component instantiation , and reflection require mechanisms of a higher level than the ones provided by the classical compiled model of the C++ language. Many of these mechanisms are provided by default for programming languages such as Java [23] or Smalltalk [40] by their run-time environments (sometimes also called virtual machines). However, even though similar systems exist for C++ as well, they do not provide all the services needed to implement the KC interface.

The solution of choice is to provide these services at a *meta-level*, i.e. in a framework that is above the C++ language. Such a framework can be implemented e.g. as a meta-language that extends the semantics of C++ with the notions needed by the KC interface, such as dataflow features, dynamic component type loading and component instantiation, reflection services, etc. The notion of a SimVis component, so far only partially supported at the C++ language level, will be clearly defined at the meta-language level, together with its interface to communicate with the kernel.

We have chosen the above solution in the architecture of VISSION by creating a simple meta-language called MC++ . MC++ is a language layer over C++ which adds dataflow semantics to object orientation in a non intrusive manner. MC++ offers a few meta-constructs that allow component developers to 'enrich' plain C++ constructs such as classes and types with dataflow functionality. There are four meta-constructs in MC++:

1. **Metaclasses:** Metaclasses extend C++ classes to form the basic components for building SimVis applications.

2. **Meta-types:** Meta-types extend C/C++ types to form the data passing interface between metaclasses.

3. **Meta-groups:** Meta-groups model composition of metaclasses into nested hierarchies.
4. **Meta-libraries:** Meta-libraries package together the metaclasses, meta-types, and meta-groups for a given SimVis application domain.

These constructs are presented in the following sections.

3.3 The Metaclass

The metaclass is the main construct in MC++. A metaclass models the type of a SimVis component. The concrete components from which a simulation or visualisation is built can be seen as instances of metaclasses, just like C++ objects are instances of C++ classes. Functionally, metaclasses extend the C++ class concept with various dataflow related information. Structurally, metaclasses can be seen as metalanguage-level wrappers around plain C++ classes that make these classes compatible with the KC interface. Similarly to C++ classes that have an interface (the `public` part) and an implementation (the `private` part), metaclasses are component interfaces whose implementations are the wrapped C++ classes. In comparison with the white-box component-kernel integration illustrated in Fig. 3.3, metaclasses allow a black-box component integration (Fig. 3.6) . The KC interface is now described above the C++ component implementation level, that is at the metaclass level. This allows the decoupling of the component interface from its implementation demanded in the previous chapter.

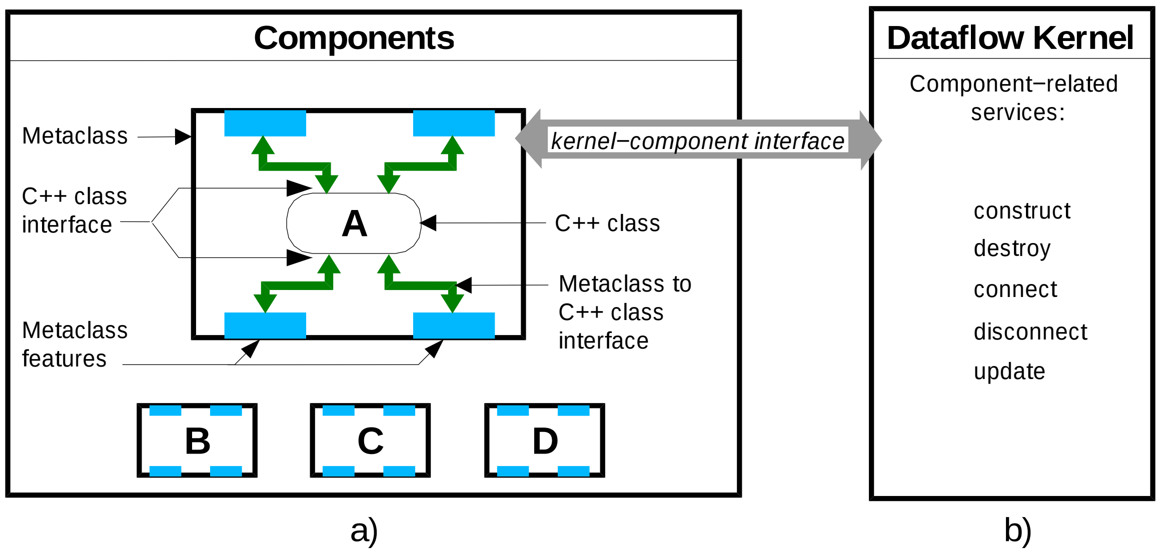


Figure 3.6: Metalanguage kernel-component interface

A metaclass is a collection of *features* , just as C++ classes are collections of members . The most important metaclass features are the input and output ports and the update operation. These features are described next. The other features will be discussed later.

3.3.1 Metaclass Ports Overview

A *port* is a metaclass feature through which a component can read or write data from, respectively to other components. A dataflow application is thus represented as a directed graph whose nodes are metaclass instances and arcs are port-to-port connections. The port-to-port connections are called links in MC++. A port has three main attributes: type, input/output, and read/write. These attributes determine whether two ports can be connected by a link, and, in the positive case, how data is transferred across the link. Port attributes can vary independently, so one may have input read, output read, input write, and output write ports, of different C++ types.

In most existing dataflow systems, ports have only the type and input/output attributes. However, these systems have difficulties supporting the several data passing mechanisms provided by programming languages, such as by-value, by-reference, by-pointer, and reference counted data passing. Small datasets are usually passed by value. In the case of large datasets, the ownership of the data (i.e. which modules create, respectively destroy the datasets) is usually provided by reference counted mechanisms hard-coded in the system kernel. If one desires to support user-defined, kernel-independent data types, this solution is however not possible. The addition of the read/write port attribute solves the above problem, as it will be shown in section 3.3.4. In this section, an overview of the port attributes is given.

The *type* of a port represents the type of values that can be read or written from, respectively to that port. Since metaclasses represent only component interfaces to C++ components, port types are native C++ types, such as integer, float, pointer, or class types, passed by value or by reference. The semantics of data passing is identical to the C++ one. For example, a port that has a reference C++ type such as `int*` or `float&` is going to read or write just a reference to a value, not the value itself. A port that has a value C++ type such as `double` or `DataSet` (where `DataSet` is some C++ class type) will read or write a copy of the passed data value, not the passed value itself. If the port is of a class type that has a copy constructor, this will be invoked at the data transfer just as it is done during an usual C++ value transfer.

The *input/output* port attribute represents the *conceptual* direction in which data flows related to the port's metaclass. As previously mentioned, a dataflow application is a *directed* graph whose nodes are the metaclasses and arcs are port-to-port connections, or links. The port input/output attribute determines the direction of links that may end at that port. Links are thought to emerge from output ports (shortly outputs) and enter into input ports (shortly inputs). The sense of the links determines thus the direction in which the dataflow network is traversed and consequently the order in which its components compute, or update. Figure 3.7 a shows the graphical representation of a metaclass. The upper and lower rectangles represent the input and output ports respectively. This representation is similar to the one used by most dataflow systems that depict components visually.

In most cases, the data transfer that actually takes places follows the links' directions, i.e. data is read from outputs and written to the inputs. However, as we shall soon see, this is not always the desired situation.

The *read/write* port attribute represents the *actual* direction in which data is transferred related to the port's metaclass. A *read* port is thus a port *from* which data can be read. A *write* port is a port *into* which data can be written. We distinguish two kinds of write ports: read-write or write-only. A read-write port is a write port which can also be read. Conceptually, the value read from a read-write port is the last value written into it. A write-only port is a port which can be only written, but not read. Since read-write ports are used in most concrete cases, we shall call them shortly write ports and use the read-write and write-only terminology only when the context requires this distinction. Ports are usually output and read, respectively input and write. However, input read and output write ports may exist as well, as mentioned above.

Two ports can be connected by a link if all the following hold:

- the port C++ types are compatible, in the sense of C++ type compatibility.

- one port is **input**, the other one is **output**.

- one port is **read**, the other one is **write**.

Data is transferred across a link by being read from the link's read port and written into the link's write port. This is why a read-write port pair is needed to make a link. After a metaclass updates, it will transfer data across all the links connected to its outputs. The metaclasses whose inputs are connected to these outputs will update as well, and so on. This is where the distinction between inputs and outputs comes into play.

Figure 3.7 b shows two connected metaclasses A and B. Data read from A's output **out** flows across the link and is written into B's input **inp**. This is the most common situation, where data 'naturally' flows along the link's direction, from outputs to inputs. Most dataflow systems such as AVS, Inventor, Oorange, VTK, Khoros, IRIS Explorer support only this case, where the sense of the traversal of the dataflow network coincides with the sense of the data transfer. This is indeed natural for the case when data is transferred by value.

However, suppose that A wants to directly modify the contents of B, e.g. by writing B's data members or calling its methods. In this case, A will hold a reference or pointer to B which is set when A and B are connected. In order to do this, a reference to B should be written into A. This is why A's **out** port is of write type and B's port **inp** is of read type. When **A** updates, it *directly* modifies **B** via its stored reference. This is why A's **out** port is of output type, and B's port **inp** is of input type. Figure 3.7 c shows the above example.

The separation of the input/output and read/write notions corresponds actually with two different flows in the network, i.e. the data and control flow. The *data* flow represents the way data is read and written between components, and corresponds to the read/write port attribute. The *control* flow represents the order in which components are updated during network traversal, and corresponds to the input/output attribute. The two senses coincide when data is transferred by value. However, as we have seen, direct modification of components via references implies a data transfer sense opposite to the network traversal sense. This issue is further detailed in section 3.3.4).

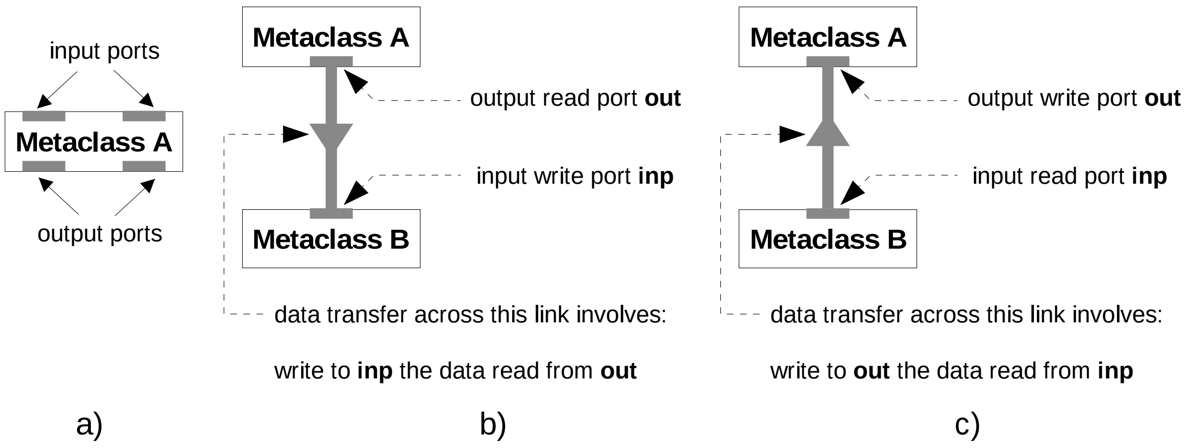

input ports
Metaclass A
output ports
Metaclass A
output read port **out**
input write port **inp**
Metaclass B
data transfer across this link involves:
write to **inp** the data read from **out**
Metaclass A
output write port **out**
input read port **inp**
Metaclass B
data transfer across this link involves:
write to **out** the data read from **inp**
a)
b)
c)

Figure 3.7: Metaclass representation (a). Data transfer from a read output to a write input (b) and from a read input to a write output (c)

When a port is read or written, the actual operation of reading or writing data is delegated by the metaclass ports to its C++ class implementation. To illustrate the above, an example follows showing how a metaclass is created out of a C++ class.

Metaclass Example

Suppose one wants to create a metaclass for a C++ class `PolyArea` (Fig. 3.8). `PolyArea` computes the area of a regular n-sided polygon, being given the polygon's edge length and the number of edges. For this, `PolyArea`'s public interface has an integer data member `edges` to specify the number of edges, and a method-pair `setSize,getSize` to set, respectively get the edge size as a float value. The polygon's area is computed by the method `computeArea()` and stored in the private member `area`. To query this computation, a method `getArea` is provided. The class provides also a constructor to initialise its state, i.e. the data members `size`,`area`, and `edges` to the zero default value.

```
class PolyArea
      {public:
                PolyArea(): edges(0),size(0),area(0) {}
         int    edges;
         void   setSize(float s) { size = s; }
         float  getSize()        { return size; }
         double getArea()        { return area; }
         void   computeArea()    { compute area as function
                                   of size and edges }
       private:

         float  size;
         double area;
      };
```

Figure 3.8: Simple C++ class to compute the area of a regular, n-sided polygon

C++ does not specify which of a class' members are 'inputs' and which are 'outputs', as it does not have the dataflow notion. However, in order to make a dataflow component out of `PolyArea`, we must identify which are the conceptual inputs and outputs through which this class communicates with the outside world. In the above case, both `edges` and the `setSize,getSize` method pair control the class' data inputs, as they provide reading or writing of the parameters on which the polygon's area depends. `PolyArea` has only one output, namely `getArea` which returns the area of the specified polygon.

Figure 3.9 shows the MC++ code that declares a `PolyArea` metaclass with two write input ports "number of edges" and "edge size" and one read output port "area". The MC++ code for `PolyArea`

```
module PolyArea
      {input:

        WRPortMem  "number of edges" "int"    (edges)
        WRPortMeth "edge size"       "float"  (setSize,getSize)

       output:

        RDPortMeth "area"            "double" (getArea)
      }
```

Figure 3.9: Metaclass declaration for the C++ class `PolyArea`

illustrates a couple of important MC++ constructs. First, a metaclass is a declaration similar to a C++ class or structure declaration. It starts with the `module` keyword, followed by the name of its implementation C++ class, and the metaclass body between braces. The metaclass body contains the declarations of the metaclass features. In the above example, two input ports and one output port were declared. Port declarations fall into an `input` or `output` access category, specified by the same keywords. All port declarations following an `input` specifier up to the metaclass body end or to a subsequent access specifier are input ports. The same holds for the `output` access specifier. Port declarations coming before any specifier are assumed by default to be input declarations.

3.3.2 Port Specification

To explain the complete `PolyArea` metaclass declaration, we shall first present the port declaration syntax. A port declaration has the following syntax (roughly similar to a C++ method declaration), where:

<*port_kind*> "<*port_name*>" "<*port_type*>" (<*C++ args*>)

- *port_kind* is a special identifier denoting the *kind* of the port whose description follows. The port kind specified which actions are to be taken when data is read or written from that port, relatively to the port's metaclass C++ implementation. Several port kinds are implemented in MC++ to support the most frequently encountered data transfer situations. The `PolyArea` metaclass illustrates the three most frequently used port kinds: `WRPortMem`, `RDPortMeth`, and `WR-PortMeth`. `WRPortMem` denotes a read-write port whose read and write actions correspond to reading, respectively writing a C++ class member. `RDPortMeth` denotes a read port whose read action corresponds to calling a C++ class method and using its return value. `WRPortMeth` denotes a write-only or read-write port whose read and write actions correspond to calling two methods of the form `T get()`, respectively `set(T)`, where `T` is the port C++ type. Other port kinds will be presented later.

- *port_name* is the name of the port in MC++, given as a string enclosed between double quotes. The port name serves to identify a given port in the KC interface. The port name does not have to be related with the names of the C++ class members that implement it, as noticeable from the `PolyArea` example above. Input ports among themselves, respectively output ports, must have different names, similarly to C++ class members. However, an input may have the same name as an output, since inputs and outputs are two different name spaces in MC++. This is desirable e.g. when both an input and output port refer to the same data element, such as the ”value” of an `Integer` module. Another example is the ”this” port, discussed in section 3.3.4.

- *port_type* is the port's C++ type, i.e. the type of the data value the port reads or writes. The port type is specified as a string and can be any valid C++ type specifier, such as ”int”, but also ”const char*”, ”const MyClass&”, etc.

- *C++ args* is a list of C++ class members, separated by quotes and enclosed in brackets. These are the names of the data members and/or methods used by the port to implement its read/write functionality. The number of the arguments, as well as their meaning, depend on the specific port kind. In the above example, the port ”number of edges” of kind `WRPortMem` needs just the name of the C++ data member it reads and writes, namely `edges`. Similarly, the port ”area” of kind `RDPortMeth` needs the name of the C++ method `getArea` it calls to get its value. The port ”edge size” of kind `WRPortMeth` needs two methods `setSize,getSize` to set, respectively get its value from the underlying C++ class.

We have so far presented how the metaclass construct adds the dataflow notions of inputs and outputs to a plain C++ class. To understand how the dataflow mechanism actually works, an example is presented where two `PolyArea` metaclasses are coupled to form a simple dataflow network.

Network Example

A simple dataflow network is built by connecting two `PolyArea` components that, for the sake of this example, will be called `comp1` and `comp2` (Fig 3.10). A link is established between the ”area” output of `comp1` and the ”edge size” input of `comp2`. The link is possible, since ”area” is a read output, ”edge size” is a write input, and ”area”’s double type is compatible with ”edge size”’s float type. Two operations take place when data flows across the link between the ”area” and the ”edge size” ports. First, the dataflow kernel reads the value of the ”area” output. This is done by calling the method `getArea` on `comp1`’s C++ object. Next, this value is converted to a float, and then copied into the

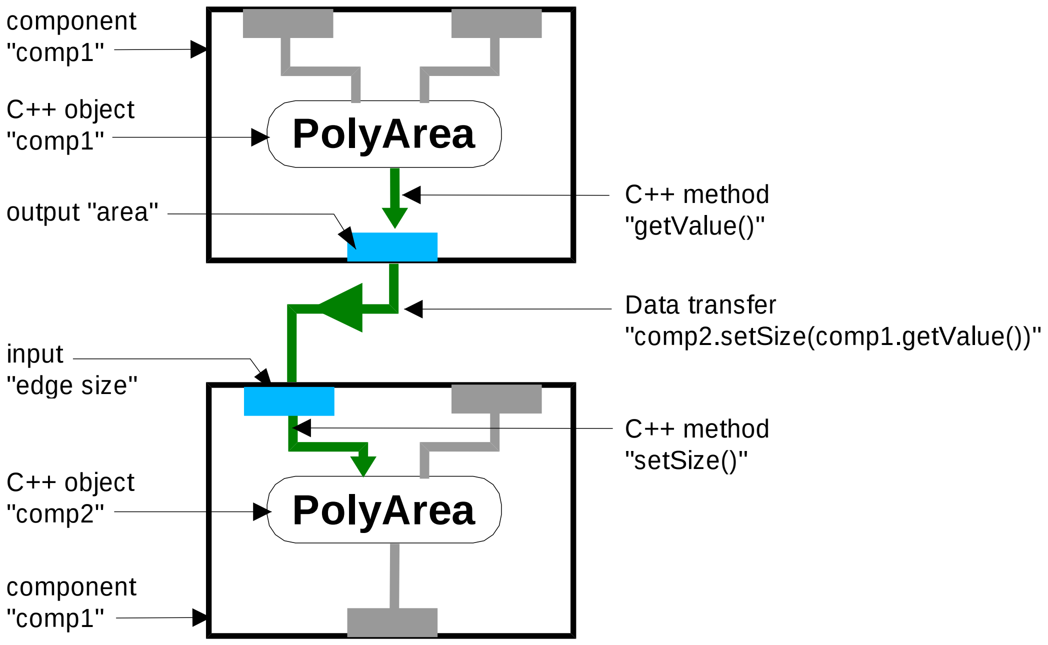


Figure 3.10: Data transfer between two metaclass components

”edge size” input. This is done by calling the method `setSize` on on `comp2`’s C++ object with the value read in the first step. Overall, the data transfer is conceptually equivalent to the execution of the C++ statement:

```
comp2.setSize(comp1.getArea());
```

Similarly, if we connected `comp1`’s output ”area” to `comp2`’s input ”number of edges”, the data transfer across the emerging link would be conceptually equivalent to the C++ statement:

```
comp2.edges = comp1.getArea();
```

In the above examples, we assumed that there would exist two C++ objects called `comp1` and `comp2` which would correspond to the metaclasses with the same names.

3.3.3 Metaclass Update Operation

We have presented so far how data is transferred between two components. Transferring data is however only half of the job done by the dataflow kernel when a component network gets traversed. The other half consists of the components' update operation. The `PolyArea` component presented above would not function correctly if, whenever an input is changed, its `computeArea` method were not called to recompute the polygon area and thus bring its "area" output up to date. Since we use an event-driven kernel model, the above operation should happen automatically whenever a component input changes (see Section 2.2).

MC++ provides exactly this functionality by its metaclass *update operation*, the second most important feature after the metaclass ports. The update operation specifies all actions to be done by the kernel on a metaclass in order to update it when its inputs change. If a metaclass has no update operation, nothing is executed to bring its state up to date when its inputs change. The dataflow kernel will assume that *all* its outputs have changed (even though nothing was really done to change them) and thus will continue the network traversal and component update from them on. This allows the insertion of 'update-less' components in a network, such as a data store between two processes, one writing it and the other reading it. These components are transparently traversed by the dataflow mechanism without the need for special mechanisms to treat them.

If, however, a metaclass needs to execute some action to recompute its outputs when its inputs change, an update operation needs to be specified. The simplest way in which this can be done in MC++ is to delegate it to a method of the C++ implementation class, similarly to the delegation of the port read and write actions. For example, to delegate the update of the `PolyArea` metaclass to the C++ method `computeArea`, the name of the C++ method is added between braces after an `update` specifier in the metaclass declaration. A metaclass may have a single update operation, or none. The above is called the *short form* of the update specification. Fig. 3.11 shows the new `PolyArea` metaclass with the added text in bold. MC++ expects the update C++ method to conform to a `void ()()` or `int`

```
module PolyArea
       {input:

         WRPortMem  "number of edges" "int"    (edges)
         WRPortMeth "edge size"       "float"  (setSize,getSize)

        output:

         RDPortMeth "area"            "double" (getArea)

        update:  { computeArea }
       }
```

Figure 3.11: Adding an update operation to a metaclass

`()()` signature. If the method has a void return type, it is assumed that the metaclass update always succeeds and that all the outputs are modified after the update. If the method returns an integer, a non-zero value specifies that the update operation succeeded, so all outputs are modified after the update. A zero value specifies that the update failed, in which case no output is modified and the network traversal stops at the respective metaclass. This offers component designers a very simple mechanism to steer the control flow from within the C++ components. Similar mechanisms are implemented by other dataflow systems such as AVS, AVS/Express, and VTK.

The short update form was used for most (over 95%) of the MC++ metaclasses we designed for various application domains. There are however cases when one a) must execute different update operations that depend on which input ports have changed; or b) wishes to mark only specific output as changed after the update operation, depending on what this does. These cases require a more detailed specification of the update operation. Specifically, we need a way to determine which inputs are modified at the update call time, and a way to selectively mark the outputs as modified after the update call completes.

For the above, MC++ allows the specification of the update operation directly as C++ source code in the metaclasses. Within this source code, mechanisms are provided to:

- check which input ports have changed to cause the update call
- execute C++ code to perform the effective update computations
- selectively mark the outputs as changed or not changed

The above are best illustrated by an example. Suppose `PolyArea` desires to implement a consistency check, as follows. If the ”number of edges” input is smaller than three, then no polygon can be formed, so the metaclass should fail to update and should not modify its ”area” output. In this case, no downstream component should execute. If the ”number of edges” input is greater or equal to three, `PolyArea` should update as usually, by calling `computeArea` and marking its ”area” output as changed. The above update scenario is specified in MC++ by the code shown in bold in Fig. 3.12, and is called the *extended update form*, in contrast with the short update form already presented.

```
module PolyArea
       {input:

         WRPortMem  "number of edges" "int"    (edges)
         WRPortMeth "edge size"       "float"  (setSize,getSize)

        output:

         RDPortMeth "area"            "double" (getArea)

        update:  int update(PolyArea& pa)
                 {
                     if (SYSTEM::isChanged("number of edges")
                         && pa.edges < 3)
                        {
                           SYSTEM::unchange("area");
                           return 0;
                        }
                     computeArea();
                     return 1;
                 }
       }
```

Figure 3.12: Extended form of update operation

The extended update form consists of the `update` specifier followed by a C++ function definition. This function should comply with the signature `int ()(T&)`, where `T` is the name of the metaclass and of its C++ implementation class as well. This function is called whenever a component of type `T` has to be updated. The argument of the function is a reference to the component's C++ implementation object. In the example in Fig. 3.12, this is a reference to a `PolyArea` object. The update function's body may contain any valid C++ statements that may perform operations on the C++ implementation object. The changed inputs can be retrieved by calling the method `isChanged` of a special class `SYSTEM`. The method `SYSTEM::isChanged` gets an input port name and returns 1 if that input is changed, else 0. As described previously, MC++ assumes that all outputs of a component are changed after this successfully updates. To mark some outputs as unchanged, the `SYSTEM::unchange` method can be used, with the output port name to mark as unchanged as argument. Finally, the update function should return 1 in case the update succeeded, or 0 in the case it failed. The extended update form allow component developers to easily write any kind of update operation for their metaclasses. As noticeable from the previous example, no modification of the C++ implementation class is needed. All the code that couples this class with the KC interface resides in the metaclass declaration.

Figure 3.13 completes the overview of the kernel-component interface based on the metaclass construct that was sketched in Fig. 3.6 at the beginning of Section 3.3. The most important element to be

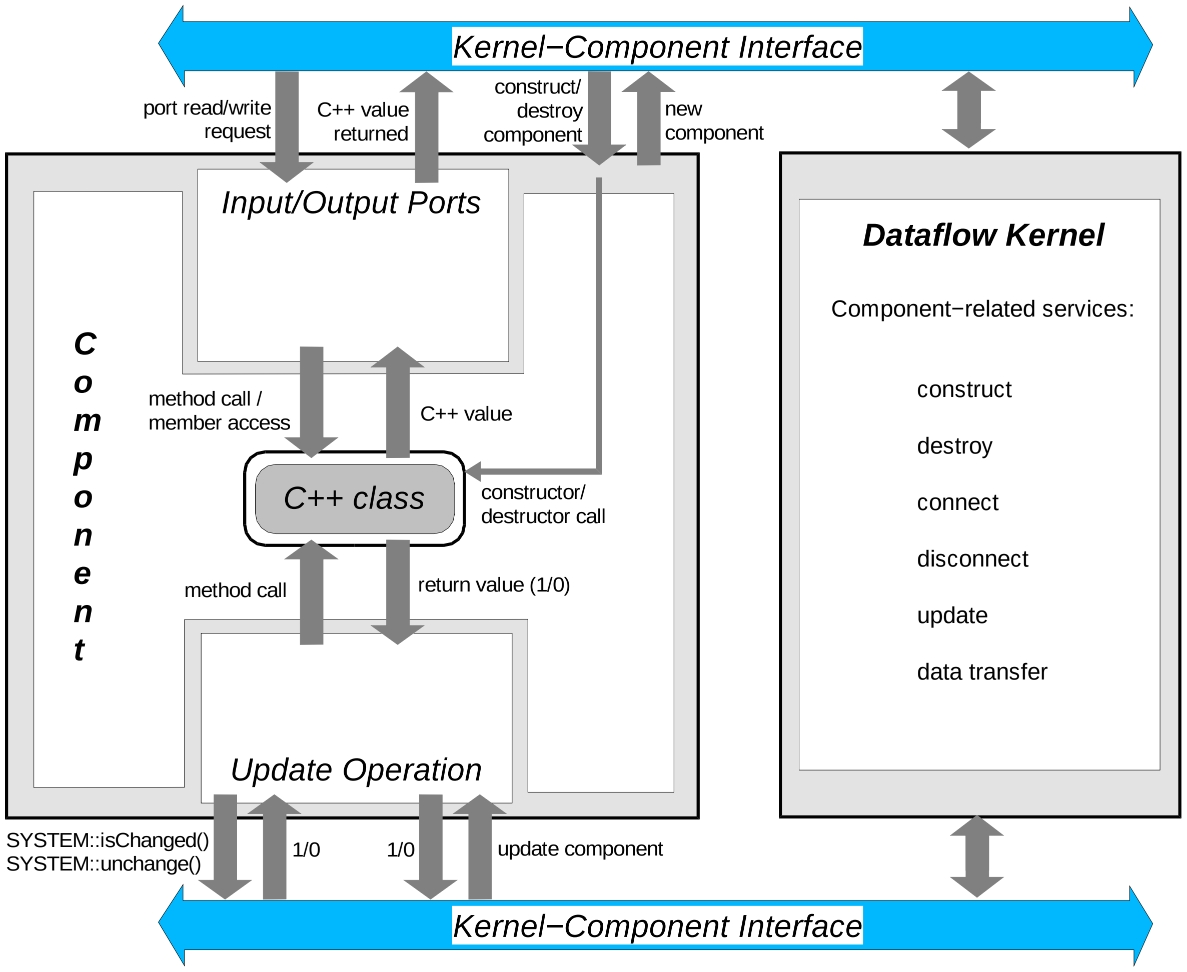


Figure 3.13: Complete picture of the kernel-component interface

noticed is the *total* decoupling of the component's actual code written in C++ and its dataflow functionality provided as a metaclass shell. All the interaction between the two takes place strictly via the

C++ class public interface. The C++ component development is thus in no way constrained. Provided with an appropriate metaclass written in MC++, any C++ components can be integrated in the dataflow framework. The presented framework is indeed a black-box one, as there is no direct connection between the framework kernel and the C++ components. It is clear now why this solution differs fundamentally from the usual white-box based SimVis architectures. Indeed, such architectures have no meta-level that decouples the component design from the framework kernel. Consequently, components must either inherit from a kernel interface (VTK, Open Inventor, Oorange, Diffpack) or contain in their code special statements for data communication and synchronisation with the kernel (AVS, CSE, VASE, SCIRun).

3.3.4 More About Metaclass Ports

In this section, the picture of the metaclass ports is finalised by providing the remaining details on their functionality.

By Value / By Reference Data Transfer

In the `PolyArea` network example presented previously, data was transferred between modules by copying the values delivered by the read ports and writing them into the write ports. There are however cases when we need to pass data by reference between component ports, for example in order to maximise performance by avoiding the overhead of copying large datasets. This can be only partially done with the port mechanisms described so far.

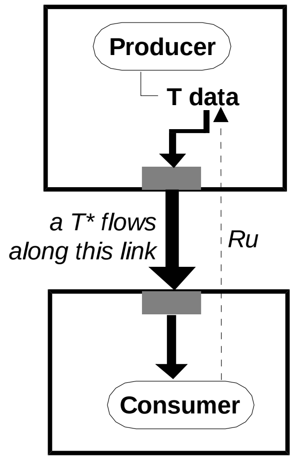
Assume, for example, that we have a component `Producer` which computes some dataset `T`, and a component `Consumer` which needs to read this dataset. The `Producer-Consumer` coupling can be modelled by a `Producer` output port of type `T` and a `Consumer` input port of the same type `T`. Data flows along the `Producer-Consumer` link by passing a `T` object by value between the two ports. This involves the call of `T`'s copy constructor and destructor, if `T` is a class type that provides them. In any case, the transfer of `T` involves a full copy of `T`'s data values. If the `Consumer` does only read the incoming data and the `T` data objects are large, it is faster to pass them by reference, i.e. pass a reference or a pointer to `T` from `Producer` to `Consumer` instead of the whole object `T`. The simplest way to do this is to declare the involved ports of type `T*` instead of `T`. Figure 3.14 shows such an example, where `Producer` outputs a pointer to a `has` member of itself of type `T`, to be used by `Consumer`. A *Ru* relationship exists between `Consumer` and the `T` member of `Producer`.

The above implementation of by reference data transfer is however unsafe. For instance, if the `Producer` component is destroyed after it has been connected to one or several `Consumer` components, these components will have dangling references to the destroyed `Producer`'s `T` member. Since components can be created and destroyed in any order, we need a mechanism to ensure the proper creation, operation, and destruction of reference *Ru* and *Wu* relationships between components.

There are basically two ways to solve this problem. The first is to assume that the components themselves manage the reference relationships between themselves. For example, the `T` objects could be reference counted. The `Producer` would then create dynamically a `T` with a reference count of 1 and manage it via a pointer instead of owning it as a class member. The `T` object would be output as a `T*` as before. When a `Producer` is destroyed, it will not destroy the `T` it created, but decrement its reference count. When a `Consumer` connects to a `T*`, it will increment the reference count of the connected object `T`. When a `Consumer` disconnects from a `T*`, it will decrement the reference count of the connected object `T`. The object `T` is destroyed when its reference count is set to zero, i.e. when there is no component that read-uses or write-uses it any more. This mechanism is implemented

```
class  Producer
{ public:

     T* getData() { return data; }

  private:

     T data;
};

module Producer
{ output:

     RDPortMeth "data" "T*" (getData)
}
```

a)



b)

Figure 3.14: Simple by reference data transfer. C++ class and metaclass (a) and actual data transfer scheme (b)

by Open Inventor and VTK, among others. Several problems occur when the components manage the reference relationships. First, all components that plan to read or write-use a `T` have to be written specially to use such a mechanism, or to be modified accordingly if they have been developed before the mechanism existed. As explained in the previous chapter, this is difficult or even not feasible for large or binary-form third party component libraries. Second, the above solution works as long as the component developers remember to perform the dataset referencing and dereferencing in all their components. This is tedious and error-prone when developing large component libraries. Third, managing the reference relationships in the components code makes the latter less readable.

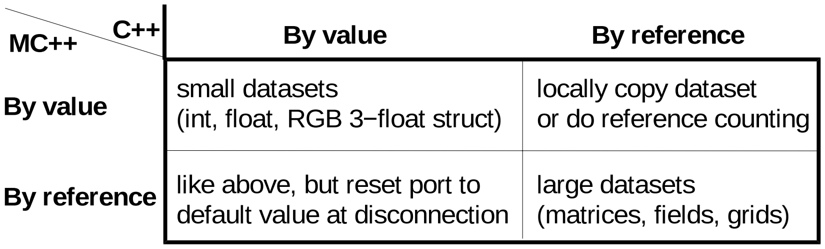
The second solution is to have the dataflow kernel provide support for the management of by reference data transfer. This is the alternative we have chosen by introducing the notion of *reference ports* in MC++. Reference ports behave similarly to the value ports we used so far, with the difference that, when a reference write port gets disconnected, it is automatically set to the *default value* of its C++ data type. Default C++ values are explained in detail in Section 3.4. For the time being it is enough to know that the default value of any C++ pointer type is the `NULL` pointer. For the example in Fig. 3.14, this means that when the `Producer`-`Consumer` link is broken, the `Consumer`'s input port is automatically set to `NULL`. The `Consumer` update code can then easily check the value of this port and prohibit any update operations if it is `NULL`. Moreover, now the `Producer` can output a pointer to a *has* data member safely. Indeed, when the `Producer` is destroyed, all links between its output and other `Consumer`s are automatically destroyed by the dataflow kernel, which causes setting all `Consumer` input ports to `NULL`. The component developer does not need to do anything in the C++ `Consumer` or `Producer` code. All he needs to do is to declare the `Consumer`'s input port and the `Producer`'s output port as reference ports. This is done in MC++ by appending an asterisk after the port's kind, similarly to a C++ pointer declaration (Fig. 3.15).

The reference concept of MC++ and the usual 'pass by reference' semantics of C++ are orthogonal concepts (Fig. 3.16). Data can be passed by value between two value ports, as in the network example in Fig. 3.10, or by reference between two value ports, as in the example in Fig. 3.14. The first is useful when the datasets to be passed are small or when they have to be locally modified. The latter is useful when the receiver copies the data passed by reference and uses the copy locally, so it

```
module Producer
{ output:

     RDPortMeth* "data" "T*" (getData)
}
```

Figure 3.15: Declaration of reference ports in MC++ for the `Producer` metaclass

becomes actually independent on the passed reference, or when the components implement some reference counting scheme. Further, data can be passed by reference between two reference ports, as the example in Fig. 3.15. This is useful when data is to be shared and the component developer doesn't want to or can't implement reference counting. Finally, data can be passed also by value between two



MC++ \ C++	By value	By reference
By value	small datasets (int, float, RGB 3−float struct)	locally copy dataset or do reference counting
By reference	like above, but reset port to default value at disconnection	large datasets (matrices, fields, grids)

Figure 3.16: By value and by reference semantics in C++ and MC++

reference ports. For example, if an integer were passed between the `Producer` and `Consumer` reference ports in the above example instead of a `T*`, the integer's value would be copied. When the ports get disconnected, the write port would be set to the default value of an integer, in this case zero.

To summarise, whatever the passed C++ type between two ports is, `T` or `T*`, by value or by reference, two scenarios can happen when these ports get disconnected. In the first case, the write-port does *not* need to know that it has been disconnected, so nothing is done at disconnection. This is the by-value semantics of MC++. If the passed C++ type is a pointer, then the component developer must ensure that there are no dangling pointers or memory leaks, etc, by e.g. deep copying the pointed object in the write-port's write operation or using some form of reference counting on the pointed object.

In the second case, the write-port *does* need to know when it is disconnected. MC++ provides for this the reference ports which get automatically set to the default value of their C++ type when they get disconnected. This is useful to inform a component that there's no longer a connection at that port. This invariant ensures that the component developer will never suffer from dangling pointers.

Required Ports

MC++ supports the notion of *required ports* . A required port must be connected before its component can update. Required ports are useful for components that have reference ports. In many such cases, the component's update needs functionality offered by the components it read- or write-uses via the reference ports. The component can thus operate correctly only *after* all its reference ports have been connected. To enforce this invariant, ports can be declared required in MC++, by the insertion of the keyword `required` at the end of the port declaration. Reference ports are automatically considered required in MC++ even if they are not declared using the `required` keyword, since they usually have to be connected before their component can operate. If however a component may operate even when one of its reference ports is not connected, since e.g. its implementation checks for the NULL value of that port, the keyword not_required can be appended to the port declaration. This specifies that the respective port is not required, even though it is a reference port. The required and not_required keywords are seldom used in MC++, as the defaults of reference ports always required and value ports always optional usually suffice.

The 'This' Port'

Sometimes a component A directly uses another component B, e.g. when the operations of the two components are intimately related. A will read-use B if A keeps a reference to B and whenever B changes, A reads the new state of B to update itself. A write-uses B if A keeps a reference to B and whenever A changes, it directly modifies B by e.g. calling B's methods.

Such relations between object-oriented components are very frequent in the architecture of component libraries (see for example the Object Adapter, Mediator, or Decorator design patterns[37]). It is thus important to be able to model these relations at the MC++ component level as well. This is indeed possible as follows . The relationship A *read-uses* B maps directly to a dataflow model in which B has a read output port of type B* corresponding to its own this pointer and A has an input write port of type B* (Fig. 3.17 a). The relation A *write uses* B maps similarly to a dataflow model where A has a write output of type B* and B a read input of type B* that corresponds to its this pointer (Fig. 3.17 b).

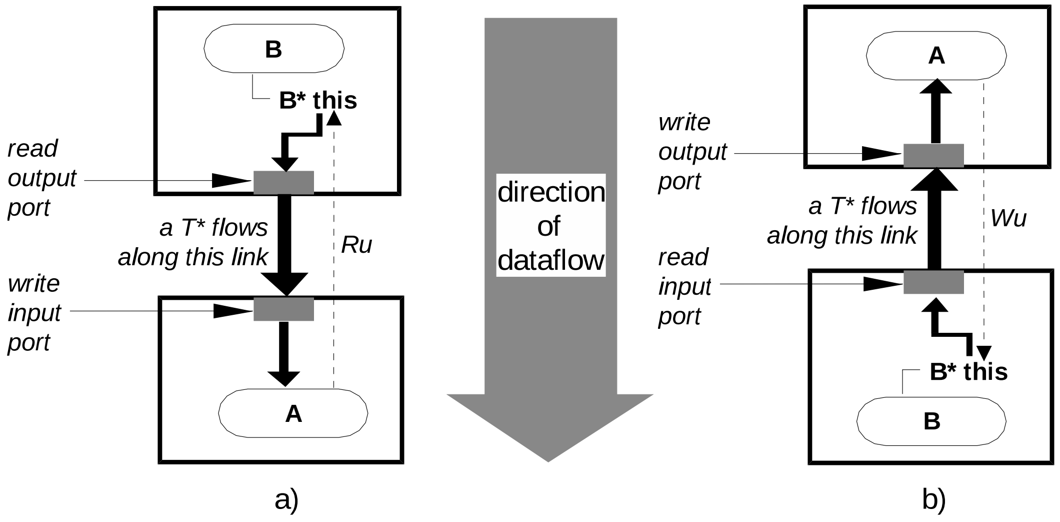


Figure 3.17: Implementation of read-use (a) and write-use (b) relationships between components

In the read-use case, the network update goes from B to A, as well as the B* this pointer which A receives. During its update, A 'pulls' the desired data from B via this pointer. In the write-use case, the network update goes from A to B, but the B* this pointer transferred along the A-B link *goes from B to A*. During its update, A 'pushes' the desired data into B via this pointer.

Write-use relations occur far less frequently that read-use ones in OO libraries. However, they are indispensable when using explicit *data-sink components*, or data stores that act as plain data repositories [91]. These have no update operation, but are updated by upstream components that write-use them. For example, a Matrix component is a data store with no update operation, which is write-used by e.g. a MatrixReader or MatrixBuilder component. Data sinks are often read-used in their turn by other components, leading to the appearance of chains of alternating *Ru* and *Wu* relations in a network. For example, a `Matrix` can be read-used by `Solver` or `Preconditioner` components.

In most dataflow systems, output ports are always read ports and input ports are write ports respectively. As explained in Section 3.3.1, this allows the implementation of read-uses relationships but not of write-use ones. By making the read/write, respectively input/output port attributes independent, MC++ can provide both uses relationships in a consistent framework.

To implement the above relations, an output read port and an input read port for the `this` pointer of a component's C++ object are needed. For a component of C++ type `T`, these could be declared as follows:

```
input:  RDPortMem "this" "T*" (this)
output:  RDPortMem "this" "T*" (this)
```

This would be tedious since the "this" port looks the same for any module. Therefore, MC++ provides automatically an input and output "this" read port for any component. The concept of "this" ports is analogous to the concept of the C++ `this` or Smalltalk `self` class members, which are automatically provided by the language to any class.

Signal Ports

Sometimes a component `A` needs to be informed that some 'event' has taken place in the dataflow network, e.g. that an output port "out" of another component `B` has changed. This can be done with the MC++ mechanisms introduced so far, by providing `A` with an input "inp" and connecting it to `B`'s output "out". This solution has however the disadvantage that once `A`'s input "inp" is built, it can be connected only to outputs that have compatible C++ types. However, the request stated above is that `A` needs only to be informed when a port has *changed*, and not of the port's new *value* as well. For example, two independent pipelines could be synchronised by making the start component of the first update automatically when the start component of the second updates. All we need in this case is to monitor when an output of the start component changes, its new value being irrelevant for the synchronisation purpose. Although such control flows are captured in the dataflow diagrams of Yourdon [120] and the UML notation [91], few actual dataflow systems implement them.

For the above, MC++ provides an untyped write port of the `WRPortSignal` kind . A `WRPortSignal` port can be connected to any other port regardless its type. Formally, `WRPortSignal` has the C++ type `void` that MC++ considers, by convention, to be compatible with any other C++ type, similar to the type `Any` in Eiffel [69]. When a link ending with a `WRSignalPort` gets activated, the port calls a provided method `mySignal()` of its C++ class and ignores the data output by the port at the other end of the link (Fig. 3.18 a). The port can be declared as follows:

```
input:  WRPortSignal "inp" "void" (signal)
```

where `mySignal()` is a C++ class method that must conform to the `void ()()` signature. If no argument list is provided to the declaration, no action will be taken when the `WRPortSignal` is activated, besides the usual triggering of its component's update operation.

MC++ provides also a read signal port kind called `RDPortSignal`. If `WRPortSignal` is seen as a generic signal receiver, `RDPortSignal` is a generic signal sender. When a write port is connected to a `RDPortSignal` that changes, the write port behaves as if it just received the data value it already had before the signal reception moment. `WRPortSignal` output ports are useful to force

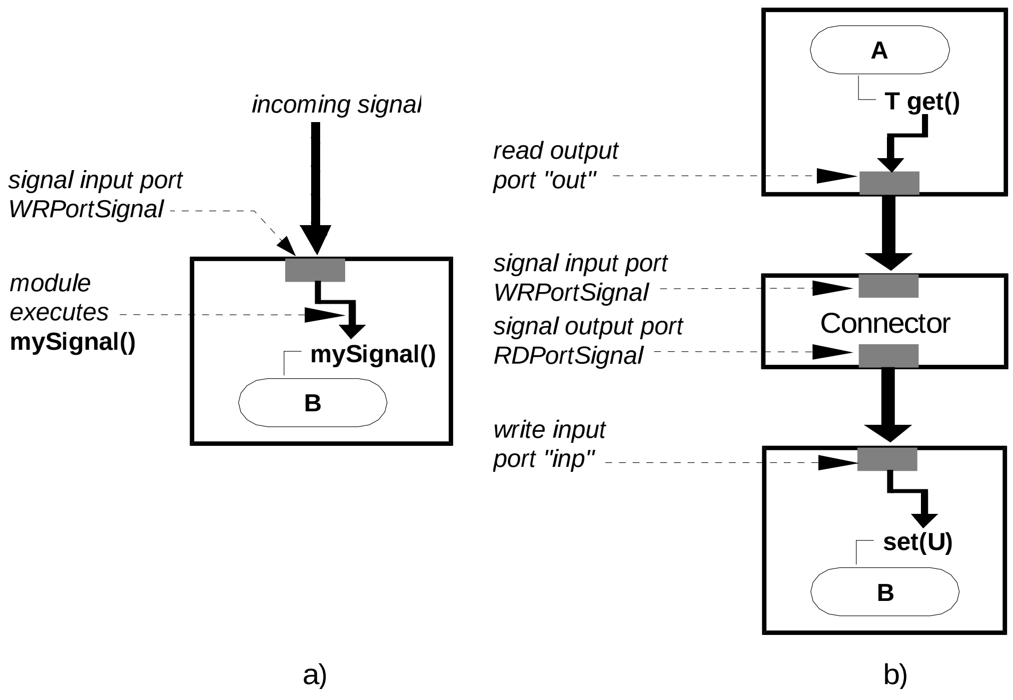

incoming signal
signal input port WRPortSignal
module executes mySignal()
mySignal()
B
a)
A
T get()
read output port "out"
signal input port WRPortSignal
Connector
signal output port RDPortSignal
write input port "inp"
set(U)
B
b)

Figure 3.18: Signal read port (a) and `Connector` component (b)

input ports of other components to 'refresh' their current value and thus determine their components to update.

The `RDPortSignal` and `WRPortSignal` port kinds can be used to build a generic connector, i.e. a component that can be used to connect two ports of any C++ types for signalling purposes. Figure 3.18 b presents an example where component `A`'s read output "out" of C++ type `T` is connected to component `B`'s write input "inp" of C++ type `U` via a `Connector` component.The generic connector has one `WRPortSignal` input with no class method associated to it, and one `RDPortSignal` output. When `A`'s "out" port changes, the `Connector`'s signal input port is activated. However, no data is read via "out"'s `get()` method, as the signal port does not need it. Next, the signal output port of `Connector` is activated. Since this is a `RDPortSignal`, `B`'s "inp" input behaves as if its current value has just been written into it via its `set()` method.

Connector components are very useful when closing loops in dataflow networks between ports of incompatible C++ types. In many such cases, the only information to be passed between components is the occurrence of a *change event*, regardless of the data values that have changed. The iteration of such loops would usually be controlled by some module that conditionally updates its outputs on a threshold value or a maximum iteration count. To prevent endless looping, several approaches are possible. In the current implementation of VISSION, the end user can interactively block the execution of a desired module or the whole network execution (Sec. 4.2.2).

Multiple Ports

Only ports bearing a single link to other ports were discussed so far. It should be remarked here that a read port can be connected to any number of write ports, as this is in accordance with the natural idea that a data value can be read by an unlimited number of readers. The write ports presented so far, however, can bear one connection. This is in accordance with the idea that one can write only a single value at a given time into a given location.

To model the number of links a port can bear, the concept of *port multiplicity* is introduced . The multiplicity of a port is the number of ports with which the port can be simultaneously connected. Read ports have thus an infinite multiplicity. The write ports presented so far have a multiplicity of one. The port multiplicity is an inherent attribute of every port kind implemented in MC++.

There are however cases when one needs to write a `list of values` into some location, e.g. when building a summator whose output is the sum of all its inputs. Constructing a summator component with several identical inputs is not a solution, since it does not allow the connection of a variable and unlimited number of inputs to the summator. What is needed is a new kind of write port with a multiplicity higher than one that can be connected to a variable number of read ports. The data provided by this port should be the list of values read from all the ports connected to it. MC++ implements the above in the form of the `WRPortDynArr` (write port using a dynamic array) port kind. A `WRPortDynArr` port ”inp” of C++ type `T` is implemented by five methods of its component's C++ class and is declared in MC++ as follows:

```
input:  WRPortDynArr "name" "T" (get,set,num,insert,erase)
```

The methods called `get,set,num,insert` and `delete` in the above example manage an ordered, random access dynamic array of elements of type `T` and have the signatures and semantics presented in Fig 3.19. The implementation of the above methods needed by the `WRPortDynArr`

WRPortDynArr methods		Method semantics
`T`	**`get`**`(int i)`	returns **i**th array element
`void`	**`set`**`(int i,T t)`	sets **i**th array element to value **t**
`int`	**`num`**`()`	returns the number of array elements
`void`	**`insert`**`(int i,T t)`	inserts value **t** at position **i** in the array
`void`	**`erase`**`(int i)`	removes **i**th array element

Figure 3.19: Methods needed for a `WRPortDynArr` port kind

port kind is left as usual at the user's freedom, e.g. using the C++ Standard Template Library containers. The dataflow network in Figure 3.20 illustrates the behaviour of the `WRPortDynArr` port kind for a summator component connected to three components that model the terms to be summed up. Fig 3.19. When a `term` component is connected to `summ`'s input, the method `insert()` is called to

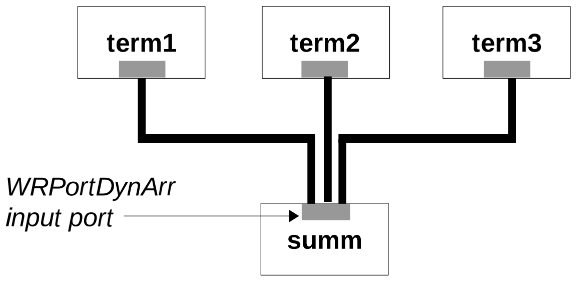


Figure 3.20: `WRPortDynArr` example

insert the data output by `term` in `summ`'s `WRPortDynArr` input port.The position in the list where the new `term` output is to be connected to the `summ` input is provided by the kernel which calls the connect operation. In some cases as in this example, the order of the connected inputs is irrelevant, so new `term` components can be simply appended at the end of the `summ`'s input port array. When a `term` component is disconnected from the `summ`'s input, the method `erase()` is called to erase the value from `summ`'s input port array that corresponds to the disconnected component. The methods `set()` and `get()` are used to transfer the data across the `fact-summ` links during the normal network traversal. Finally, the method `num()` is used by the dataflow kernel to determine how many ports are still connected to a `WRPortDynArr` port.

3.3.5 Metaclasses and Object-Orientation

In the above discussion of the metaclass concept, there are three ways in which MC++ is related to the object-oriented mechanisms provided by C++: metaclasses use C++ classes as their implementation, C++ types for their ports, and C++ source code for the extended form of their update operation. However powerful, the above component specification mechanisms provided by MC++ permits only to construct components out of unrelated classes. As presented in chapter 1, OO application libraries come in practice as *class hierarchies* and not as unrelated classes. In order to support the integration of class hierarchies, MC++ enhances metaclasses with the concept of feature inheritance.

A metaclass can have several base metaclasses, or bases shortly. This concept is similar to the public inheritance concept provided by C++. A metaclass inherits from its bases all their features, e.g. ports and update operations. The inheritance construct syntax in MC++ is similar to the one offered by C++. Figure 3.21 shows an example in which metaclass `C` inherits port "inp" from its base `A` and port "out" and the update operation from its other base `B`.

```
module A  { input:    WRPortMem "foo" . . . }
module B  { output:   RDPortMem "bar" . . .
              update:  { update }
            }

module C : A, B {  }
```

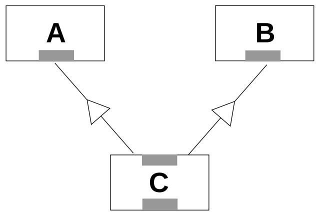


Figure 3.21: Metaclass inheritance example

Metaclass hierarchies created in the process of component library construction are usually homeomorphic to the C++ implementation class hierarchies, in the sense that the *is a* and *uses* relationships between metaclasses, respectively C++ classes are similar [91](Fig 3.22 a). The interface-implementation relationships between metaclasses and their C++ counterparts can be seen as instances of the Bridge design pattern [37]. However, the homeomorphism between metaclass and C++ class hierarchies is not mandatory in MC++. One could have, for example, a metaclass hierarchy with less elements than the corresponding C++ hierarchy, since it might be useless to expose the finer granularity of the latter in the MC++ space (Fig. 3.22 b). In this example, the component developer has skipped the mapping of the class `D` together with its ancestor `B` in the inheritance path `A-E`. This is dome by simply declaring that metaclass `E` inherits from `A`, thus skipping `B` and `D` completely from the MC++ hierarchy.

The decision of providing the inheritance concept in MC++ has a couple of important consequences. First, it is possible to have metaclasses that correspond to abstract C++ classes, i.e. classes which have pure virtual methods . This requires an extension of the concept of metaclass instantiability. In the absence of inheritance in MC++, a metaclass would be considered instantiable if its C++ class provides a publicly accessible default constructor and a public destructor. At the instantiation of

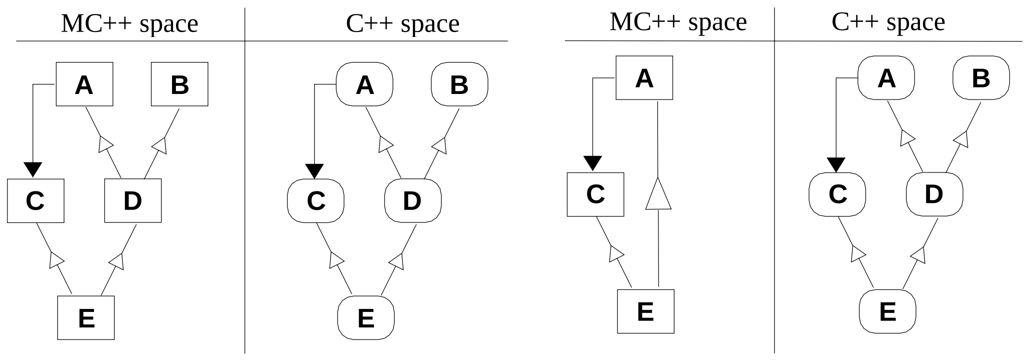


Figure 3.22: Homomorphism between MC++ and C++ class hierarchies (a) Simplified MC++ hierarchy for C++ class hierarchy (b)

a metaclass, a C++ object would be automatically created and used to provide the services required by the metaclass interface. When a metaclass instance is destroyed, its underlying C++ object would be destroyed as well. In the presence of inheritance, a metaclass is instantiable if its C++ class provides a publicly accessible default constructor, a public destructor, *and* is not abstract. The base-to-derived construction process and the reverse derived-to-base destruction process semantics of metaclasses are identical to the C++ one.

The second implication of inheritance regards the management of inheriting identical features from different bases and the overriding of features. The way MC++ approaches this issue is discussed in the following.

Inheritance and Feature Overriding

If a metaclass declares a feature with the same name as an inherited one, the new feature overrides (replaces) the inherited one. Since MC++ supports multiple inheritance, a scheme is needed to resolve the possible clashes between several inherited features with the same name. This scheme works differently for different features, as follows.

A metaclass that inherits several features with the same name from its direct bases will actually inherit the feature from its *last* base, as specified in the metaclass' base list declaration. This resolution scheme is used for all metaclass features except the update operation. If a unique update operation exists among all the direct bases, this will be inherited. If more direct bases provide an update operation, none will be inherited. The rationale for the above is that if several bases provide update operations, a metaclass that inherits from these bases should almost surely *combine* the inherited updates in its update operation. If more updates are provided, the safest is to inherit none. For the other features, the chance of multiply inheriting the same feature from different bases, e.g. an input port with the same name, is much smaller than for the update. In such a case, it was noticed from practice that a convenient default would be to inherit the feature encountered the last in the base list. Note that the input and output ports have different name spaces, so an input port could have the same name as an output port, as it is actually the case for the ”this” ports presented in Section 3.3.4.

The issue of dealing with multiple inheritance and name resolution is still a subject of debate in the OO languages community. For example, C++ accepts the inheritance of the same feature from different bases, but requires its explicit qualification with the base name for use afterwards. Other languages such as Eiffel [69] offer renaming mechanisms to rename the multiply inherited features to different names. MC++ chooses thus a middle way, as described above.

Feature Hiding

Feature hiding, also present in Eiffel and partially in C++, enables a metaclass to select which features to inherit from its bases. Feature hiding is implemented only for ports, since in all the other cases features can be redefined to take a default value, e.g. a void update operation. Figure 3.23 exemplifies feature hiding (similar in syntax to the analogous Eiffel language operation): metaclass *B* inherits inputs ”inp1”, ”inp2”, and output ”out” from its base A. All these ports are hidden by the presented construct that follows the base list, which contains a comma-separated list of inputs to be hidden,followed by a semicolon and a list of outputs to be hidden). In this example, B inherits actually none of the three ports.

```
module A { input:  WRPortMem "inp1" ...
                   WRPortMem "inp2" ...
           output: RDPortMem "out"  ...
         }

module B : A ("out","foo";"bar")
         { }
```

Figure 3.23: Hiding inherited features

3.4 The Meta-Type

Section 3.3.4 introduced the concept of default value for a C++ type. Default values have several uses in an interactive SimVis system:

- **initialisation:** reference ports could be initialised automatically by the system to their default values at component creation time. This would save the component developer the burden of remembering to set the corresponding data members to their default values.
- **security:** default values can be used to signal that a reference port has been disconnected, by resetting it to the default value of its C++ type. This could easily avoid problems caused by e.g. dangling pointers.
- **customisation:** the default value for a C++ type could be easily changed to a new value without recompilation (useful, for example, as several end users might have different opinions on what the default value of e.g. a RGB colour is).

Default values extend the C++ notion of data type in the same way metaclasses extend the C++ class concept. We would like to provide default values to all fundamental or class C++ types in the same non-intrusive manner we did for C++ classes by the metaclass construct. The best solution is thus to introduce a new MC++ construct. This construct is called a *meta-type* .

A meta-type is a MC++ language construct that adds several dataflow-related features to a C++ type. Formally, the data values that flow between components in a dataflow network are meta-type instances. The features provided by a meta-type are the default value and the serialisation operation. These features are presented in the following.

3.4.1 Default Values

To add a default value to a C++ type, a meta-type has to be created in MC++. The following example illustrates how the default values of zero, for integers, and the null string, for character pointers, can be added in MC++.

```
type "int" { default:  "0" }
type "char*" { default:  "NULL" }
```

The meta-type construct is similar to the metaclass one , i.e. it has a name, identical to the name of the C++ type it extends, followed by a list of features between braces. The above presents the default value feature which is introduced by the `default` keyword, followed by a string which contains a C++ expression that gives the default value of the C++ type. Any conforming C++ expressions can be provided as default values. For example, consider the C++ class `Point` that represents a two-dimensional point, and provides a constructor of the form `Point(float,float)`. If one desires to express the fact that the default value of a `Point` is the point at coordinates (0,0), this can be done by the following meta-type declaration:

```
type "Point" { default:  "Point(0,0)" }
```

Meta-types are object-oriented entities. If, for example, one declares a C++ class `Point3D` which extends the above `Point` to model 3D points, and a default value is provided for `Point3D`s but not for `Point`s, then the `Point3D` default value will be used also for the `Point` type.

3.4.2 Serialisation Operation

There are many cases when the state of a component has to be copied to a different location, e.g. when components are to be cloned, transferred to a different place over a network, or saved on external support for later use, in the case of persistent systems. All these operations require a generic *serialisation mechanism* that can transform a component's state into a system-independent representation which can be transferred, cloned, or stored in a save file.

The state of a SimVis application is composed of the dataflow network topology and the states of all the dataflow components. The state of a dataflow component is entirely described by the values of its inputs and outputs. Since we model inter-component connections as links between ports, the state of the entire system is given by the values of all the component ports. Operations such as saving the state of a dataflow network, cloning networks, or transferring them over a machine's boundary can be thus easily implemented once we provide a mechanism to serialise the value of any component port. Since ports in MC++ bear C++ values, we need a mechanism to serialise any C++ type. This implies representing the value of an instance of that type in some generic way, e.g. as an ASCII string, and constructing an instance of that type from the serialised representation. The above operations are trivial for fundamental C++ types, whose serialisation is simply their value, e.g. ”12” for an integer, or ”abc” for a *char**, etc. This is however less trivial for class types which have no default serialisation mechanism, i.e. no standard way to encode their value to a string and restore it from the same string.

The above serialisation functionality can be provided however at the meta-type level in MC++, by the `store` meta-type feature. Similar to the extended form of the metaclass `update` operation, the meta-type `store` feature specifies a C++ function in source form. For a C++ type `T`, the `store` function expects a `char*` buffer and an object of type `T`, and should serialise `T` in the provided buffer. The function signature is thus `void ()(char*,T&`, where `T` is the type to be serialised (Fig. 3.24). The

```
port "T" {
          store: void store(char* buf,T& obj)
                 { ... serialize obj into buf ... }
          }
```

Figure 3.24: The `store` meta-type feature

question still to be answered is how to implement the serialisation of any type `T` via the above interface. Rendering `T`'s internals into some string format is not an option, since then a special procedure to restore `T` from that format should be devised as well. This involves a complex, hard to generalise and use mechanism. Moreover, this would break the C++ encapsulation principle by allowing code external to the C++ class to directly set the class internals. We can avoid this by noticing that any C++ type has a natural way to be created from other data, namely its constructor that uses that data as parameters. Consequently, the `store` operation should simply write in the provided string buffer a C++ expression denoting the constructor call which, if evaluated, would produce an object identical to the one being serialised. For example, consider the class `Point` discussed above (Fig. 3.25 a). The

```
class Point                         port "Point"
{                                   {
public:                             store:

      Point(float x_,float y_)      void store(char* buf,Point& obj)
             :x(x_),y(y_) { }            {
float getX() { return x; }                 sprintf(buf,"Point(%f,%f)",
float getY() { return y; }                         obj.getX(),
                                                   obj.getY());
private:                                 }
      float x,y;
};
             a)                                  b)
```

Figure 3.25: C++ type `Point` (a) and its meta-type serialisation operation (b)

`Point` constructor offers a perfect way to serialise a `Point` value, as shown in the `Point` meta-type (Fig. 3.25 b).

Reference ports represent a special case, since their value is not only a C++ data value but represents also a reference to another port. Representing the value of a reference port as its C+ data value would be incorrect. Reference ports carrying C++ pointers, for example, would be serialised incorrectly if one would e.g. simply copy their pointer values across machine boundaries. Consequently, reference port values are defined in a different way.

A reference port value is defined by a string with the following structure:

```
<objname:portname.index>
```

where `portname` is the referred (target) port, `objname` is the name of the component that owns the referred port, and `index` is an integer that indicates the position of the connection in `portname`, as explained in Section 3.3.4. For example, a reference port connected at position 1 to the ”info” port of the object `data` will have the value `<obj1:info.1>`. The value of an unconnected reference port is the empty string. The use of the above reference port value encoding scheme is demonstrated in Sections 4.4.4.

3.5 The Meta-Group

An important problem applications designers face is that applications built as dataflow networks give raise to very large component networks. Managing large graph-like diagrams in e.g. a visual programming environment can be difficult, since all dataflow components would be displayed on the same abstraction level. What one needs is a way to represent *structure* in such a graph. For example, a dataflow network like the one in Fig. 3.26 (a) could be divided into several sub-networks by grouping components related by their functionality into three groups (Fig. 3.26). Finally, application designers could use the simplified network containing just one component D and the three component groups .

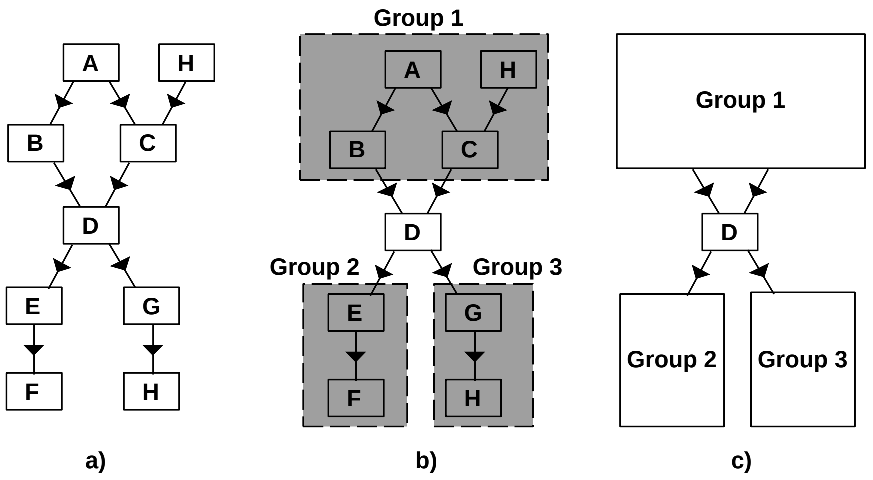


Figure 3.26: Dataflow network: unstructured version (a), introduction of groups (b), and simplified version (c)

The idea of aggregating dataflow components into larger groups implements actually the third software modelling relationship that missed from our dataflow-based analogy. If the OO metaclass concept models encapsulation and the *is a* inheritance relationship, the port-to-port links model the *uses* relationship, then component aggregation int groups model the *has a* relationship.

Component groups address two important application designer requirements. First, they help managing complexity by breaking down a network into a few component groups. Second, component groups could represent higher units of software reuse than components. For example, an application designer might want to reuse a whole sub-network with a specific functionality in different contexts. For this, we need a mechanism to manipulate the sub-network as a dataflow component in itself, which has a well-defined structure, interface, and functionality. This mechanism was implemented in MC++ by the *meta-group* concept.

A meta-group has a similar dataflow interface with a metaclass, i.e. it offers input and output ports, and an update operation. Meta-groups are built as recursive aggregations of other meta-groups or metaclasses. They offer to the component developer a simple mechanism to describe complex functionality by recursive aggregation of *children* components into a *parent* meta-group.

3.5.1 Example

We shall illustrate the meta-group construct by an example. Suppose one has the dataflow network in Fig. 3.27 a, composed of two components `A` and `B`. The MC++ code in Fig. 3.27 b declares a meta-group which is equivalent to this network. Similar to other MC++ meta-construct declarations, the meta-group declaration starts with the name of the meta-group `G`, followed by the group's features between braces. The group has two children, i.e. the metaclasses `A` and `B` declared in `G`'s `children` section. These are given the names `obj1`, respectively `obj2` inside the meta-group scope. The `children` section is practically identical to the data member declarations of other OO languages. Next, `G` declares that `obj1`'s input "inp1" will become a group port by the name of "input 1", and similarly for `obj1`'s input "inp2" and `obj2`'s output "out". To identify a child port inside a group declaration, MC++ provides the `<child name>.<port name>` syntax, similar to the aggregate member referencing syntax of most OO languages. The exported ports have the same properties and type as the ports to which they delegate.

Children ports such as `obj2."inp4"` that are not exported at the group level remain hidden inside the group and are not accessible by the group's clients. Finally, the `connect` section of `G`'s declaration specifies the internal connections between the group's children, such as the link between `obj1."out1"` and `obj2."inp3"`.

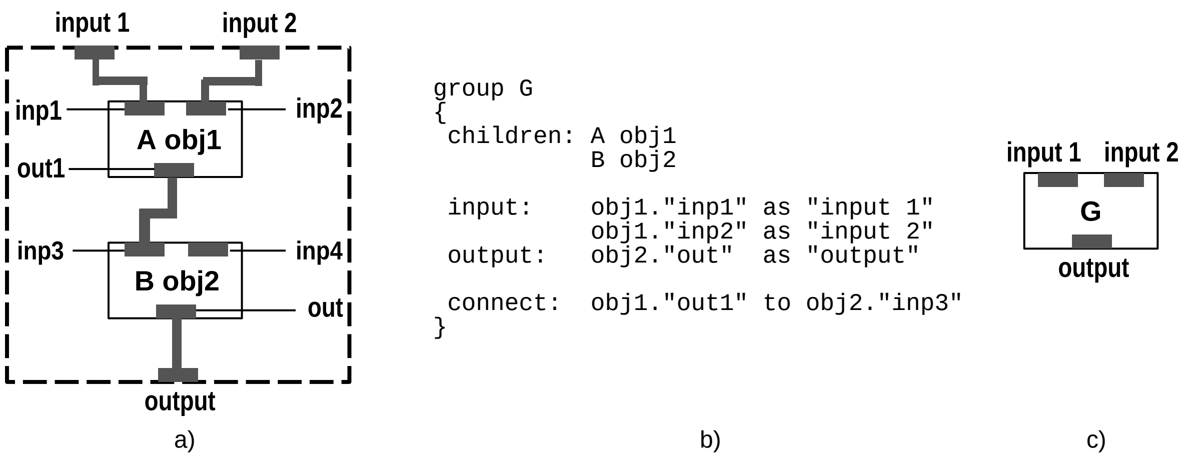
3.5.2 Meta-group Advantages

After its declaration, a meta-group can be used either as a part of another meta-group's declaration by the component developer, or directly by the application designer in the construction of networks. When a meta-group is instantiated, all its children are recursively instantiated and connected as the group's declaration specified. Next, the group component is accessible exactly as a meta-class component, via its ports. When a group updates, its internal sub-network of components is traversed from the group inputs to the outputs and every child is updated. Consequently, the meta-group behaves identically as if its internal component sub-network were manually built by the application developer instead of the group. Meta-groups provide a metalanguage-level mechanism to encapsulation, aggregation, and reuse for the dataflow model, much in the same manner that it is done by OO languages for component design. The advantage of using meta-groups to build newer components instead of coding their aggregation in C++ is that MC++ offers a much simpler and safer programming manner. Furthermore, as it will be presented in chapter 4, meta-groups built at the MC++ level are available for interactive exploration in a visual programming environment, which could not happen if they were designed as C++ level code aggregates.

The idea of language-level aggregations is not unique to MC++. Open Inventor, for example, offers the similar concept of *node kits* which are assemblies of C++ components, called *parts*. Node kits model 3D objects having a fixed structure, given by the parts' assembly, but customisable attributes, given by the input ports of the node kit. Due to the statically compiled nature of Open Inventor, node kits have however a rather complex implementation and are difficult to be built by the component developer. Besides Open Inventor, only few other SimVis systems raise component aggregation to the level of a type.

3.6 The Meta-Library

The *meta-library* is the largest code unit for dataflow modelling. A meta-library is a collection of meta-constructs such as metaclasses, meta-types, and meta-groups, which is created by the component de-



```
group G
{
 children: A obj1
           B obj2

 input:    obj1."inp1" as "input 1"
           obj1."inp2" as "input 2"
 output:   obj2."out"  as "output"

 connect:  obj1."out1" to obj2."inp3"
}
```

Figure 3.27: Meta-group example: initial network (a), MC++ code (b), and final meta-group (c)

veloper and used by the application designer to build dataflow networks. Usually a meta-library contains all components needed to model a given application domain, such as 3D graphics, scientific data representation, data visualisation, or numerical computations. The second role of the meta-library is to provide the mechanisms that connect the MC++ declarations to their compiled C++ implementation. The structure of a meta-library file is illustrated by means of an example (Fig. 3.28). Two meta-

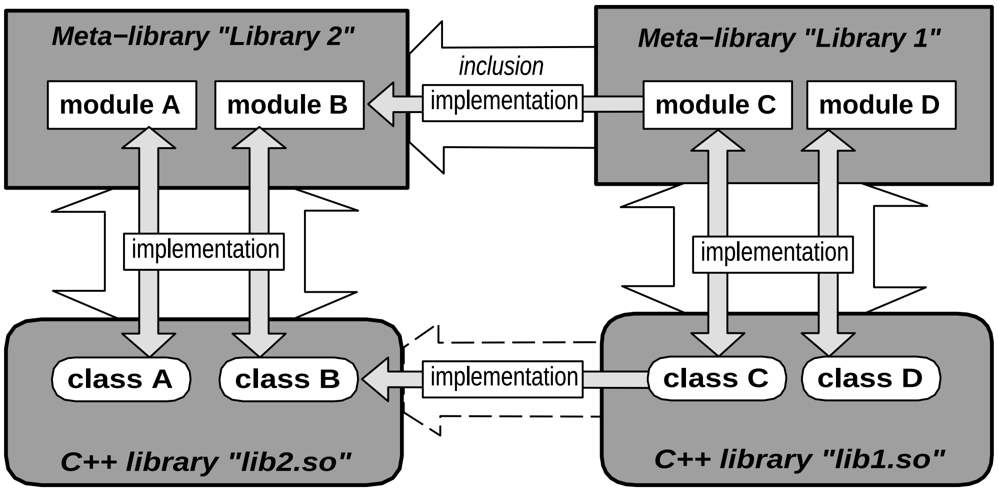


Figure 3.28: Meta-library example: *Library 2* includes *Library 1* and is implemented by *lib2.so*

libraries `Library 1` and `Library 2` are declared, which offer the metaclasses `C,D`, respectively `A,B`. These metaclasses are implemented by the C++ classes with the same names located in the binary libraries `lib1.so` and `lib2.so`, respectively. Finally, there is a dependency between `Library 1` and `Library 2`, in the form that metaclass `B` inherits from metaclass `C`. Other similar dependencies could exist, such as a meta-group from `Library 1` having a child declared in `Library 2`, etc. The meta-libraries and their implementation counterparts exhibit a homeomorphic architectural pattern similar to the one exhibited, at a finer level, by the metaclasses and their C++ implementations (Section 3.3.5). The two libraries `Library 1` and `Library 2` are actually two MC++ files `lib1.mh` and `lib2.mh` whose structure is presented in Fig. 3.29 a,b. The structure of a MC++ file

```
library:          "Library 1"
implementation:   "lib1.so"
initialization:   { printf("Library 1 loaded\n");   }
finalization:     { printf("Library 1 unloaded\n"); }

module C         { .... }
module D         { .... }
```

a)

```
library:          "Library 2"
include:          "lib1.mh"
implementation:   "lib2.so"

module B : C     { .... }
module A         { .... }
```

b)

Figure 3.29: Meta-library MC++ code examples

is similar to the one of a Java package or a C++ header. The library declaration starts with the library name, followed by optional `include` statements that specify the meta-libraries on which the current library depends. Libraries depend on each other in much the same way the application domains they model depend on each other. Next, the C++ implementation of the meta-library is given. In our actual implementation, this is a compiled C++ shared object library. Next, an `initialization` section can provide optional C++ source code to be executed just after the library is loaded. For example, `Library 1` in Fig. 3.28 a) prints a message when it is loaded by the dataflow kernel. Initialisation sections are useful, for example, for X11/OpenGL [77, 119] rendering libraries that have to initialise their rendering contexts before they perform any other rendering action. Initialisation code can call e.g. global library functions or static class methods. Similar to this section, a `finalization` section may optionally provide actions to be executed just before the library is unloaded. Finally, the MC++ declarations of the library's components follow.

Meta-libraries are similar to e.g. Java packages or C++ class libraries. The dataflow kernel is able to dynamically load and unload such libraries and their provided functionality. Loading a meta-library triggers the loading of all libraries on which it depends that are not already loaded. Similarly, unloading a meta-library triggers the unloading of all libraries on which it depends that are not used by any other loaded library. Although MC++ does not specify the above operational semantics, this is (and should be) part of the OO dataflow model, in the same way that the C++ object model that specifies the C+ run-time semantics is part of the C++ language standard [30, 58].

3.7 Conclusion

This chapter has presented a new SimVis system architectural model. The new model combines the advantages of the library, framework, and turn-key application architectures and eliminates their major disadvantages. The main feature of the new model is a high-level framework-application component interface that combines object-oriented and dataflow modelling in a non-intrusive manner. The interface is implemented at a meta-language level, as the MC++ component specification language. MC++ provides several constructs that allow classical C++ components to be smoothly merged into a dataflow environment without requiring their modification or interface adaptation. On the other hand, the powerful features of the C++ object model remain available to be used by the dataflow kernel. The strict separation of the MC++ dataflow and C++ OO modelling helps further in the transparent integration and intercommunication of existing SimVis libraries. Keeping the two modelling worlds separate simplifies the component development path and makes understanding and extending component code easier. Finally, the C++ application code is truly system-independent, and it can be easily used in any other context or system, since all its dependencies on VISSION are confined to the MC++ 'adapter' structures.

The OO-dataflow combination represents the foundation on which VISSION, a flexible general-purpose SimVis environment has been built. This chapter has explained the advantages of the new architectural model based on MC++ from the perspective of the component developer. The next chapter will present VISSION from the perspective of the application designer and end user, and illustrate the advantages of the architectural model presented here from these two perspectives. The implementation of VISSION's architecture presented here is the subject of chapter 5.

Chapter 4

Application Design and Use in VISSION

In the previous chapter we presented the architectural basis on which the VISSION SimVis environment is built. The advantages of the new OO-dataflow combination based on the MC++ meta-language were presented from the component developer perspective. However, unlike in similar SimVis application builders, the architectural model is not opaque to the application designers and end users that work with VISSION. An important feature of VISSION is that it uses the same application model for all its user groups. This chapter shows how the OO-dataflow application model presented in the previous chapter is reflected in the application designer and end user interaction with VISSION. Throughout this chapter, several advantages of using a single application model for all user groups will become apparent.

4.1 Background

Existing SimVis systems offer a large variety of user interfaces to their different classes of users. For application designers, such interfaces range from simple script-based application specification to complex visual programming and steering GUIs, as the ones shown in Figs. 2.2 , 2.3. For end users, interfaces also range from simple text-based input-output scenarios to interactive GUI-based visualisation and steering tools.

Many SimVis systems exhibit no visible structural correspondence between the application internal representation and the user interface. The user interface is often conceived as a back end that is loosely coupled with the SimVis kernel. This fact is not problematic for turn-key systems in which the application internal structure is anyway hard-coded and not of interest to the end user. Environments that support the application designer role must however offer a user interface to the application structure, in order to edit it by e.g. adding or deleting components. Moreover, systems that support the component developer role must also offer a way to represent new components in their application designer and end user interfaces.

Most SimVis systems based on the dataflow application model approach the above tasks by three different user interfaces:

1. First, a text-based interface provides commands to construct an application by instantiating components, connecting them together, and setting their inputs to the desired values. This interface can serve the end user as well. Commands are provided to change the value of the components' inputs, re-execute the application, and query component output values.

2. Secondly, application design is provided also graphically by editing an icon-based visual representation of the dataflow network.

3. Finally, end users can inspect or steer the components by using GUI dialog panels. These panels show the components' input and output values by means of various widgets such as text fields, dials, sliders, or even 3D renderings. Examples of the above interfaces are shown in Fig. 2.3 for several SimVis systems.

The user interfaces of most component-based SimVis frameworks have a couple of limitations:

1. **GUI Construction:** Constructing a new interface for a new component added to the system, or for a whole application, is usually complicated. This implies modifying the component code to insert GUI code (AVS or Matlab), manual construction of the component's GUI in a text-based or visual GUI editor (AVS/Express), or manual coding a GUI interface in a separate programming language (e.g. the Tcl/Tk languages for VTK and Oorange). In all cases the integration of new components in the system is far from automatic. Moreover, the end user requirement for a component GUI propagates back to the component developer. Component code is either not system independent any longer or a separate GUI component must be written in some special programming language. These clearly contradict our requirements presented in the previous chapters. However, the option of visually building a GUI for an entire application or a component is an important feature that we would like to have in VISSION as well.

2. **Extensibility:** As shown in the previous chapter, components may have various input and output types, ranging from integers and strings to complex user-defined types. Most SimVis systems provide however only a few, hard-coded (thus fixed) GUI widgets that can edit only a few basic types. It is usually impossible to construct a custom GUI for a user-defined type such as a three-float vector or a RGB colour, especially if this is an object-oriented type. The only solution left requires elaborate manual mapping of that type to the supported types.

3. **Completeness:** Text interface or scripting languages are usually less powerful than the component development language. Consequently not all the component facilities are accessible to the end user scripts or text interface, as detailed in Section 2.5.2.

Conceptually, the above limitations of user interfaces stem from the fact that existing SimVis systems do not have an adequate representation of the component notion. This chapter shows how the component notion based on the OO-dataflow model presented in the previous chapter addresses the above problems in a generic manner.

4.2 Application Design User Interface

First we address the user interfaces offered to the application designer. As discussed in the previous chapters, application designers require tools to assemble an application, represented as a dataflow network, from components provided by the component designer. In VISSION, components are represented as constructs of the MC++ language (metaclasses, meta-groups, and meta-types) gathered in meta-libraries. It is hence easy and convenient to construct the application design user interface directly based on the MC++ concepts. The mapping of the various MC++ concepts to a visual representation in the application design GUI is the subject of the next sections.

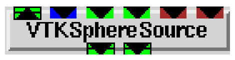
4.2.1 Icons and Networks

The first element provided by the application design GUI is the *iconic representation* of metaclasses and meta-groups and of their instances . Every metaclass, meta-group, or instance thereof in VIS-

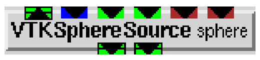
SION is visually represented by an icon. The icon displays graphically several features of the represented MC++ entity such as the input and output port types and kinds, the entity's type, and the entity's instance name. We shall illustrate the above by an example. The metaclass `VTKSphereSource` (Fig. 4.1 a) is an actor that generates a faceted approximation of a sphere sector. Its input parameters are the sphere radius, the angles θ_0,θ_1,ϕ_0, and ϕ_1, that delimit the start and end of the spherical sector to be approximated, in the polar coordinates θ and ϕ, and the resolution of the polygonal discretisation in both polar directions. These parameters are modelled by the float input ”radius”, the `Vec2f` inputs ”theta” and ”phi” (where `Vec2f` models a 2-float vector), and the integer inputs ”theta res” and ”phi res” respectively. The metaclass outputs a dataset of C++ type `VTKPolyData` that carries the sphere's tessellation as a list of polygons.

```
module VTKSphereSource
{
input:  WRPortMeth "radius"    "float"       (...)
        WRPortMeth "theta"     "Vec2f"       (...)
        WRPortMeth "phi"       "Vec2f"       (...)
        WRPortMeth "theta res" "int"         (...)
        WRPortMeth "phi res"   "int"         (...)

output: RDPortMeth* "output"   "VTKPolyData" (...)
}
```

a)



b)



c)

Figure 4.1: Icons for metaclass `VTKSphereSource` (b) and metaclass instance `sphere` (c)

The icons for the metaclass `VTKSphereSource` and an instance of it, called *sphere*, are shown in Fig. 4.1 b,c respectively. The metaclass icon, shown in detail in Fig. 4.2, displays the input and output ports of its underlying metaclass in the upper respectively lower part of the icon, the metaclass type. If the icon represents an instance, the instance name is displayed as well. Every meta-entity instance has a unique name in VISSION, much as a variable has a name in a program. The arrows on the small rectangular port symbols depict the port read/write attribute: an arrow going into the icon denotes a write port, while the opposite direction denotes a read port. The port symbols are framed with a black line if the port is a reference port, as it is the case for the ”this” and ”output” ports. The colour of the port, rendered as gray shades in Fig. 4.2, encodes the C++ port type. The mapping between types and colours can be specified by the component developer when writing the MC++ meta-libraries. The

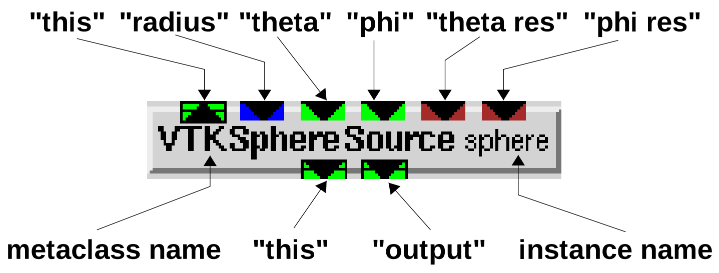


Figure 4.2: Graphic details of metaclass icon

metaclass icon representation is very similar to the ones used by the GUIs of other SimVis systems, such as AVS, Data Explorer [1], or Oorange. Metaclass icons are however visual displays of OO entities, which means that the icon of a derived metaclass will automatically contain all the ports it inherits from its ancestors, besides the ports it declares itself. Hidden ports (see Section 3.3.5) are not displayed, since they are actually not inherited.

A dataflow network is visually represented as a graph where the nodes are icons corresponding to the dataflow objects and the arcs are the links between the connected inputs and outputs. For example, let us consider a simple visualisation network for the tessellated sphere produced by `VTKSphereSource` (Fig. 4.3 a). The network starts with a `VTKSphereSource` *sphere* that generates a polygonal dataset, followed by a `VTKDataSetMapper` instance *mapper* that maps the dataset to drawable primitives. A `VTKActor` *actor* represents the viewable object which is finally rendered by a 3D viewer `VTKViewer`. Figure 4.3 b shows the image produced by the pipeline for the input values ”theta”=(0,180), ”phi”=(0,270), and ”theta res”=”phi res”=20.

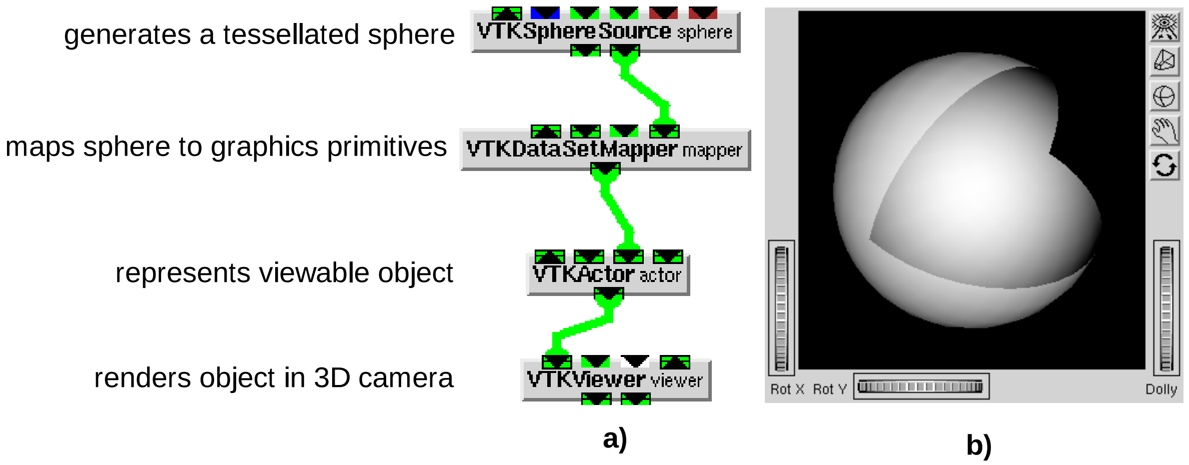

generates a tessellated sphere
VTKSphereSource sphere
maps sphere to graphics primitives
VTKDataSetMapper mapper
represents viewable object
VTKActor actor
renders object in 3D camera
VTKViewer viewer
Rot X
Rot Y
Dolly
a)
b)

Figure 4.3: Simple visualisation pipeline (a) and resulting image (b)

The mapper-actor sub-pipeline is frequently met in the back-end of visualisation networks. In order to simplify such networks, these two metaclasses can be grouped into a meta-group, whose MC++ declaration is given in Fig. 4.4 a. The simplified pipeline using the `VTKActorSet` meta-group icon is presented in Fig. 4.4 b. The meta-group icon is identical to the metaclass icon with the exception that the meta-group name is displayed between angular brackets to indicate that the icon corresponds to a meta-group and not to a metaclass.

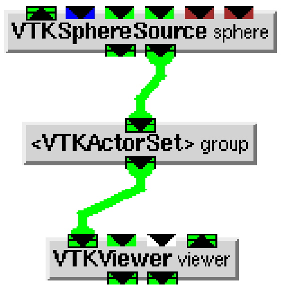
4.2.2 The Network Manager

The icons presented above are the graphics building bricks for a dataflow network. VISSION provides a GUI, called the *network manager*, in which these icons can be manipulated to actually perform the application design. The network manager consists of two visual components: the library browser and the network editor. Overall, the network manager is built on the model offered by the AVS and Iris Explorer visualisation systems. Figure 4.5 shows the simple visualisation application used as an example in the previous section in the network manager.

The *library browser* is an automatically constructed visual representation of the meta-libraries that are loaded into VISSION. The instantiable metaclasses and meta-groups present in these libraries are displayed in several scrollable icon stacks, depending on the components’ *categories*. Every component in a meta-library belongs to a category. Categories allow grouping related components together,

```
group VTKActorSet
{
children:     VTKDataSetMapper mapper
              VTKActor         actor
input:        mapper."input"
output:       actor."this"

connect:      mapper."this" to actor."mapper"
}
```

a)



b)

Figure 4.4: Group icon used in visualisation pipeline

such as mappers, filters, data sources, data writers, or datasets. Inheritance or other OO relationship is not mandatory between components in the same category. Indeed, inheritance reflects a commonality in interface and implementation. Categories, however, reflect different concerns such as a common functionality as seen by application designers. Modules related by inheritance are thus often grouped in different categories. An example are the VTK data filters which inherit from the `VTKDataSetFilter` module which, in its turn, specialises the `VTKDataSetSource` module. The VTK data readers are also specialisations of `VTKDataSetSource`. However, filters and readers are traditionally seen as different categories by application designers, even though they share the same `VTKDataSetSource` base in the VTK component library.

The category attribute is inherited by components exactly as the port attributes (Section 3.3.5). The component category is thus just a label attached to a component for grouping functionally related components together in the library browser. In this respect, categories are similar to the aspect concept used in the field of aspect oriented programming [54].

A component's category can be specified by the MC++ *category* attribute, followed by the category name, in the component's declaration. Figure 4.6 shows the declaration of the `VTKActorSet` metagroup that falls into the ”mapper” category. Components in each category are displayed in a separate stack in the library browser. For example, Fig. 4.5 shows component stacks for the ”filter”, ”mapper”, and ”data” categories. Components with no explicitly specified category are displayed in a separate stack labelled ”others”.

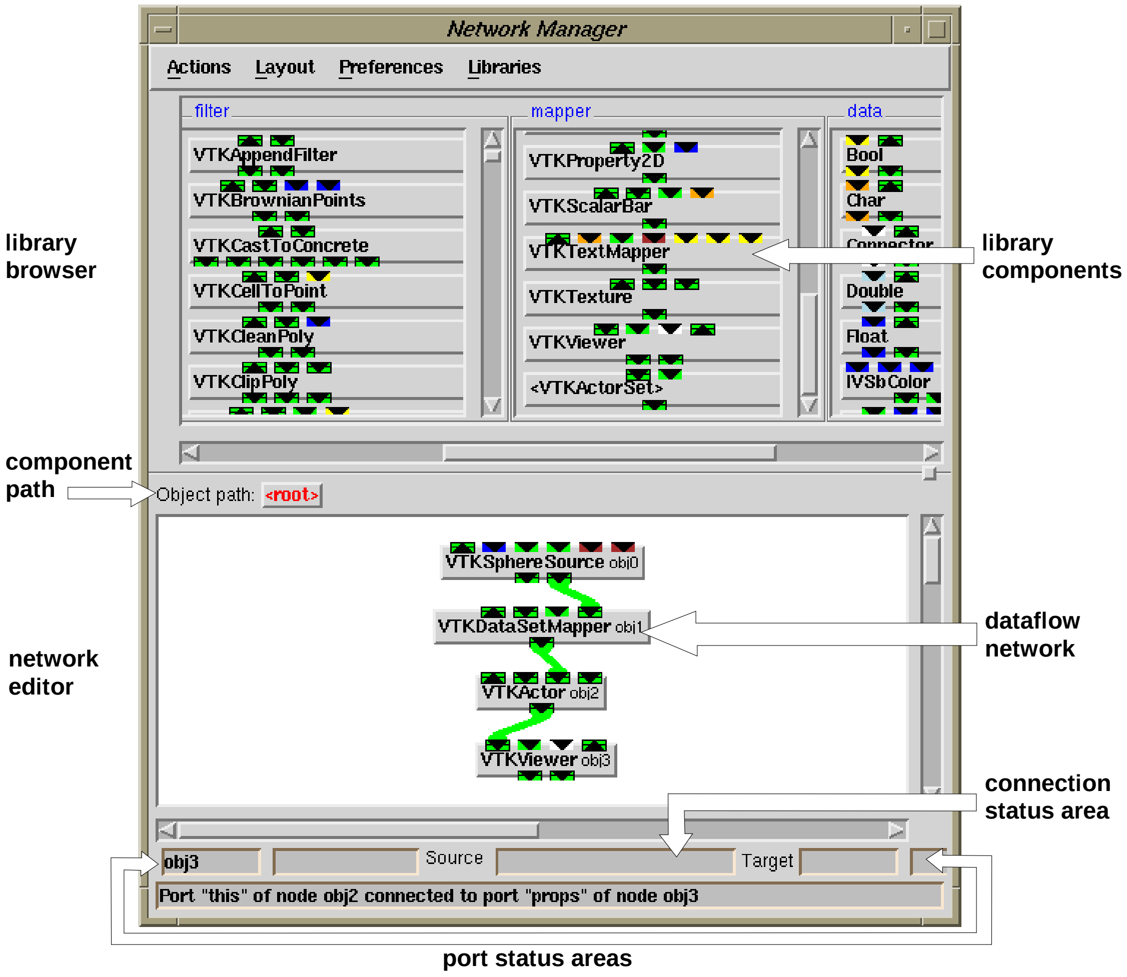
The *network editor* displays a visual representation of the dataflow network existent in VISSION. Next, the network editor offers various network editing functions. The most important functions are the following:

- Instantiation of components by dragging their icons from the library browser into the network editor.

- Deletion of components by using a popup menu on the component icons. The deletion of a component starts by destroying its links, followed by the destruction of the actual component.

- Cloning of components. When a component is cloned, a new component of the same type and with the same input values is created. Cloning is thus the dataflow equivalent of the OO concept of copy constructing [102].

- Renaming of components by means of a GUI dialog.



Port "this" of node obj2 connected to port "props" of node obj3

Figure 4.5: The network manager

- Interactive navigation into meta-groups. Every component has in VISSION a parent component, as described in Section 3.5. The components that are not children of any group are regarded as children of the *root* group. The root group is the invisible root for the parent-child component hierarchy. The network editor displays by default the contents of this root group. The application designer can navigate into group components by double-clicking on their icons. When a group is entered, the network editor displays the contents of that group, i.e. its children components. The path from the currently displayed group to the root group is shown on the *component path* widget of the network editor (Fig. 4.5. This widget allows also an easy navigation upwards from the current group. Alternatively, one can go one level up in the hierarchy by double-clicking on the network editor's canvas.

- Connecting components. Links between component ports can be established in the network editor by a click-drag-release procedure starting at one port and ending at another port. If the two ports are compatible, as described in Section 3.3, they are connected and a link is displayed between them in the network editor. If the ports are not compatible, the incompatibility reason is displayed in the network manager's connection status area and the connection attempt is declined. During the connection, information on the selected ports such as port and component

```
group VTKActorSet
{
children:    VTKDataSetMapper mapper
             VTKActor         actor
input:       mapper."input"
output:      actor."this"

connect:     mapper."this" to actor."mapper"
category:    "mapper"
}
```

Figure 4.6: Category declaration

names and types is displayed in the additional port status areas. In the case of a multiple port (see Section 3.3.4), the link is created at the last port position, which is equivalent to appending the link to the existing ones. Another method of connecting components which allows insertion of the link in a different position into a multiple port is presented in Section 4.3.1.

- Disconnecting components. Links between component ports can be destroyed in the network editor by a similar click-drag-release procedure used for connecting the ports.
- Locking and unlocking components. In order to block the execution of a given network branch, one can lock a component icon by using a mouse-based popup menu. The network traversal will stop when a locked component is reached. Unlocking the component will resume the network update from that component. In this way, one can stop infinite loops or selectively control the network execution.

In many visual programming systems the application designer can construct groups interactively by adding components to an empty group. An example hereof is the ’macro module’ concept of AVS and AVS/Express [113]. Such groups differ from the meta-group concept presented so far in two ways:

1. First, the former are built by the application designer on the fly, while the latter are types built first by the component developer and then instantiated by the application designer.
2. Macro modules may be edited after they have been constructed. In contrast, meta-group instances are basically type instances and thus have a fixed, immutable structure. This means no children can be added to or deleted from a meta-group. The same is valid for links among group children. Allowing meta-group instances to be edited would violate the principle that an instance respects the invariants stated by its type and would effectively mutate the type at run-time.

Building groups on the fly is a powerful design tool needed in the case one wants to hide a sub-network’s complexity for a single application. Since the group is used in a single application, creating a meta-group in a meta-library would be too heavy-weight.

VISSION provides for the above the concept of *empty groups*. An empty group is a special meta-group with no children and no ports, which can be interactively edited by the application designer. The empty group is not part of any meta-library, but is built into VISSION. Its icon is displayed in the library browser and has, by convention, the empty string as type name. In the editing process, children can be added and connected into the empty group in the normal fashion presented above. Ports of these children can be ’exported’ to the outside of the group to allow communication of the group children with external components. This is done by double-clicking on the desired ports. The semantics of these interactively editable groups is identical to a meta-group instance that has the same structure. The only difference is that the structure of the latter can not be changed, while the one of the former can.

Another network editing operation is component *reparenting*. A group child can be reparented, i.e. moved into another group as its child. The network editor provides mouse-based functions to reparent components between groups by dragging them into or outside of the desired group icon. Reparenting is allowed only between editable groups for the reasons described above. Moreover, all links between a component that undergoes reparenting and other components outside its parent are severed during the reparenting operation. This prevents the coupling of components with different parents via links that do not pass through the parents' ports. This is forbidden since it is a violation of the principle that a component, thus also a group, communicates with its outside *only* via its ports.

Besides the above operations, the network editor provides various visual layout tools, such as icon dragging, aligning, link and icon drawing customisation, and so on.

4.2.3 The MC++ Browser

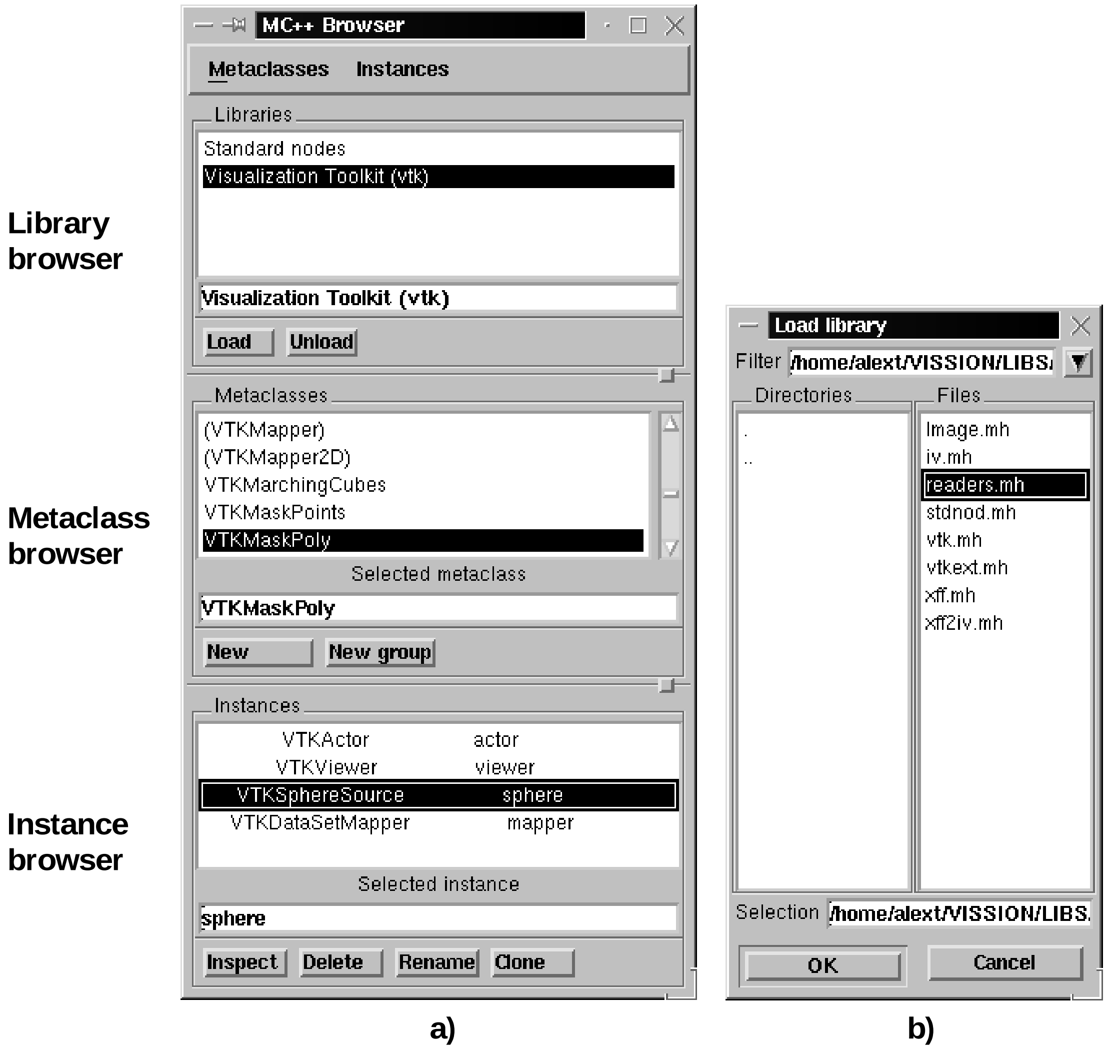
The network manager presented in the previous section provides a graphic interface to library browsing, component instantiation and deletion, cloning, reparenting and renaming, group navigation, and port-to-port connection editing. Application designers may sometimes however prefer a text interface for the above operations which may be faster e.g. for experienced users familiar with the names of the components present in application libraries.

The above and also the operation of loading and unloading component libraries are provided by the MC++ browser interface. The MC++ browser is a GUI consisting of three panels: the library browser, the metaclass browser, and the instance browser. Figure 4.7 a) shows the MC++ browser for the simple visualisation example discussed in the previous section.

The *library browser* panel shows that names of the loaded libraries, as they appear on their MC++ `library` declaration. The sphere visualisation example uses the ”Visualization Toolkit” library, which includes the ”Standard nodes” library. Both library names are shown in the library browser in Fig. 4.7 a). This panel allows also loading MC++ libraries by using a file selector widget. Figure 4.7 b) shows such a widget displaying several meta-libraries we have created for various application domains. Loading a library triggers the loading of all the included libraries, if these are not already present in the system. Unloading a library will unload all included libraries if these are not used any more, as described in section 3.6.

The *metaclass browser* panel allows browsing the names of all the metaclasses and meta-groups loaded in the system in several ways (alphabetically, by library, by category, etc). If desired, the non-instantiable metaclasses can be browsed here as well, displayed between brackets, such as `VTKMapper` and `VTKMapper2D` in Fig. 4.7 a. This option proves to be useful for component designers that need to check the correctness of the meta-libraries they design. A correctly written metaclass (instantiable or not) loaded by VISSION will appear in the metaclass browser, while an incorrect one will not. This panel allows also selecting metaclasses and metagroups and instantiating them, as well as creating empty groups.

The `instance browser` panel allows browsing the names of all the metaclass and meta-group instances present in the system. The browser in Fig. 4.7 a) shows the four metaclass instances of the sphere visualisation example. Instances can be selected and inspected by creating component interactors for them (see next section), or deleted, renamed, or cloned.



Libraries: Standard nodes; Visualization Toolkit (vtk)

Visualization Toolkit (vtk)

Metaclasses: (VTKMapper); (VTKMapper2D); VTKMarchingCubes; VTKMaskPoints; VTKMaskPoly

Selected metaclass: VTKMaskPoly

VTKActor	actor
VTKViewer	viewer
VTKSphereSource	sphere
VTKDataSetMapper	mapper

Selected instance: sphere

Filter: /home/alext/VISSION/LIBS/

Directories: . ; ..

Files: Image.mh; iv.mh; readers.mh; stdnod.mh; vtk.mh; vtkext.mh; xff.mh; xff2iv.mh

Selection: /home/alext/VISSION/LIBS.

Figure 4.7: The MC++ browser

4.3 End User Interface

The end user interface assists end users in the tasks of loading and starting existing simulations, controlling the simulation execution, editing various parameters of these simulations, and monitoring their results, in textual or graphical form. Similarly to the application design interface, the end user interface consists of two main GUIs that are designed to closely reflect the application structure. Besides these GUIs, a text-mode interface is provided for users more comfortable with the command-line interaction paradigm. These interfaces are presented in the following.

4.3.1 The Component Interactors

A VISSION application consists of a network of components that communicate with the outside world via their component interfaces, i.e. their ports and update operations. End users should be able to access to these *component interfaces* via a simple *component user interface*. Most component-based SimVis systems provide this in the form of component-specific GUIs that allow monitoring and editing the component interface by means of various widgets such as sliders, buttons, text fields, and so on. The same idea is used in VISSION. Every metaclass or meta-group instance in VISSION is automatically provided with a so called *interactor* . An interactor provides GUI widgets to inspect and edit all input and output ports of its component. In contrast to other systems, VISSION's component ports are not restricted to a fixed number of basic types, but can be of any user-defined C++ type (Section 3.3). VISSION's interactors were consequently designed to directly support editing of an open set of types, by an open set of custom GUI widgets.

Figure 4.8 illustrates the above for a metaclass `Example` (Fig. 4.8 a). The icon for an instance *object* of this metaclass in the network manager is shown in Fig. 4.8 b, and its GUI interactor is shown in Fig. 4.8 c. The structure of the GUI interactor follows the metaclass structure. The interactor has a section for the input port widgets and one for the output port widgets. These widgets are a numeric type-in, a slider, a file selection field, a 2-float field, a 3-float field, a toggle, a text display, and a numeric display, from top to bottom. These correspond to `Example`'s port types integer, float, character string, `Vec2f` (a 2-float vector), `Color` (a RGB colour triplet), boolean, character string, and double respectively. The widgets are labelled by their port names "port 1","port 2", and so on. The output port widgets are read-only, since the output ports can not be written to. Interactors let end users monitor component ports values as they are automatically updated whenever the ports' values change. Further, end users can also directly modify the input port values by using their widgets.

4.3.2 Text Interface

As discussed in the previous chapters, GUIs should not be the single interaction method offered by a SimVis system. Sometimes a 'classical' text-based command-line interface is more effective. Experienced users may work faster by, or simply be more comfortable with, typing the desired commands directly at a command-line prompt than by issuing them via several menu-driven GUIs. Many SimVis systems do therefore provide their functionality both in terms of GUIs and text-based interfaces.

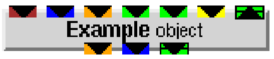
VISSION takes the same approach by providing a text-based command-line prompt at which all application designer and end user tasks can be carried away in terms of simple commands. Such commands can also be gathered in script files which can be loaded and executed by VISSION to perform the same actions.

An aspect where VISSION differs sensibly from other similar SimVis environments is the choice of its scripting language. Following the rationale of having a single language framework discussed in the previous chapter, VISSION's scripting language was chosen to be C++, its component development language. The commands the user can issue at VISSION's prompt (or write in script files to be executed) can thus be any valid C++ statements. These commands are executed dynamically (interpreted) by the system. The choice for C++ as scripting language leads to the following important advantages for the end user and application developer:

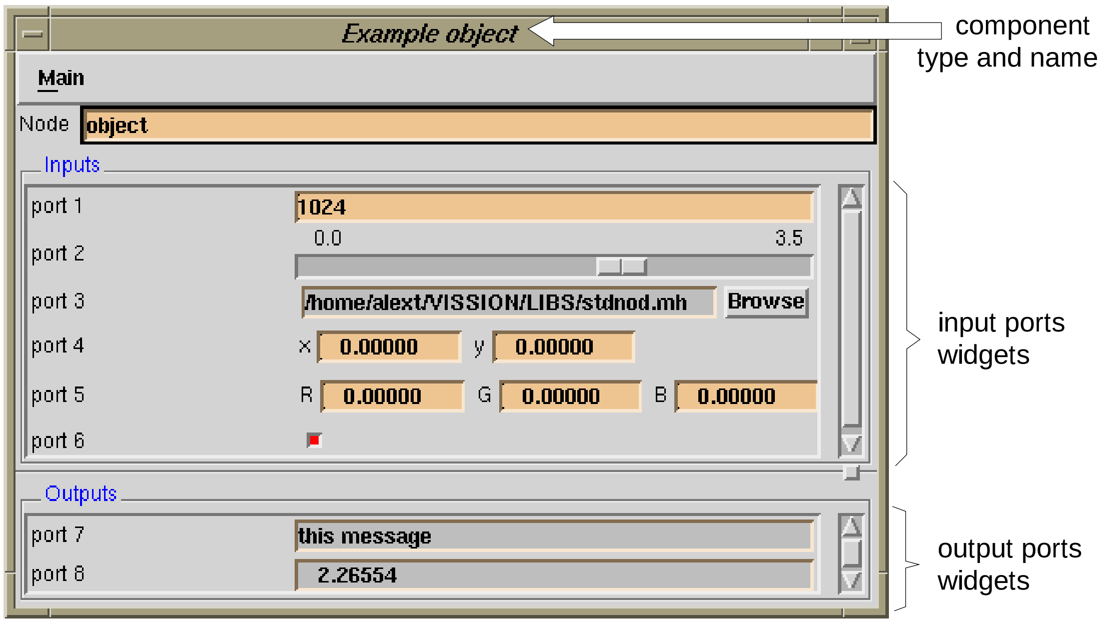
- **generality:** by using interpreted C++ as scripting language, one can describe arbitrarily complex scenarios. These can range from simple end-user commands issued on the fly at the system's prompt, such as the evaluation of a variable or the call of a method, to script files containing complex control structures or object manipulation. Such scripts are very useful when designing 'animation-like' scenarios where a given set of carefully planned operations is played back e.g. for presentation purposes. Infrequent users need thus only a basic knowledge of C++, similar to the knowledge they would need to learn any scripting language. Advanced users however are not confined to simple scripting commands, as it happens in many systems whose scripting languages are less powerful than their development ones.

```
module Example
{
input:
    WRPortMeth "port 1" "int"    (...)
    WRPortMeth "port 2" "float"  (...)
    WRPortMeth "port 3" "char*"  (...)
    WRPortMeth "port 4" "Vec2f"  (...)
    WRPortMeth "port 5" "Color"  (...)
    WRPortMeth "port 6" "BOOL"   (...)
output:
    RDPortMeth "port 7" "char*"  (...)
    RDPortMeth "port 8" "double" (...)
}
```

a)



b)



Node object

port 1 1024
port 2 0.0 3.5
port 3 /home/alext/VISSION/LIBS/stdnod.mh
port 4 x 0.00000 y 0.00000
port 5 R 0.00000 G 0.00000 B 0.00000
port 6

port 7 this message
port 8 2.26554

c)

Figure 4.8: Interactor example: metaclass (a), icon (b), and GUI interactor (c)

- **uniformity:** all user roles need to know a single language to communicate (besides the component developer who must use MC++ for writing the component meta-constructions). For example, a researcher might develop a C++ class library (or get an existing one), then write a test scenario C++ script, and then execute the script by loading it in VISSION. Next, the script can be easily re-edited and re-executed, or query commands on its results can be issued at VISSION's prompt. This scenario is typical for e.g. Matlab or Mathematica users or C++ users that needed to write and compile a main program to run a test. Such users can thus preserve their way of work.

- **ease of implementation:** from VISSION's design point of view, using C++ as a scripting language instead of another language is convenient. Indeed, VISSION needs dynamic C++ code loading and execution mechanisms for the implementation of its dataflow engine, as described in detail in chapter 5. Using the same mechanisms for providing C++ scripting capabilities is thus easier than implementing a new interpreter for a different scripting language.

4.4 Interactor Construction

As mentioned above, interactors are automatically constructed for any VISSION component. The widgets to be used in the interactor construction are inferred out of three factors (Fig. 4.9):

1. the port types from the component declaration
2. the available widgets stored in a widget database
3. the eventual user preferences

. These factors are described in the following.

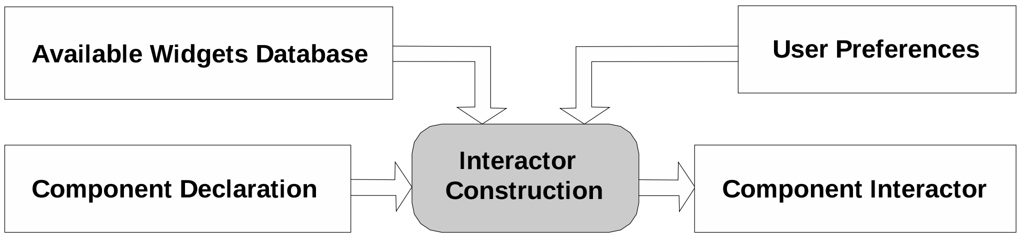


Figure 4.9: Interactor construction process

4.4.1 The Widget Database

Since VISSION accepts an open set of port types that belong to components defined in various meta-libraries, it has no built-in widgets. Instead, widgets are provided by the component developer as part of the meta-libraries he creates. When the components of a meta-library are loaded in VISSION, the associated widgets are loaded into a widget database maintained by the system and made available to the interactor construction algorithm presented further in section 5.2. In this way, component libraries can provide specific widgets for the specific port types of their components, and do not have to rely upon the existence of a fixed set of widgets hard-coded in the system.

MC++ library files specify the widgets they provide to VISSION in a widget specification section right after the implementation section (Section 3.6). This section is introduced by the MC++ keyword `gui` followed by a body between curly braces, as in the example in Fig. 4.10.

Widgets can be implemented by subclassing the C++ class `MOTIFWidget` which is hard-coded in VISSION. `MOTIFWidget` declares the widget communication protocol of new widgets with VISSION and offers to subclasses mechanisms to implement new widgets using the Xt [78] and Motif [33] GUI toolkits. The `MOTIFWidget` interface is detailed further in section 5.2. Widget development and integration in VISSION uses thus the white-box framework approach, since component developers have to follow the `MOTIFWidget` class interface for the new widgets they write. We could have implemented widget development based on a black box approach as well. However, the white-box approach based on inheritance was chosen for its simplicity of implementation and execution efficiency, and has proven to be convenient in practice.

The `gui` section starts with an `implementation` specifier that gives the name of the compiled library that contains the implementation of the GUI widgets, i.e. ”widgets.so” in the above example. This library has a similar structure to the libraries that contain the metaclass and meta-type implementations (Section 3.6). The main difference is that the widget classes have to inherit from the `MOTIFWidget` baseclass to comply with VISSION’s Motif-based GUI protocol.

```
gui
{
  implementation:   "widgets.so"

  widget       MOTIFEdGeneric              *      { "*" }
  widget       MOTIFEdSlider               write  { "int", "float", "double" }
  widget       MOTIFEdThumbWheel           write  { "float", "double"  }
  widget       MOTIFEdDial                 *      { "int", "float", "double" }
  widget       MOTIFEdToggle               *      { "BOOL"  }
  widget       MOTIFEdFileName             write  { "char*" }
  widget       MOTIFEdVec2f                *      { "Vec2f" }
  widget       MOTIFEdVec3f                *      { "Vec3f" }
  widget       MOTIFEdColor                *      { "Color" }
  widget       MOTIFEdColWheel             *      { "Color" }
  widget       MOTIFEdColSliders           *      { "Color" }
}
```

Figure 4.10: Widget specification section for meta-libraries

Next, the `gui` section contains the widget declarations. Each declaration starts with the `widget` keyword, followed by the name *name* of the `MOTIFWidget` subclass that implements the widget, and two other arguments arg_1 and arg_2:

`editor` *name* arg_1 $\{$ arg_2 $\}$

The first argument arg_1 may take the values `read`,`write`, or `*`, and specifies if the widget is to be used only for read ports, write ports, or both types of ports respectively (see section 3.3 for the port read/write attribute presentation). This argument is useful since some widgets, such as a thumb-wheel, have a meaningful utilisation only when they edit a read-only port, while others, such as a type-in, can be used for both read and write ports. The second argument arg_2 specifies the C++ types the widget can edit between curly braces. If the widget can edit any C++ type, the list contains a single element `"*"`.

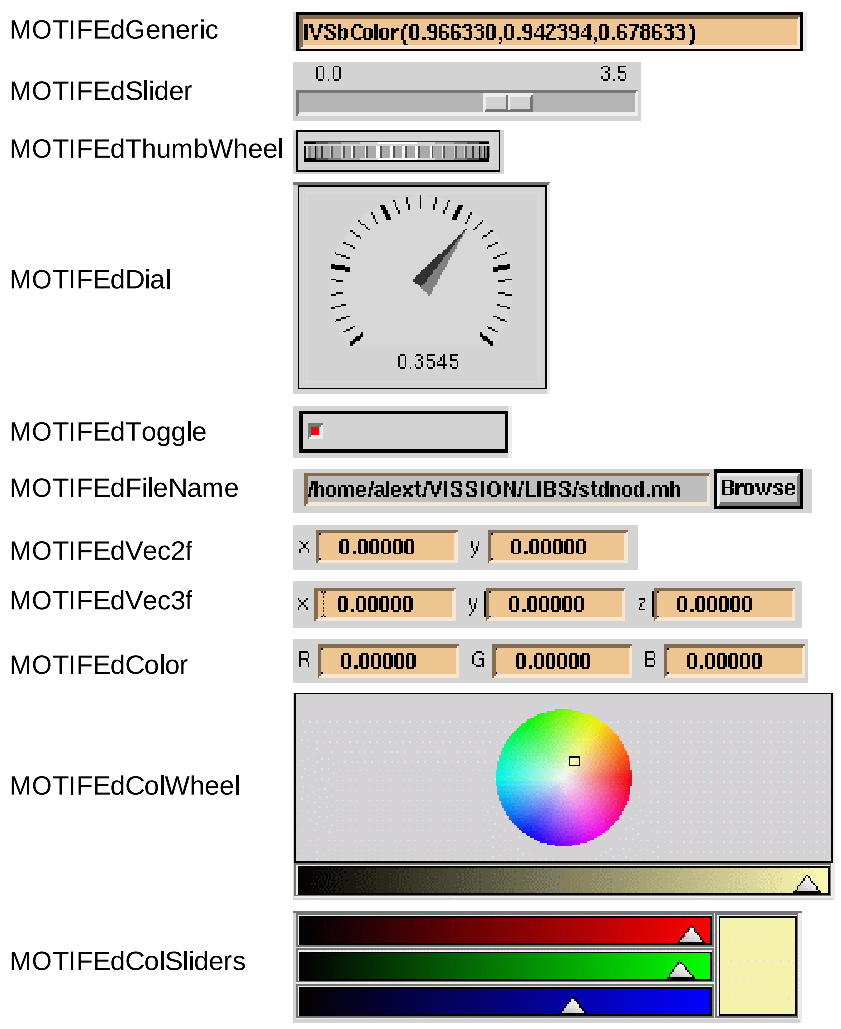
The GUI specification section example in Fig. 4.10 declares several widgets that can edit various C++ types, such as:

- basic numerical types: `MOTIFEdSlider` and `MOTIFEdDial`
- browsers for character string file names: `MOTIFEdFileName`
- user defined types such as: `Vec2f` (2-float vector), `Vec3f` (3-float vector), and `Color` (a RGB triplet)
- a generic widget for any C++ type (`MOTIFEdGeneric`)

The respective widgets, nearly identical to the ones provided by the interfaces of Open Inventor or AVS, are displayed in Fig. 4.11. The above widgets have been actually implemented by a meta-library which is included by most other application-specific meta-libraries developed for VISSION. Consequently, the widgets are loaded into VISSION's widget database and used by most component interactors. The mechanism that associates widgets with component ports is presented in the next section.

4.4.2 The Interactor Construction Algorithm

Interactors are automatically constructed by VISSION for any component. This is done by selecting for every component port the widget from the widget database which best matches the port's type.



IVSbColor(0.966330,0.942394,0.678633)

0.3545

/home/alext/VISSION/LIBS/stdnod.mh

x 0.00000 y 0.00000

x 0.00000 y 0.00000 z 0.00000

R 0.00000 G 0.00000 B 0.00000

Figure 4.11: Widget examples

The widget-to-type matching algorithm has four steps, as follows:

- *STEP 1:* all the widgets that can edit a C++ type T_1 compatible with the port's C++ type T_2 are selected and ordered in decreasing order of the distance between T_1 and T_2. The distance between C++ types is based on the type space metric defined in the C++ reference standard [30]. For example, a widget that can edit a class A can be used for a port of type B if B is a base of A or there is a user-defined conversion from A to B. If, however, a widget that edits B exists, it will be preferred to the one that edits an A since the B-B type distance is zero, while the B-A distance implies one derived-to-base conversion, so it is greater than zero.
- *STEP 2:* the widgets that match the port's read/write type are selected from the ones obtained in step 1.
- *STEP 3:* the widgets with the smallest distance in type space are selected from the ones obtained in step 2.
- *STEP 4:* if step 3 yields several widgets, the one which was first loaded in the widget database,

i.e. appeared first in the `gui` section, is used.

The algorithm executes steps 1-4 in order and stops as soon as the selection narrows down to a single widget.

Among the open set of widgets, the widget `MOTIFEdGeneric` is a special one. As mentioned in the previous section, this widget can edit any C++ type. `MOTIFEdGeneric` is actually a text type-in which edits any C++ port by reading and writing its *serialised value*, as defined in section 3.4. For example, the `MOTIFEdGeneric` widget in Fig. 4.11 edits the value of a `Color` port by displaying its serialised value in the form of a `Color` class constructor expression. `MOTIFEdGeneric` is declared to edit any C++ type for both read and write ports (Fig. 4.10). Consequently, this widget provides a 'fallback' that makes the widget selection algorithm always succeed to produce a widget for any port type. In most cases however, the algorithm's type matching rules will yield more specific widgets. The algorithm is applied dynamically whenever the end user requests a new component interactor (see Section 4.4.4). This practically means that the end user interface construction is fully automated.

4.4.3 Interface Preferences Specification

VISSION offers four ways for customising the component interactor construction, as follows:

1. write a new C++ widget subclass of `MOTIFWidget`
2. specify that a certain widget is to be used for a given port type, in the `gui` specification section
3. specify that a certain widget is to be used for a given metaclass or meta-group port, in the meta-class or meta-group declaration
4. specify that a certain widget is to be used for a given metaclass or meta-group instance, at run-time

The first two methods have been presented in the previous section. The other two are presented in the following.

Interactors can be customised also on a per-component type basis. For this, two constructs are added to MC++. First, the construct `editor` *name* can be added at the end of a metaclass port declaration in order to specify that the respective port should be edited by the widget *name*. If no such widget is found in the widget database or if the widget can't edit the port's C++ type, the interactor construction algorithm proceeds as described in the previous section. Secondly, the keyword `optional` can be added to the port declaration to specify that the respective port should not have any widget in its interactor. This is useful e.g. for ports that should only be modified or accessed by components via links but are of no interest to the end user. The same holds for ports which produce large, complex datasets, e.g. a whole mesh.

Secondly, the end user can change the widget that edits a certain port in the interactor itself. Every interactor widget has a popup menu in which all widgets able to edit that port, determined at step 2 in the widget selection algorithm, are displayed. End users can thus switch between several widgets in interactors by a simple mouse click. Figure 4.12 shows an example in which a port of type float is edited by four different widgets.

4.4.4 Reference Ports

Interactor construction treats reference ports specially since their values carry the additional semantics of *references* to other component ports. Reference ports widgets display thus references to other ports

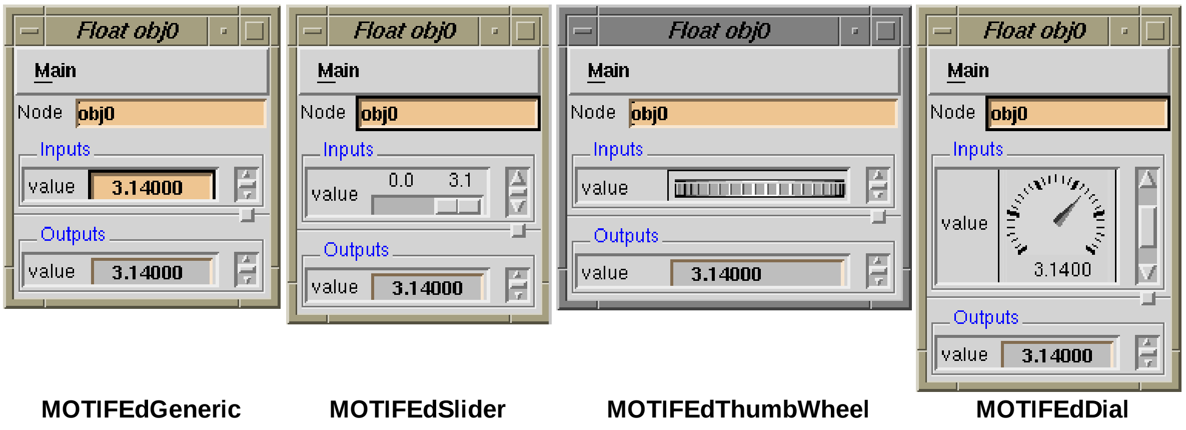


Figure 4.12: Several ways to edit a port of type `float`

instead of port values, using the reference port value syntax presented in section 3.3.4. In the example in Fig. 4.13 a, `VTKDataSetMapper`'s reference port "input" is not connected, so its widget displays the empty string. Figure 4.13 b shows the situation when this port is connected to component `obj0`'s port `output`, at position 0. The two ports can be connected or disconnected either by using the mouse-

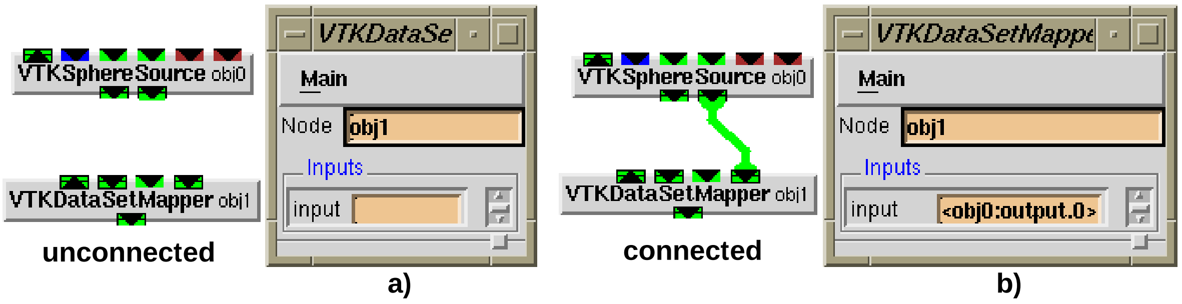


Figure 4.13: Reference port editor: unconnected (a) and connected (b) case

based network editor (Section 4.2.2) or alternatively by typing reference port values in the interactors. Dataflow network construction can thus be performed using the keyboard-driven interactor interface, besides the mouse operated network editor.

The Toplevel Interface

All GUI-based functions of VISSION are accessible via its *toplevel interface* . Using this interface, meta-libraries can be loaded and unloaded from the system, applications can be loaded from saved files, or the current state of the system can be saved to a file. Figure 4.14 illustrates a typical VISSION session for the visualisation example introduced in Section 4.2. The toplevel interface shown in the lower-left window displays icons that allow the access to other system interfaces, such as the network manager (shown in the background), the help manager, and the interactors for the four instantiated components (`VTKSphereSource`, `VTKDataSetMapper`, `VTKActor`, and `VTKViewer`). The interactors for `VTKSphereSource` and `VTKViewer` are visible in the two right windows. The icons are customised to display images that suggest the functionality of their components. Finally, the lower part of the toplevel interface shows various system messages in a text widget.

4.5 Conclusion

This chapter has presented the user interfaces offered by VISSION to the application designer and end user. The application representation concepts on which VISSION is based (meta-classes, components, dataflow networks, OO typing) are brought to a visual representation such that they can be interactively manipulated by the application designer and end user.

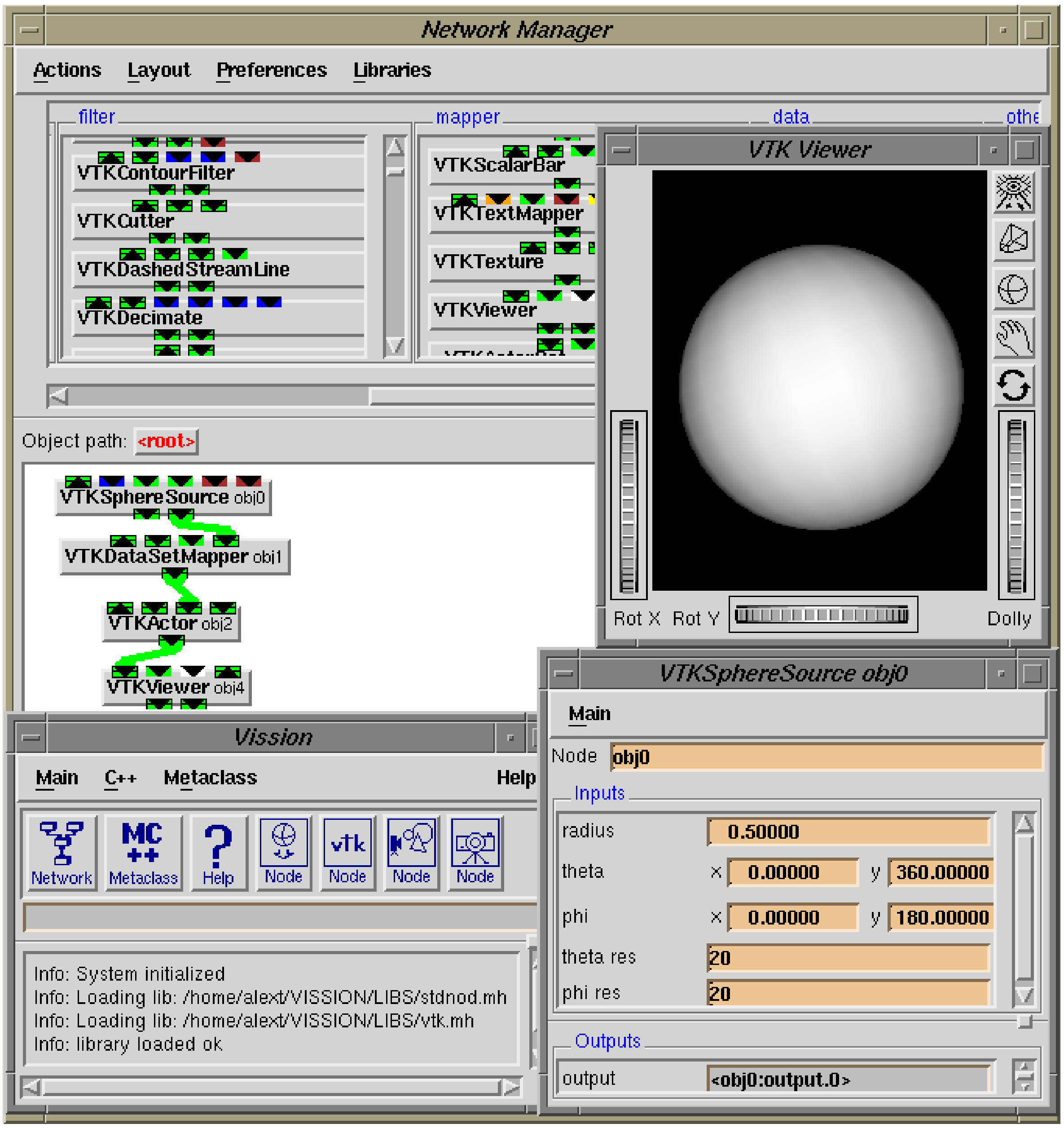
The visual programming metaphors presented here are as easy to use as the ones promoted by similar SimVis systems. The application designer and end user are not required to understand or even be aware of VISSION's object-oriented fundaments. For example, complex OO type checking and data conversions take place dynamically and automatically in the process of network construction. The application designer does need to focus only on the visual process of icon dragging and port to port connection.

A second advantage of VISSION's OO-dataflow foundation introduced in the previous chapter is the fully automatic construction of the application design and end user interface elements, i.e. the component icons and interactors. These interfaces directly reflect the structure of the component libraries, which are in their turn derived from the domain analysis and modelling. Consequently, the user interfaces exhibit a certain consistency which makes them simple to understand and use. Since VISSION's user interface is practically provided for free for new components, the developer-designer-end user role transition is a simple, automatic process. This visibly contrasts with most SimVis systems in which user interface creation is a major time consuming, usually very technical process.

Thirdly, the fact that VISSION provides a generic and open user interface for any types of components does not back propagate to extra requirements or restrictions for the component development. The component developer can, but is not obliged to, design custom GUI widgets for his components. These widgets exploit fully VISSION's OO structure and are maintained separately from the component implementation code such that the latter continues to be system independent. Similarly to component libraries, widget libraries can be developed and then directly plugged in the system to be used with existing components without any need to recode or relink the two.

Finally, VISSION also provides the possibility to construct GUIs for whole applications. This is simply done by enclosing the whole dataflow network in a group and exporting to the outside all ports of the various modules that the end user desires to interact with. The dataflow structure is invisible to the end user, who interacts with a single GUI instead of the per-component interactors. In this way, one has the same look-and-feel as given by the customised GUI of a turn-key application.

The common denominator of the above features is the existence of the OO-dataflow combination on which VISSION is founded. The next question is how this combination is technically implemented in VISSION. This is the subject of the next chapter.



Object path: <root>

Info: System initialized
Info: Loading lib: /home/alext/VISSION/LIBS/stdnod.mh
Info: Loading lib: /home/alext/VISSION/LIBS/vtk.mh
Info: library loaded ok

Node	obj0			
radius	0.50000			
theta	x	0.00000	y	360.00000
phi	x	0.00000	y	180.00000
theta res	20			
phi res	20			
output	<obj0:output.0>			

Figure 4.14: VISSION session with toplevel, network manager, and component interfaces

Chapter 5

Design and Implementation of VISSION

Chapter 3 has described VISSION's new architectural model. The key element of this architecture is a high-level framework-component interface, which combines object-oriented and dataflow modelling in a non-intrusive manner. In chapter 4, we described the several user interfaces of VISSION. These interfaces map VISSION's modelling concepts (metaclasses, components, dataflow networks, OO typing) to the visual representations of component icons and GUI interactors.

This chapter describes the design and implementation of VISSION. It is divided in two main parts. Section 5.1 describes the *kernel* of VISSION which implements the framework-component interface in terms of the MC++ and C++ languages. This explains how the functionality presented in Chapter 3 is realized. Section 5.2 describes how the several user interfaces, presented in Chapter 4, are implemented atop of the kernel.

5.1 The Kernel

VISSION is built on a variant of the client-server architecture known as the model-view-controller (MVC)[56], or the Observer-Observable design pattern [37], similarly to many other SimVis environments [113, 117, 44]. This design contains two parts (Fig. 5.1): the system *kernel*, described in this section, and several *views*, which are the object of Section 5.2.

The kernel acts as a server which provides services in response to messages received from several clients (or views), via the *kernel-view (KV) interface*. The views implement interaction interfaces for the application developer or end user, such as the network manager, the GUI interactors, the text interface, and the toplevel interface presented in chapter 4. New views could be created as an extension or alternative to the existing ones, as long as they comply with the KV interface. The decoupled design of the views and the kernel makes the implementation, maintenance, and extension of the two easier than a monolithic structure. In this respect, both the application domain components written in MC++ and the views are software *components* that connect to the kernel *framework* (via different interfaces, however).

The kernel services offered via the KV interface are ultimately delegated to application specific components, via the *kernel-component (KC) interface* introduced in Section 3.2.2. The remaining of this section describes the kernel and its KC interface in detail.

5.1.1 The Kernel-Component Interface

As presented in Section 3.2.2 (see e.g. Fig 3.13), the role of the KC interface is to:

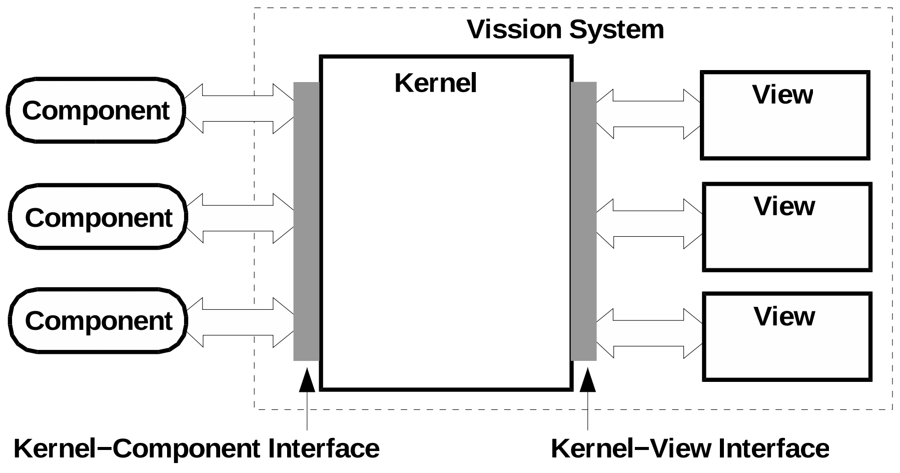


Figure 5.1: VISSION client-server architecture

1. directly represent all possible value types supported by component interfaces;
2. load and unload component libraries;
3. instantiate and destroy components;
4. read and write component port values;
5. connect and disconnect component ports;
6. ask components to update.

We have already explained in Section 3.3 how the above operations are modelled by the combination of MC++ and C++ languages. Components are modelled by MC++ metaclasses and meta-groups, which are ultimately implemented by C++ classes. Component inputs and outputs are implemented as C++ value types. Inter-component data transfer and component update map to C++ method calls or data member evaluations. In implementation terms, the above operations of the KC interface are equivalent to providing the following C++-related mechanisms:

1. support all possible C++ value types;
2. support C++ function call and data member evaluation;
3. support dynamic loading and unloading of C++ classes.

Another requirement is that arbitrary C++ can be executed dynamically, following from the text user interface requirement (Section 4.3.2). Together, all these requirements strongly suggest to base the kernel implementation on a *C++ virtual machine* (or C++ interpreter).

It is important at this point to find out if other implementation alternatives may exist for the kernel of VISSION. Any chosen implementation must, in any case, provide the three C++-related features mentioned above. Assuming one had not chosen to use a C++ interpreter, the following should have been implemented:

1. a type table mechanism for the C++ type system (fundamental types, structures, classes, type conformance, etc). This is a complex task once OO features such as derived classes, multiple inheritance, etc, are to be represented.

2. a simple parser and interpreter for member evaluation and C++ function call. Implementing such a parser is rather simple. Implementing the actual execution of the parsed actions is however complex, as function evaluations could use by-reference or by-value data passing of class types involving chains of constructor and destructor calls.

3. a dynamic C++ library loader and a parser for 'headers' containing the declarations of the compiled code's symbols. Implementing a dynamic loader is reasonably simple. Implementing a declaration parser is however one of the most complex parts of C++ language analysis [100].

. Overall, it is clearly advantageous to reuse an existing C++ virtual machine for our purposes. The integration of such a C++ virtual machine in VISSION's kernel is presented in the following.

5.1.2 The C++ Virtual Machine

VISSION's kernel is built on the top of an existing C++ interpreter, called CINT, originally developed at Hewlett-Packard Japan [15]. CINT is a C/C++ interpreter coming both as a C++ library callable for client code and a standalone program that interprets source files, similar to most interpreters. CINT is used by several large software systems. For example, the ROOT system [15] is a generic framework for scientific data analysis, postprocessing, and visualisation which uses CINT to dynamically load, parse, and interpret C and C++ code.

CINT offers the following services:

- dynamic loading and unloading of compiled or source-level C++ code. Loading code makes all involved C++ symbols (functions, classes, variables, typedefs, types, macro definitions, and so on) available to CINT, while unloading relinquishes the code and names of the loaded symbols.

- dynamic C++ code execution. CINT can load arbitrary C++ source code and execute it dynamically.

- reflection API. CINT offers a reflection API by which all loaded symbols can be examined. For example, one can find out which are the methods of a given class, what are their types, or which are the bases of a class.

- C++ expression evaluation. CINT can evaluate any valid C++ expression and return its value. The returned values can be examined further via the reflection API, whether they are basic data types such as `int` or `float`, or complex structured types.

CINT is a complex piece of machinery of approximately 80,000 lines of C/C++ code, containing a C++ language parser, a dynamic library loader, the usual class and object tables present in most interpreters and compilers, and a bytecode compiler for loop optimisation. Its reasonably simple API and its very fast C++ code loading and execution render it ideal for the dynamic demands of the VISSION kernel. Since CINT comes as a client-callable library, it can be easily embedded into VISSION's kernel, the latter maintaining the control. This is not the case with many interpreters or virtual machines, which are stand alone, independently running programs.

5.1.3 Kernel Structure

VISSION's kernel consists of four main components implemented in C++ (the names of the C++ implementation classes shown also in Fig. 5.2 are given between brackets): the C++ interpreter

(`Interpreter`), the MC++ manager (`MetaCPPManager`), the library manager (`Library-Manager`), and the dataflow manager (`DataflowManager`). The KC interface has eleven main operations or commands, as follows: **library_load**, **library_unload**, **new_component**, **del_component**, **port_read**, **port_write**, **connect**, **disconnect**, **update**, **file_load**, and file_save. These operations are implemented by the `Kernel` class which manages the `Interpreter`, `MetaCPPManager`, `LibraryManager`, and `DataflowManager` classes. The first nine operations have been already

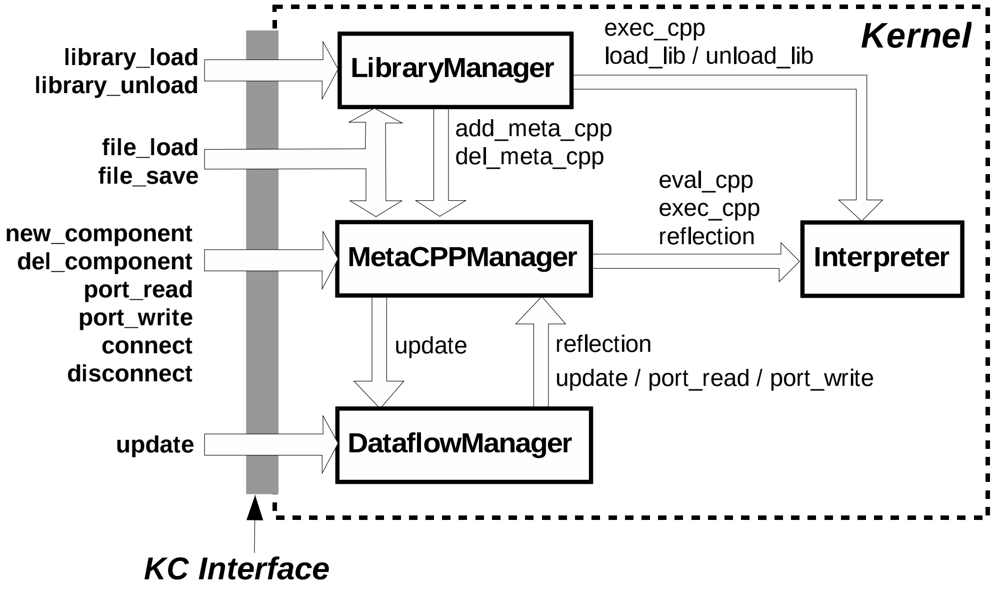


Figure 5.2: Kernel structure

introduced in Section 5.1.1. The last operation (**file_save**) saves the state of the kernel as a C++ script file. This file can be loaded to restore the current state, via the **file_load** operation.

The KC interface operations are implemented by the `Interpreter`, `MetaCPPManager`, `LibraryManager`, and `DataflowManager` subsystems. These are described next, in a bottom-up fashion.

5.1.4 The `Interpreter` Subsystem

The `Interpreter` subsystem is a C++ wrapper around the CINT virtual machine described earlier in this chapter. Its presence serves three purposes:

- **encapsulation:** the `Interpreter` subsystem hides the CINT API from VISSION's kernel via a delegation layer. Given another (e.g. faster) C++ virtual machine that complies with the `Interpreter` interface, it can be easily used instead of CINT. Alternatively, if CINT's API changes, the changes to be done to VISSION's kernel are localised to the implementation of the `Interpreter` subsystem. The class Adapter design pattern proved useful here.

- **enhancement:** CINT's API does not directly provide all functionality the kernel needs. For example, it does not let one determine if two C++ types are compatible, which is needed to find out if two metaclass ports are connectable or not (see Section 3.3). This functionality has been implemented in the `Interpreter` class, by using CINT's reflection API, as described below. The Decorator pattern proved useful here.

- **simplicity:** CINT's API contains various complex constructions making it hard to use and understand, being what it is usually called a *fat interface* [102]. By hiding it under the minimal `Interpreter` interface (using the Facade pattern), it is much easier to understand and manage all its interactions with the rest of the kernel.

The `Interpreter`'s API (Fig. 5.2) provides the five operations introduced in Section 5.1.2: loading and unloading a C++ compiled library or source file (**load_lib**,**unload_lib**), executing and evaluating C++ source code and expressions passed as a text buffer (**exec_cpp**,**eval_cpp**), and enhancing CINT's reflection API with a couple of operations (**reflection**).

The above `Interpreter` interface actually defines the virtual machine concept in the context of VISSION. Given any language L and a virtual machine for L that provides the above five operations, one could base the whole VISSION system on that language, simply by providing the appropriate `Interpreter` interface which translates VISSION's generic operations in the specific syntax of L. In this way, we could implement VISSION based on Java or Objective-C, for example. The language-independent design of VISSION strongly differs from the design of other similar interactive visualisation systems or application builders, which are designed in strict relationship with a given programming language.

As mentioned above, CINT does not offer a way to determine if two C++ types are compatible or not. However, its reflection API lets one determine the kind of a type (fundamental or class), the base classes, if any, and their methods, in terms of full signatures. Using this information, `Interpreter` implements the `reflection` operation that determines whether two types are compatible by checking all the complex C++ type conversion rules (direct, trivial, derived to base, constructor based, and conversion operator based) [102]. A second important enhancement provided by `Interpreter` to CINT's API is to determine whether a given (class) type is instantiable or not, needed to determine the instantiability of a metaclass (Section 3.3.5).

5.1.5 The `MetaCPPManager` Subsystem

The `MetaCPPManager` subsystem manages the MC++ entities in the kernel, i.e. the metaclasses, meta-groups, meta-types (collectively called meta-entities), and the instances of the first two. `MetaCPPManager` has two subcomponents (Fig. 5.3):

1. an **entity table** containing all meta-entities present in the kernel
2. an **instance table** containing all metaclass and meta-group instances

This subsystem provides the following operations:

1. **add_meta_cpp**: add a new meta-entity to the entity table
2. **del_meta_cpp**: delete a meta-entity from the entity table and all its instances from the instance table
3. **new_component**: given a metaclass or meta-group `T` and a name for the new component (e.g. `obj`, see Fig. 5.4 a), create a component of type `T`, if `T` instantiable, and add it to the instance table (2). The constructor for the appropriate C++ class (i.e. class `T`) is called dynamically, by sending the code fragment `"new T"` to the `eval_cpp` operation of `Interpreter` (3). The returned pointer `ptr` to a new `T` C++ object (4) is then stored in the metaclass instance `obj` (5) for later use (Fig. 5.4 b).

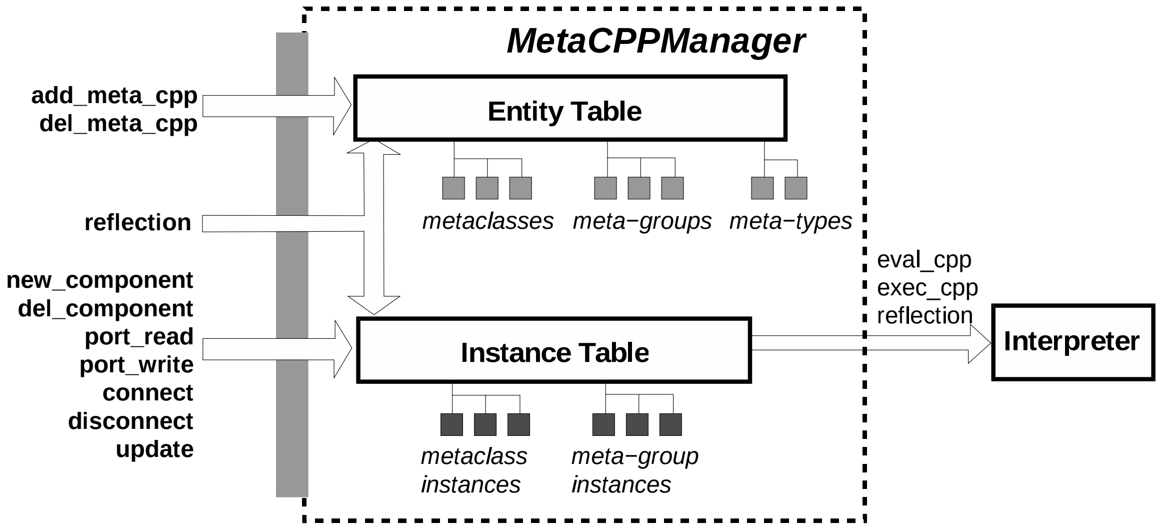


Figure 5.3: `MetaCPPManager` structure

4. **del_component**: deletes a component from the instance table, given the component name (e.g. `obj`, see Fig. 5.5). The component's C++ object `*ptr` is dynamically destroyed (2). After the C++ object is deleted (3), the metaclass instance is removed from the instance table and deleted as well (4).

5. **update**: calls a component's update operation, if any. The update operation, whether delegated to a C++ class method or given as source text (Section 3.3), is sent to the `Interpreter`.

6. **port_read,port_write**: reads, respectively writes data from, respectively to a given component port

7. **connect,disconnect**: connects, respectively disconnects two given ports. The `reflection` services of `Interpreter` presented in the previous section are used here to determine the port type compatibility.

8. **reflection**: offers services to browse the instance and object table, inquire about the features and bases of a metaclass or meta-group, etc.

`MetaCPPManager` is very similar to an interpreter. In fact, it can be seen as an interpreter for the MC++ language that consists of the metaclass, meta-group, and meta-type data structures, and the eight operations presented above. These operations are specified either on the own instance and entity tables, or delegated further to the `Interpreter` subsystem described previously. The structural relation between `MetaCPPManager` and `Interpreter` is in this sense similar to the relation between MC++ and C++.

The `read_value` and `write_value` operations deserve special attention. The arguments of these operations are string-encoded *port values*, as introduced in Section 3.4. In the case of by-value ports, `MetaCPPManager` simply delegates the reading or writing to the `Interpreter` subsystem's `eval_cpp` operation, by sending for execution code fragments similar to the ones shown in the network example in Section 3.3. These operations are called on the component's C++ object `*ptr` (Fig. 5.4 b). For by-reference ports, `MetaCPPManager` translates between the reference port value string syntax and its internal representation. When a reference port value is written into a reference

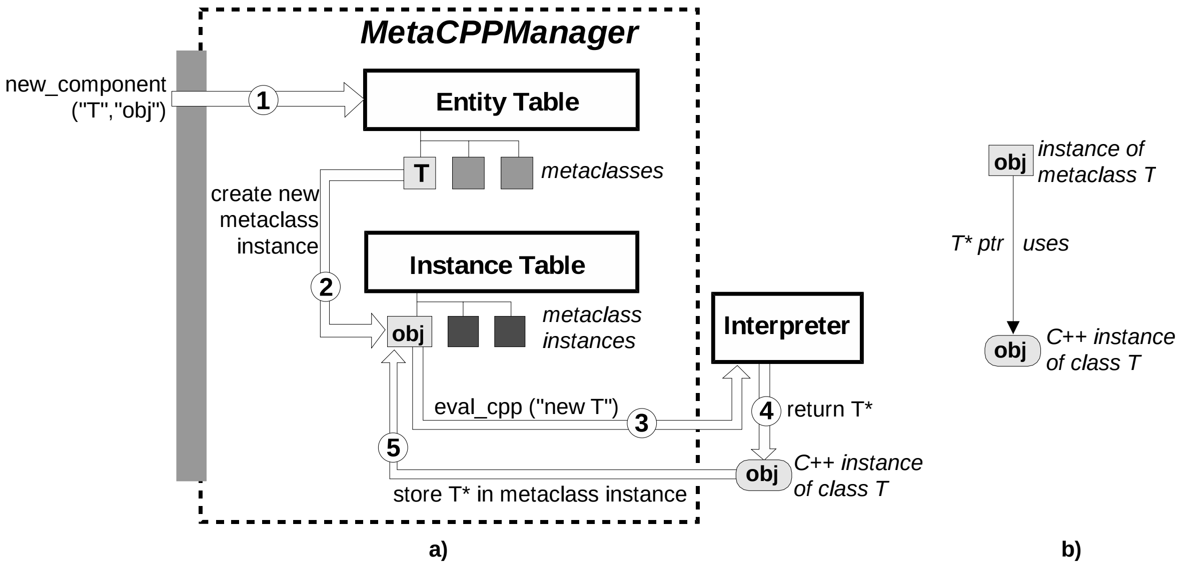


Figure 5.4: Implementation of the `new_component` operation

write port, the ports get connected as by a `connect` operation. If the written reference port value is 'nil' (i.e. the empty string), the write port gets disconnected from whatever it was connected to, as by a `disconnect` operation. The above represents the implementation of the mechanism discussed in Section 4.4.4.

In the above only the most important `MetaCPPManager` operations were described. This subsystem provides also other operations, such as adding and removing a child from a group, creating an empty group, renaming a component instance. As these operations do not pose any complex issues, they are not described here.

5.1.6 The `LibraryManager` Subsystem

The `LibraryManager` subsystem manages the kernel's interaction with the *meta-libraries*. As described in Section 3.6, a meta-library consists of two parts: a C++ compiled dynamic loadable library and a MC++ file with meta-entities implemented by the C++ library. The `LibraryManager` has three subcomponents (Fig. 5.6):

1. an **MC++ parser** that encodes the MC++ language grammar. The parser takes as input an MC++ source file and outputs the parsed MC++ language elements (metaclasses, meta-groups, meta-types, include statements, initialisation code, etc).

2. a **library dependency graph** that maintains the dependencies between meta-libraries. The graph nodes are the loaded meta-libraries, while its arcs represent the inclusion relationships between libraries (if `lib2` includes a `lib1`, there is an arc from `lib2` to `lib1`). The usage of this graph will be explained below.

3. a **widget table** that maintains the executable code of the `MOTIFWidget` subclasses loaded dynamically with the meta-library (see Section 4.4).

The `LibraryManager` provides the following operations:

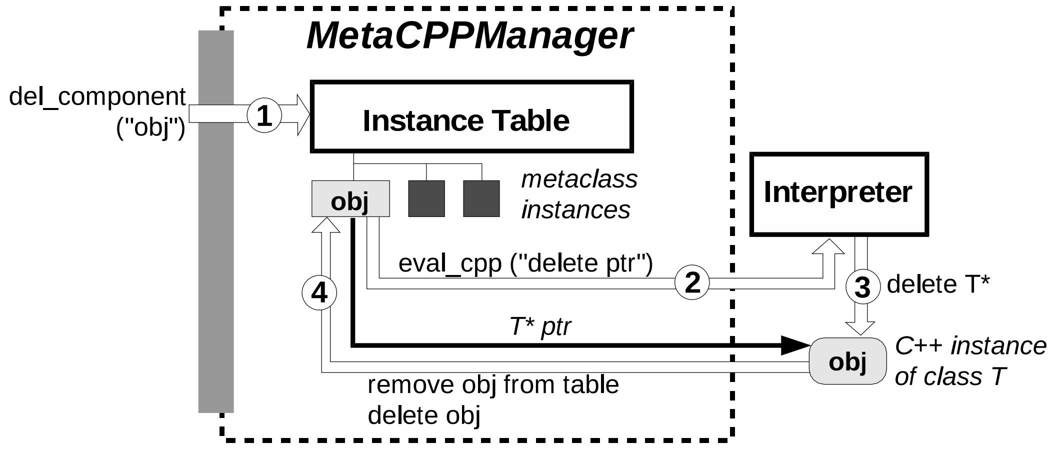


Figure 5.5: Implementation of the `del_component` operation

1. **library_load:** Given a meta-library `lib.mh` to be loaded, all libraries included by it are fetched from its `include` MC++ statements. If these libraries are not present in the dependency graph, they are loaded by issuing (possibly recursively) **library_load** commands, and the graph is updated correspondingly by the **add_lib** internal command. Next, the `Interpreter` loads the library's implementation (if any) via its **load_lib** command. All parsed meta-entities are added to the `MetaCPPManager`'s entity table by issuing **add_meta_cpp** commands. If the library contains widget specification sections (see Section 4.4), the declared GUI widgets and their compiled implementations are loaded into the `WidgetTable` by the `add_widget` command. Finally, the library's initialisation code (if any) is passed to the `Interpreter`'s **exec_cpp** command.

2. **library_unload:** unloads a loaded library. Given a meta-library to be unloaded, all meta-entities it contains are unloaded from the `MetaCPPManager` via **del_meta_cpp** commands. If the library declared any widgets, their declarations and executable code is unloaded via `del_widgets command` from the `WidgetTable`. Next, the library's finalisation code is passed to `Interpreter`'s **exec_cpp** command. All libraries the current library included are unloaded if they are used just by the library undergoing unloading. The dependency graph is then updated via its **remove_lib** command.

The `file_load` and `file_save` Operations

As section 5.1.3 outlined, the KC interface provides the **file_save** and **file_load** commands to save the system's state to a file, respectively load a saved state. These commands are implemented together by `MetaCPPManager` and `LibraryManager` (see Fig. 5.2), as discussed next.

The *state* of a dataflow system is the state of every component of the dataflow network plus the network topology, i.e. the network components and the links that connect them. In VISSION, component *write ports* serve both for modifying the state of the components and for connecting them (Section 3.3).

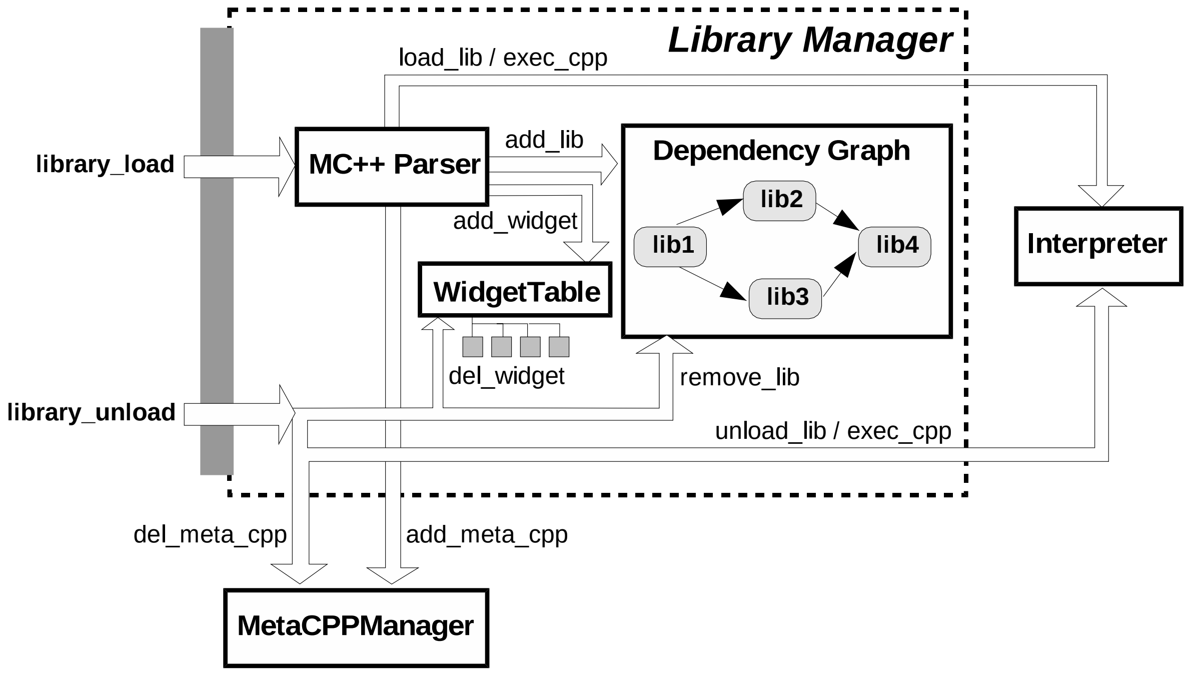


Figure 5.6: Structure of the `LibraryManager`

The state of a VISSION application is thus given by the port values of all component write ports, plus the existence of the component instances themselves. Saving the current state means thus saving:

1. the port values for all *modified* write ports, i.e. ports which were at least once written to during their lifetime. Saving port values for unmodified ports is redundant.
2. the existence of all component instances, i.e. their names and types
3. the *minimal set* of loaded meta-libraries, i.e. those libraries which (transitively) include all the loaded libraries. For example, the minimal set of the meta-libraries `lib1`,`lib2`,`lib3`, and `lib4` shown in Fig. 5.6 is just `lib4`, since `lib1`,`lib2`, and `lib3` are included transitively from `lib4`. Saving just the commands to load the minimal set of libraries eliminates redundancy.

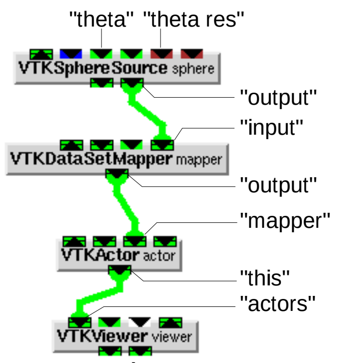
Given the above, loading a saved state means executing the above actions in opposite order:

1. all saved libraries are loaded via **library_load** commands
2. all saved components are instantiated with their saved names via **new_component** commands
3. all saved write port values are written back into their ports via **port_write** commands

We have chosen C++ to be the save file language. This complies once more with our single language desiderate. To implement the load and save operations, we extend the `SYSTEM` C++ class with three static methods `loadLibrary`, `newComponent`, and `portWrite` which simply delegate to the KC interface. The **file_save** operation generates a C++ source file containing the above commands. The **file_load** operation feeds the save file to `Interpreter`'s **exec_cpp** operation. There is thus no distinction between VISSION save files and VISSION scripts (see Section 4.3.2). Both are plain C++ source files in which `SYSTEM` class commands can be freely mixed with other C++ operations to produce the desired effect, e.g. a playback scenario or an animation involving arbitrary sequences of component creation, deletion, connection, or port value variations.

Example

We illustrate the above mechanism for the visualisation network discussed in Section 4.2.1 (see Fig. 5.7 a). Its save file is shown in Fig. 5.7 b. The file starts with the meta-library ”vtk.mh”. The four component instances follow. Finally, the modified write ports and their values are listed. Port values are written in the port value syntax by using the meta-type serialisation operation (Section 3.4). For instance, port ”actors” of `viewer` in the above example has the value ”¡actor:this.0¿” representing a reference to the ”this” port of `actor`. Port ”theta res” of `sphere` has the C++ integer value 20, while port ”theta” of type `Vec2f` has the value ”Vec2f(0,180)” which denotes a 2-float vector by calling class `Vec2f`’s two-argument constructor.



a)

```
SYSTEM::load("vtk.mh");

SYSTEM::newComponent("VTKSphereSource","sphere");
SYSTEM::newComponent("VTKDataSetMapper","mapper");
SYSTEM::newComponent("VTKActor","actor");
SYSTEM::newComponent("VTKViewer","viewer");

SYSTEM::portWrite("mapper.input","<sphere:output.0>");
SYSTEM::portWrite("actor.mapper","<mapper:output.0>");
SYSTEM::portWrite("viewer.actors","<actor:this.0>");
SYSTEM::portWrite("sphere.theta","Vec2f(0,180)");
SYSTEM::portWrite("sphere.theta res","20");
```

b)

Figure 5.7: Simple visualisation network (a) and its save file (b)

5.1.7 The `DataflowManager` Subsystem

The `DataflowManager` manages the updating of the dataflow network. The dataflow network is implicitly represented by the directed acyclic graph formed by the component instances of `MetaCPP-Manager`’s instance table and their port-to-port connections established by the **connect** command of the latter.

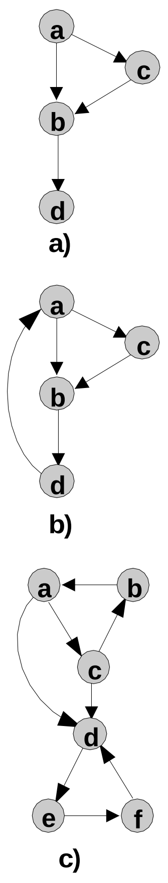
The `DataflowManager` provides a single operation: **update**. Given a metaclass or meta-group instance, **update** performs two actions:

1. updates the component itself by calling `MetaCPPManager`’s `update` operation. As described previously, this calls the component’s own update operation which, depending on its definition, executes some actions on the component’s C++ object and marks some of the output ports as changed (Section 3.3).

2. after the component is updated, data is transferred from its modified outputs to all connected component inputs, via `MetaCPPManager`’s **port_read** and **port_write** operations. Next, a dataflow network traversal is performed to update all components which (recursively) depend on the initially modified component.

The `DataflowManager` implements thus the event-driven update model described in Section 2.2. In order to preserve the invariant that the dataflow network is always up to date, the **update** operation of `DataflowManager` is called automatically by `MetaCPPManager` whenever a **port_write** operation occurred. Consequently, the `DataflowManager` and the `MetaCPPManager` call each other cyclically (Fig. 5.2) until all existing components are up to date.

Factoring the dataflow network traversal and update in the separate `DataflowManager` component has the important advantage of separating the *intrinsic* (component) update mechanism (provided by `MetaCPPManager`) from the *extrinsic* (network) one (provided by `DataflowManager`). Several network traversal mechanisms can thus be coded as `DataflowManager` subclasses without having to change anything else in the kernel.



```
Traverse(NODE src)
{
  NODE new; QUEUE open;
  put src in open;
  mark all nodes as undiscovered;

  do
  {
    if (all nodes in open discovered)
      new = youngest node in open;
    else
      new = first node in open;

    remove new from open;
    make new oldest node and mark it discovered;

    if (new has at least one modified input)
    {
      update new;

      for_each(NODE s = successor of new)
      {
        if (s undiscovered)
        {
          if (s leads to no node in open)
           put s in open;
          else
           replace first node in open to whom s leads by s;
        }
      }
    }
  } while (open not empty)
}
```

d)

Figure 5.8: Cyclic and acyclic dataflow networks (a,b,c). Cyclic network traversal algorithm (d)

Most graph traversal engines implemented by dataflow systems such as AVS [113], Inventor [116], Java3D [104], or Explorer [44] use topological sorting of the graph nodes, as described by e.g. Cormen et al in [22]. Topological sorting has the advantage that a node is updated only when all nodes it depends on have been updated. For example, the traversal of the network in Fig. 5.8 a) starting from a would be done in the order $a-c-b-d$. However, such an engine can not be used to manage dataflow networks with loops, as they appear in the numerical iterative simulations or direct manipulation networks presented in Sections 6.2.4 , 6.8 , 7.4. For example, the traversal of the network in Fig. 5.8 b) starting from node a would not lead to the (possibly infinite) sequence $a-c-b-d-a-c-b-d-...$ as one would expect in practice, but would fail producing any traversal, as cyclic graphs do not support topological ordering.

We have hence designed a `DataflowManager` engine which can traverse any (cyclic or acyclic) dataflow network to perform its update. This engine is based on the following rules:

1. graph nodes can be either discovered or undiscovered. At the beginning of the traversal, all nodes are undiscovered.

2. every discovered node has an age, which may only increase during the traversal;

3. the ages of the of nodes encountered during the traversal form a strictly increasing sequence;

4. the algorithm attempts to maintain the topological order property *locally*, if possible. For example, when traversing the two successors b, c of node a in the graph of Fig. 5.8 b), c should be traversed *before* b since b can be reached from c.

The `DataflowManager` traversal algorithm pseudocode is shown in Fig. 5.8 d. In the pseudocode, `open` is a queue of nodes supporting operations such as insertion, deletion, and iteration over its elements. Practically, the algorithm combines a least-recently-used (LRU) policy [22] to iterate over nodes (rules 2,3) with the attempt to keep topological order (rule 4). The LRU policy ensures that loops are handled properly, while the topological sort ensures that nodes are not traversed uselessly. For example, the traversal of the graph in Fig. 5.8 b) from node a yields the infinite sequence $a - c - b - d - a - c - b - d - ...$, where c is traversed before b (topological order), and d is followed by a (LRU closing the loop). Traversing the network in Fig. 5.8 c) from node a would lead the sequence $a - c - b - d - e - f - a - c - b - d - e - f...$, where c is traversed before d (topological sort), and d (and not a) follows b (LRU). The overall effect is the one ’naturally’ anticipated by the application designer, i.e. of having two loops running ’in parallel’.

The above `DataflowManager` engine has proven very useful in practice, as cyclic networks occur in many application domains, e.g. numerical iterative processes, animations, data presentation, movie creation, etc.

5.2 The Views

As outlined at the beginning of this chapter, VISSION consists of a kernel that interacts with its various users via several *views* . A view is a software subsystem that presents to its users a specialised interface to the kernel functionality. For example, an application design view provides a GUI editor for building the dataflow network from iconic component representations. The views discussed in this section are `NetworkManager`, which implements the GUI discussed in Section 4.2.2, the `MC++Browser`, which implements the GUI discussed in Section 4.2.3, and `InteractorManager`, which implements the component interactors discussed in Section 4.3.1.

Views interact with the kernel in three ways: send commands to the kernel, read and write various data from, respectively to the kernel, and listen for events sent by the kernel. The first two operations are accomplished via the kernel-component interface already described in the previous sections and accounts for the Controller aspect of the views and the Model aspect of the kernel. The last operation is done via a separate kernel-view (KV) interface, which implements the View aspect of the MVC model. The following presents the KV interface, as well as the most important views.

The Kernel-View Interface

VISSION's kernel implements several services and data structures which are demanded (called), respectively read (viewed) by the views. The execution of a kernel command changes however several of its data structures which may be monitored by several views. To maintain consistency, the kernel informs the views of the changes it has undergone, so they can update themselves, via an event mechanism.

The kernel declares a set of events which signal the various changes that can take place to the information read by the views. For example, the `NEW_COMPONENT` event signals that a **new_component** command has created a new component, hence the `InstanceTable` has changed. The other KC interface commands that modify the kernel visible state generate similar events, such as `DEL_COMPONENT`, `CONNECT`, `PORT_WRITE`, and so on. An event can have a set of *arguments*, which carry additional information. For example, the `NEW_COMPONENT` event carries two strings denoting the metaclass or meta-group to instantiate and the new component's name respectively.

A view can register for interest in several events. When an event is emitted, the kernel signals the interested views by calling a notification callback they provide and passing the event as argument. The views can update themselves by reading the kernel data structures whose change was signalled by the event.

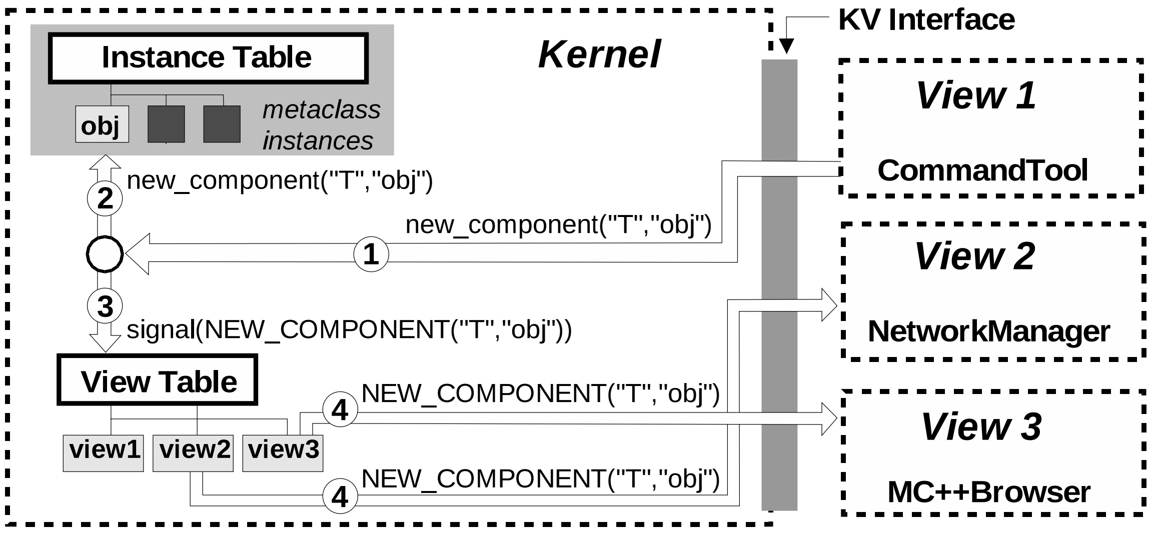


Figure 5.9: Kernel-view interface implementation

Figure 5.9 exemplifies the above. First, a a command to create an instance `obj` of the component `T` is sent to the kernel (1) via the `CommandTool` text interface view presented in Section 4.3.2. After the new instance is created, the kernel must signal all interested views about the creation event. The kernel maintains a list of all views and of the events in which they are interested in the `ViewTable` subsystem. The `ViewTable` is asked to signal all the views interested in the `NEW_COMPONENT` event (3). The event and its parameters are broadcasted (4) to the `NetworkManager` and the `MC++Browser`, described in Sections 4.2.2 and 4.2.3. Finally, the views update themselves: `NetworkManager` displays a new component instance icon and `MC++Browser` displays a new line `T obj` in its instance list browser.

The `NetworkManager` View

The `NetworkManager` view implements the network manager GUI presented in Section 4.2.2. This view consists of a collection of GUI widgets implemented in C++ atop of the Motif toolkit [33] grouped in two subsystems (Fig. 5.10).

The `LibraryBrowser` subsystem implements the meta-library browser described in Section 4.2.2. The `LibraryBrowser` displays an icon for every instantiable component in the `MetaCPP-Manager`'s entity table (Section 5.1.5). Whenever libraries are loaded or unloaded, entities in this table change, so the `LibraryBrowser` rebuilds its icons from the new table contents. `Library-Browser` monitors thus the `LIBRARY_LOAD` and `LIBRARY_UNLOAD` events.

The `NetworkEditor` subsystem implements the network editor described in Section 4.2.2. The `NetworkEditor` has two roles. First, it acts as a view on `MetaCPPManager`'s entity table (Section 5.1.5) by displaying the dataflow network as icons connected by graphic links. Second, it offers a GUI to instantiate, delete, rename, clone, reparent, connect, and disconnect components. All these operations are delegated to `MetaCPPManager`. `NetworkEditor` monitors thus the `NEW_COMPONENT`, `DEL_COMPONENT`, `CONNECT`, `DISCONNECT`, and `UPDATE` events. Monitoring the last event allows flashing the icons to indicate component updates, as in the AVS or Oorange systems. By monitoring the above events, `NetworkEditor` always offers a GUI which reflects the actual state of the instance table. The events can be caused either by `NetworkEditor` itself or by other views (see for instance the example in Fig. 5.9).

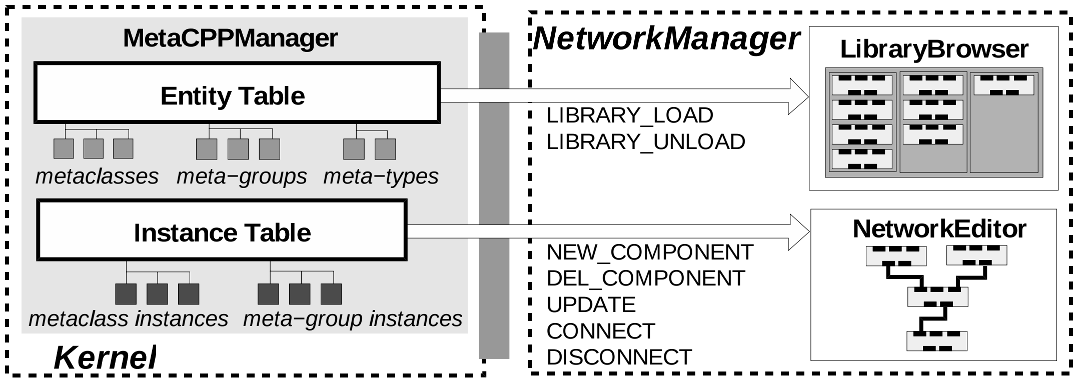


Figure 5.10: The `NetworkManager` view

The `MC++Browser` View

The `MC++Browser` view implements the MC++ browser GUI presented in Section 4.2.3. The view contains three Motif GUI panels for the library browser, metaclass browser, and instance browser, and observe the events indicated respectively in Fig. 5.11. The `MC++Browser` is thus a view on the `MetaCPPManager` and `LibraryManager` kernel subsystems. Besides observing the respective events, `MC++Browser` does also send commands to the kernel for loading and unloading libraries, and creating, deleting, renaming, cloning, and reparenting component instances.

The `InteractorManager` View

The `InteractorManager` view implements the component interactor GUIs presented in Section 4.3.1. `InteractorManager` has two main functions. First, it constructs the component inter-

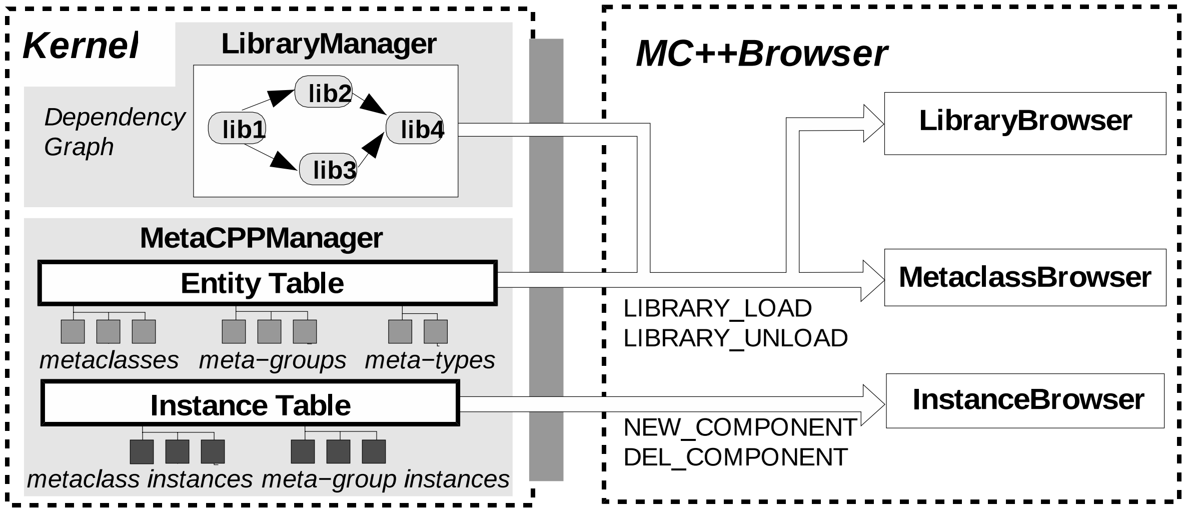


Figure 5.11: The `MC++Browser` view

actors from their MC++ declarations. Secondly, it manages the component interactors, i.e. updates them when their port values change and writes their component ports when the end user interacts with their widgets.

`InteractorManager` consists of three subsystems: `InteractorMaster`, `PortEditor`, and `InteractorTable`. The way in which the subsystems of `InteractorManager` implement its two functions is described next.

Interactor Construction

The `InteractorMaster` and `PortEditor` subsystems implement the component interactor construction algorithm described in Section 4.4. The construction process, exemplified for the `Example` metaclass discussed in Section 4.3.1, Fig. 4.8, proceeds as follows (see also Fig. 5.12):

1. a request `construct("object")` to construct an interactor for the instance `object` of metaclass `Example` is sent to the `InteractorMaster`. The request can be sent by e.g. the `MC++Browser` or the `NetworkManager`.

2. the `InteractorMaster` retrieves the MC++ declaration of `object`, i.e. the `Example` metaclass, from `MetaCPPManager`'s entity table.

3. a GUI widget is constructed for every port whose MC++ declaration does not specify the `optional` option (Section 4.4.3). The widgets are then assembled into the final interactor. In order to construct a widget for a specific port, `InteractorMaster` calls upon `PortEditor` as explained below.

4. `PortEditor` searches for the `MOTIFWidget` subclass that can best edit the given port's metatype, as explained in Section 4.4. This is done by determining, for each widgets in `LibraryManager`'s `WidgetTable`, the likelihood to use it as editor for the given port (step 4.1). This involves computing metrics in the C++ type space (Section 4.4), by calling on the `Interpreter` reflection functions discussed in Section 5.1.4 (step 4.2)

5. the constructed GUI interactor is inserted into the `InteractorTable` and then displayed.

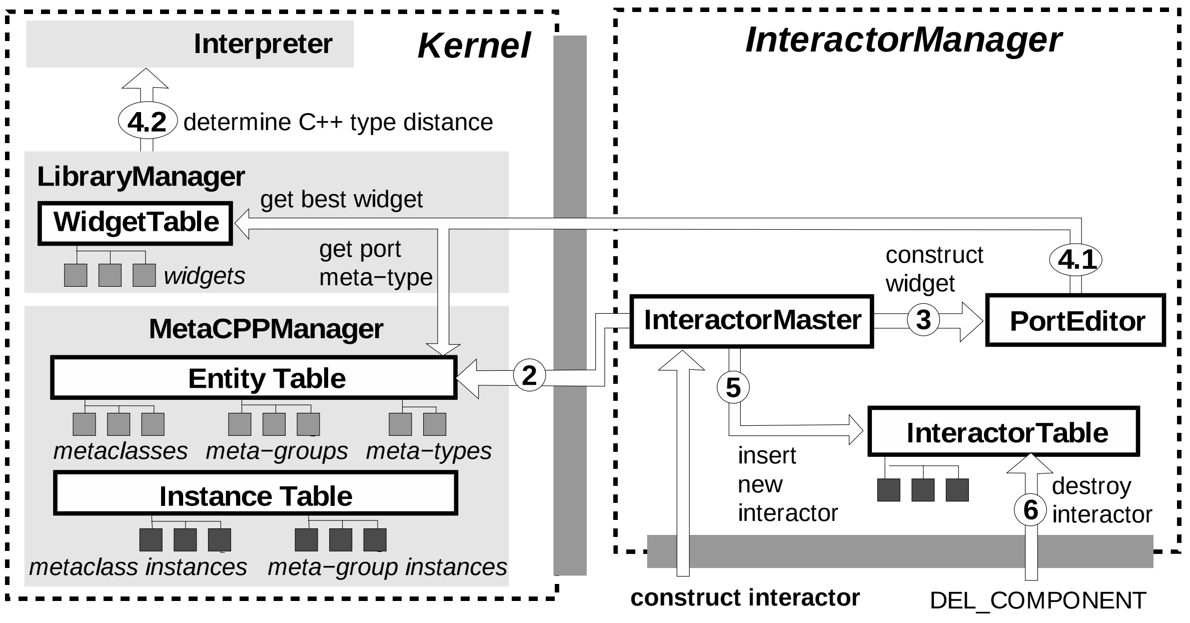


Figure 5.12: Interactor construction process

Since `InteractorManager` observes the `DEL_COMPONENT` event, interactors are destroyed automatically when their components are destroyed (step 6 in Fig. 5.12).

Interactor Management

After a GUI interactor is constructed, it is used both to display the values of its component's ports and to modify its component's write ports. Both processes are illustrated by means of an example (Fig. 5.13) featuring a component `obj` of type `Float`. This component represents a float number and has an input and output port, both of the C++ float type.

At this point we introduce the interface of the `MOTIFWidget` class. As outlined in Section 4.4.1, `MOTIFWidget` is a C++ base class for all user written widgets. This class declares two virtual methods `char* getValue()` and `void setValue(char*)` which represent the widget - VISSION interface. Concrete widget implementations override `setValue` to set the widget's visual value representation to the value passed as a text C++ expression and `getValue` to return the current widget value as a text C++ expression.

Since the GUI interactor for `obj` has to be updated whenever `obj1` updates, `Interactor-Manager` observes the `UPDATE` event. Upon reception of this event, the interactor is updated as follows (see Fig. 5.13). First, the interactor for the component specified as the `UPDATE` event argument is fetched from the `InteractorTable` (1). Next, the fetched interactor is passed to `Inter-actorMaster` to be updated (2), which asks `PortEditor` to update the widgets of all modified component ports (3). For every such widget, `PortEditor` calls its `setValue` method with the C++ value of its port, obtained from `MetaCPPManager`'s `read_port` (4). In the above example, this means that the dial and the float label widget are redrawn to reflect their new port values.

Steering the component ports by the end user via GUI interactors is implemented as follows (see Fig. 5.13). First, the end user interacts with a write port widget, e.g. turns the dial's hand with the mouse. The widget's Motif callback, activated by the X event system, signals the `PortEditor` subsystem (5). Finally, `PortEditor` notifies the kernel by calling `MetaCPPManager`'s `port_write`

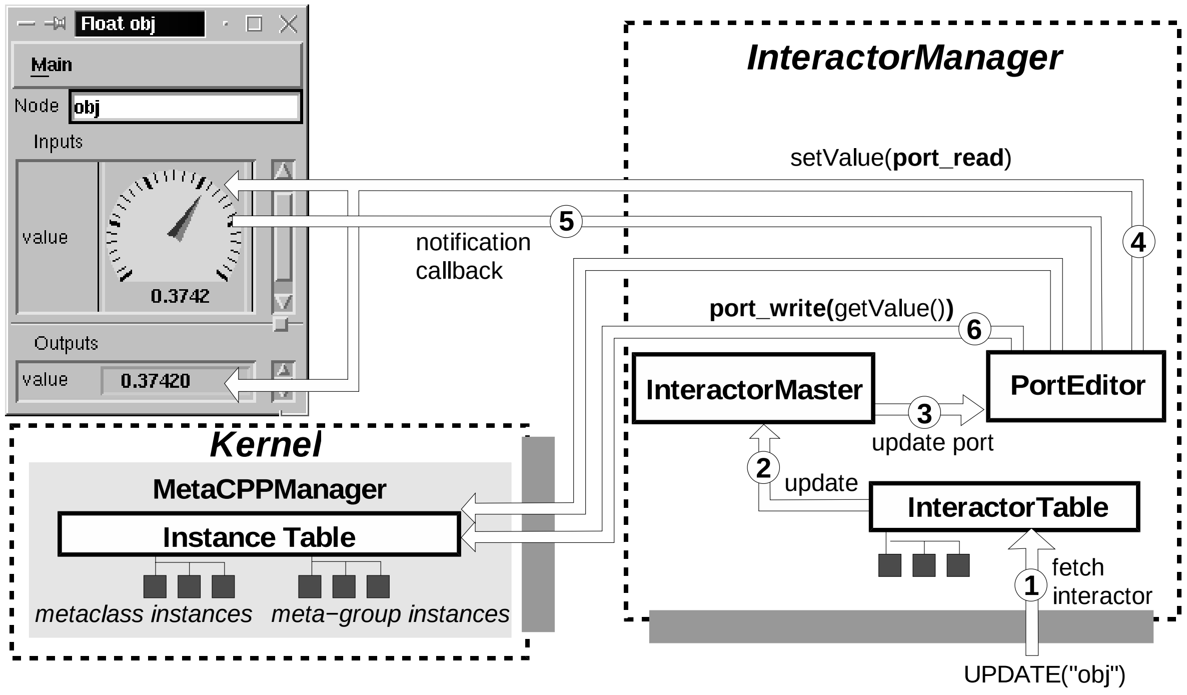


Figure 5.13: Interactor management process

operation with the value delivered by the widget's `getValue` method.

The above GUI interactor implementation can accommodate any type of widget as long as it complies with the very simple `setValue/getValue` `MOTIFWidget` interface. Several widgets have been easily implemented as `MOTIFWidget` subclasses. The minimal widget interface gives the widget developer total freedom to design the widget's policy and look-and-feel. For instance, a dial widget could notify the `PortEditor` of a value change when the user releases the hand at the end of turning, or alternatively at every mouse motion during the turning. The former option is preferred for complex dataflow networks which update slowly or when the edited value has to be carefully set. The latter option is very useful for fast dataflow networks to e.g. perform interactive data exploration and is essential for simulation steering.

5.3 Conclusions

This chapter has presented the main aspects involved in the design and implementation of VISSION. VISSION's design reflects closely its *fundaments* which are based on the combination of object orientation and dataflow modelling realised by the C++ and MC++ language combination, and its *usage*, based on several user interfaces which address the several application design, component design, and end user tasks.

VISSION is consequently structured in two parts: the kernel and the views. The kernel implements the combination of the C++ and MC++ languages which realises the kernel-component interface described in Chapter 4. This implementation is based on a C++ virtual machine which offers the mechanisms needed to manipulate all the C++ language constructs in VISSION's dynamic environment. The views implement several complementary user interfaces which provide visual component manipulation and dataflow network construction, component browsing facilities, GUI interaction facilities with the components of a running application, command-line like interaction, and more. The views are complementing and not competing with each other, and can be used independently by different user categories. New views can be programmed and added without any modification to the kernel, as the two interact via a well-defined kernel-view interface based on the event model. New views could be devised either to augment the existing ones or to provide the same functionality in a different, more convenient manner.

Throughout VISSION's design and implementation a few particular requirements were sought:

- **completeness**: VISSION should be an exact implementation of the OO-dataflow architectural model presented in Chapter 4. Since we have shown that this model does satisfy the combination of our user requirements, summarised in the problem statement of the conclusions of Chapters 2 and 3, we are confident that its exact implementation will indeed address this problem statement in practice. This is reflected in several design choices such as the choice of a full-fledged independent C++ virtual machine for the system's kernel.

- **concern segregation**: as mentioned, VISSION's implementation is based on the OO-dataflow architectural model. This model specifies clearly the system's infrastructure but does not (and can not) prescribe elements above this infrastructure. For example, the end user or application designer GUIs can reflect the OO-dataflow combination in various ways, depending on the convenience of the actual users. VISSION's actual implementation mirrors the separation of these concerns by the client-server architecture based on the single kernel and open set of views.

In the next chapter, concrete SimVis applications will demonstrate how the actual implementation of VISSION, based on the new OO-dataflow architectural model, satisfies our original problem statement introduced in Chapter 1.

Chapter 6

Numerical Simulation Applications

The goal of this thesis, detailed in the second chapter, was the construction of a flexible software system for simulation and visualisation. The desired system should answer the requirements of a research environment, where users often need to switch quickly between the roles of end user, application designer, and component developer.

In this chapter and the next one, we present several applications of VISSION for various scientific simulation and visualisation tasks. Next to these application, we will also present how specific functionality was added to VISSION by means of external application component libraries. For each application case, the advantages offered by VISSION as compared to alternative designs are discussed.

The outline of this chapter is as follows. Section 6.2 presents the application of VISSION for the computational steering of a particular numerical simulation. In the remainder of this chapter, we show how the numerical setup presented in the first section is extended to a generic numerical library that covers a large class of computational applications. The integration of scientific visualisation components within VISSION is the subject of the next chapter.

6.1 Introduction

As mentioned in Chapter 1, one of the main goals of this work is the integration of numerical simulation and visualisation functionalities in a single interactive application. The dataflow application model is generic and can describe both visualisation and computational applications, the difference between a simulation and a visualisation network being only semantic, not structural. Building computational steering applications in VISSION should therefore be an easy task.

Although reducible to a dataflow network structure, computational software has several particular characteristics. These particularities raise several integration problems, as follows.

- numerical software comes most often in as libraries written in FORTRAN, or as monolithic applications with file-based data interfaces. Both these architectures are directly incompatible with VISSION's is component model built atop of the C++ class model described in the previous chapters. The integration of such software in VISSION requires thus the construction of a component-based interface to the existing libraries of monolithic applications. Depending on the architectural model used by the numerical software to integrate, this operation may be very simple, such as the construction of class wrappers , or might imply massive adaptation or even rewriting of the numerical code.

- if we desire to add visualisation capabilities to existing monolithic numerical applications, there is no way in which VISSION's dataflow engine could gain control to process events during the time the numerical code executes. This limitation, stemming from VISSION's single threaded synchronous execution architecture, can severely decrease the interactivity of a back-end visualisation coupled to a slow numerical engine.

- many numerical applications have an intrinsic iterative structure during which the solution to a certain problem is progressively refined. Several hierarchies of loops might be present in a single application, such as the internal loop of an iterative solver and the external loop involving the solver in a time dependent simulation. When such software is integrated with visualisation operations in a VISSION network, we would like to expose one or both loops to VISSION's network traversal mechanism to be able to see the gradual solution refinement as well as the time dependent evolution.

Several solutions can be envisaged for the above problems, depending on the requirements of the concrete SimVis application. In the following, several VISSION applications that integrate numerical computation and visualisation are presented. These applications illustrate the above integration problems and various solutions that were adopted in function of their respective contexts.

6.2 Electrochemical Drilling

The electrochemical drilling (ECD) process is used by the engine construction industry to drill holes in hard metal objects such as turbine blades. These holes are needed for cooling the blades, either for internal cooling or film cooling. The cooling is caused by the flow of relatively cool air through the holes. This cooling extends the lifetime of the blades significantly and even may allow for higher working temperatures (i.e. higher efficiency) in the engine. In order to increase the heat transfer in the holes, the wall of the cooling passage is provided with multiple ribs or *turbulators* which cause turbulence in the air flow and consequently a better air-metal heat transfer.

The ECD technique is used for drilling the complex geometry of the turbulator holes. An electrolytic process is employed where an anode, acting like a drill, advances slowly into the turbine blade acting as a cathode. The anode speed and the voltage applied between the drill and the plate must be varied in time in a well controlled manner in order to produce the desired turbulator geometry. Finding the correct voltage and speed variations in time that produce the desired geometry involves many expensive experiments. To reduce these costs, a numerical simulation of the ECD process was produced.

An important issue in modelling the ECD process is its real time characteristics and the ability to simulate it with a computer program. In the above process, there are several physical parameters which have to vary in time in order to obtain a desired shape for the turbulators. Finding the values of the parameters that produce the desired geometry involves in fact solving an inverse problem. This implies the ability to run the ECD numerical simulation, change its input parameters interactively, and visualise the process evolution in real time. What is actually needed is an ECD process simulation providing computational steering and on-line visualisation. The remainder of this section presents how such a simulation has been constructed, starting from the physical model, followed by the numerical approach used, and ending with the software implementation of the coupling between the visualisation and simulation parts (see also [76]).

6.2.1 Physical Modelling

As already mentioned, the ECD process is based on electrolysis. A cylindrical drill insulated on the outside is lowered into the material with a certain speed and voltage applied to it, in order to produce a cylindrically shaped hole. Because of the problem's axisymmetry, a two-dimensional computational model is used (Fig. 6.1 a).

To describe the electrolysis process, Faraday's law is used. This yields for the material volumetric removal rate:

$$\frac{dV}{dt} = \frac{e_a}{\rho_a} I, \tag{6.1}$$

where V is the anode's volume, and ρ_a and e_a are the anode's density and electrochemical equivalent respectively, both assumed to be constant. The current flow **J** in the electrolyte is described by Ohm's law

$$\mathbf{J} = -k_e \operatorname{grad} \phi, \tag{6.2}$$

where **J** is the current density caused by a potential difference ϕ and k_e is the electrolytic conductivity assumed constant. As the electrolyte solution is overall electrically neutral, the electric field is assumed to be divergence free

$$\operatorname{div} \mathbf{J} = 0. \tag{6.3}$$

Overall, equations (6.2) and (6.3) result in the Laplace equation for describing the electric potential ϕ.

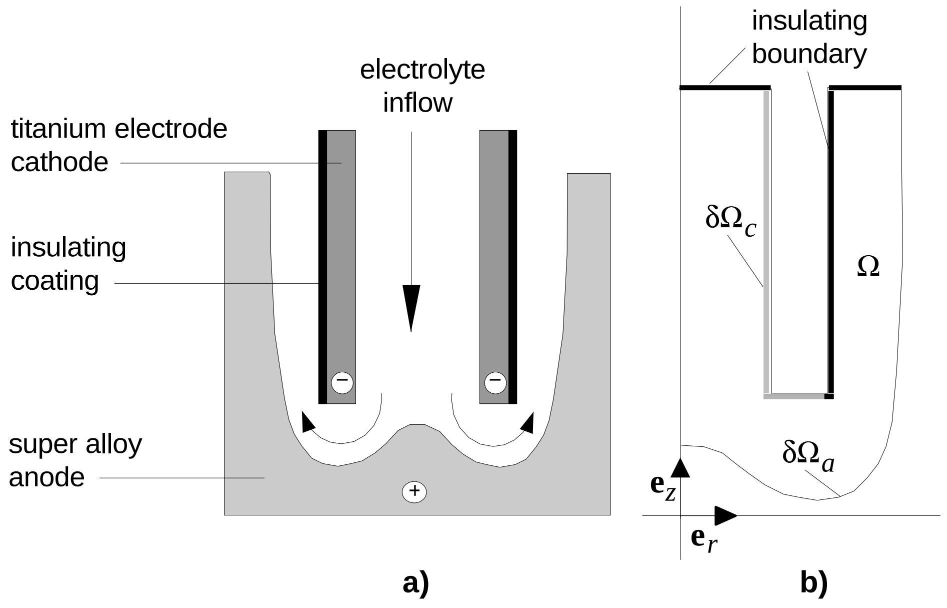


Figure 6.1: Electrochemical drilling

As the electrodes are assumed to be perfectly conducting, essential boundary conditions are imposed on ϕ at the electrode's surface.

6.2.2 Mathematical Equations

As previously mentioned, the ECD process is modelled by the Laplace equation. The two-dimensional boundary $\delta\Omega$ of the considered computational domain Ω is split into the anode boundary $\delta\Omega_a$ which represents the instantaneous surface of the drilled hole, the cathode boundary $\delta\Omega_c$ which is maintained at a constant potential U. On the remainder of the boundary, a homogeneous Neumann condition is imposed to model its insulating characteristics. Now the formulation for the electric potential ϕ yields:

$$\begin{aligned} &\mathrm{div}(-k_e \mathrm{grad}\phi) = 0 && \text{in } \Omega \\ &\phi = 0 && \text{on } \delta\Omega_a \\ &\phi = -U && \text{on } \delta\Omega_c \\ &\frac{\delta\phi}{\delta\mathbf{n}} = 0 && \text{on } \delta\Omega - (\delta\Omega_a + \delta\Omega_c), \end{aligned} \tag{6.4}$$

where **n** is the outwards normal vector to the boundary $\delta\Omega$.

Assuming that the current density is constant for a short time interval, the shift of anode boundary $\delta\Omega_a$, along its normal due to the electrolytic erosion, can be computed from:

$$\frac{d\mathbf{x}}{dt} = \frac{e_a}{\rho_a}(\mathbf{J}(\mathbf{x}), \mathbf{n})\mathbf{n}, \tag{6.5}$$

for all points **x** on $\delta\Omega_a$. Time is treated explicitly, as both U and the position of the boundary $\delta\Omega_c$ are functions of time. Starting from an initial hole geometry, the potential distribution can be computed. Next, the current density on the anode boundary and the anode recession are computed. The process loop is closed by recomputing the new potential distribution, moving the anode boundary, and so on.

6.2.3 Numerical Approach

The ECD process equations presented above are discretised numerically by using a finite element approach with conformal elements. From the discrete approximation of ϕ, the current density flux **J** is computed by numerical derivation. If a mixed finite element approach were employed, one could compute both the potential ϕ and the flux **J** instead of obtaining the latter by derivation of ϕ. Solving the flux implicitly with the potential would produce more accurate results and handle well singularities in the solution such as, for example, close to the cathode's tip. However, it can be shown that the mixed method is almost 2.5 times more expensive per iteration than the conforming method [76]. As the domain Ω of the ECD simulation is constantly expanding, the number of elements can increase quite rapidly. To keep the computation time within interactive or nearly interactive limits, we prefer to use the conforming method.

We have implemented the ECD numerical simulation by using a general purpose finite element library written by us in C++ [107, 76]. The library provides software components for describing the problem computational domain, boundary conditions, and differential equation to be solved. Moreover, the library implements several iterative solvers, preconditioners, and matrix storage formats. In this approach, a numerical simulation is described as a dataflow network of computational objects. Its integration in VISSION posed therefore no particular problems.

Figure 6.2 shows the VISSION dataflow network for the ECD numerical simulation. The network starts with the `BoundaryCurves` component that describes the various curves which create the computational domain's boundary, as described in the previous sections. The boundary curves are read by a mesh generator engine that produces a triangulation of the 2D computational domain Ω in function of various user-specified meshing parameters such as average element size, mesh refinement factors, and so on. The produced triangulation is stored in the `Mesh` dataset object. Several mesh generators specialisations are provided, such as a Delaunay-based and an own triangulator, both capable to handle arbitrary concave domains containing holes and local mesh refinements. for the ECD simulation,

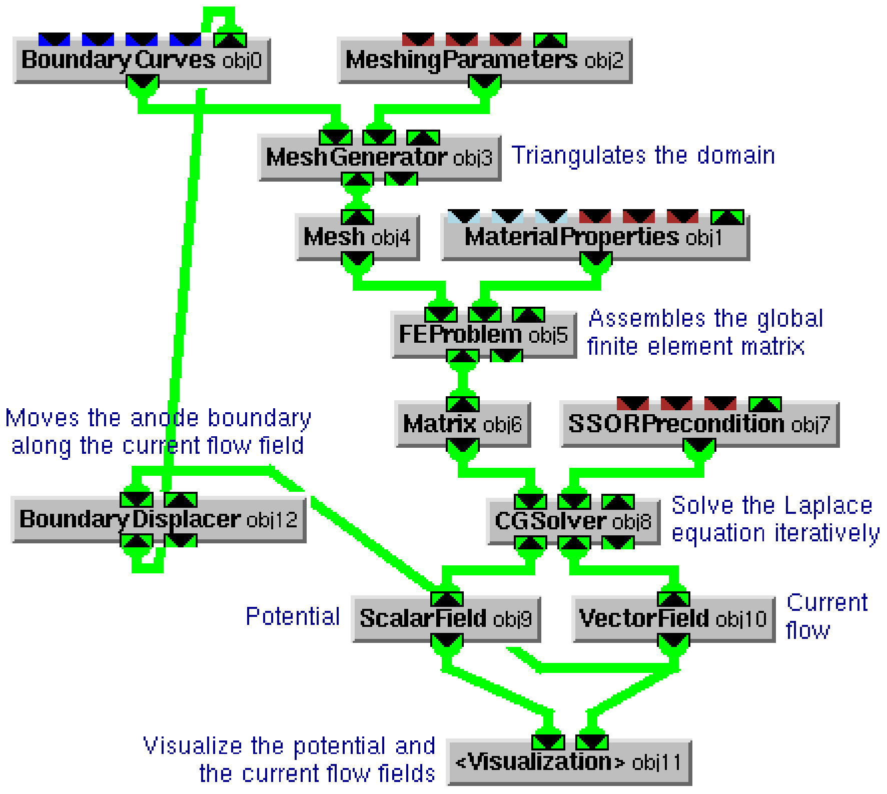


Figure 6.2: ECD simulation dataflow pipeline

Following the mesh computation, the finite element discretisation of the Laplace equation is modelled by the `FEProblem` module. The same module also provides inputs to specify the types and values of the boundary conditions and material properties of the domain Ω. The `FEProblem` assembles the global stiffness matrix in the `Matrix` dataset connected to its output. Once the matrix is assembled, a numerical solver is used to compute the discrete solution for the potential ϕ. In the example illustrated in Fig. 6.2, a conjugate gradient iterative solver with a successive overrelaxation (SSOR) preconditioner is used. The potential solution ϕ and the current density flux **J** calculated by numerical derivation are computed as a 2D scalar and vector field respectively by the `ScalarField` and `VectorField` dataset objects. These objects are the input for the visualisation pipeline, which is contained in the `Visualization` meta-group module.

As described in the previous sections, the computational domain Ω is changing in time. The boundary $\delta\Omega_c$ modelling the drill's tip advances with a constant distance every time step. Also the boundary $\delta\Omega_a$ evolves freely, displaced by the vector field **J** as specified by Faraday's law (Equation 6.1). In order to obtain the new domain boundary $\delta\Omega^{i+1}$, every point **x** of the current discrete boundary $\delta\Omega^i$ is displaced by forward Euler integration

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \Delta t \frac{e_a}{\rho_a}(\mathbf{J}(\mathbf{x}^i), \mathbf{n})\mathbf{n}, \qquad \mathbf{x} \in \delta\Omega^i_a \tag{6.6}$$

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \Delta t s(t^i)\mathbf{e}_z, \qquad \mathbf{x} \in \delta\Omega^i_c \tag{6.7}$$

where $s(t^i)$ is the drill's speed at moment t^i, and Δt is the time step, taken constant. The above equations for moving the boundary are implemented by the `BoundaryDisplacer` modules that takes the current density flux **J** as input and modifies the `BoundaryCurves` object by writing into it.

6.2.4 Computational Steering

The ECD simulation contains two loops. The first loop is encapsulated in the `CGSolver` module and represents the iterative solving procedure for the linear equation system output from the problem discretisation. As this loop takes place within one module, its iterations are invisible to VISSION and thus also to the end user, who notices a new solution only after it has fully converged or the maximum number of iterations allowed has been spent. The second loop is closed by the `BoundaryDisplacer` module which uses the computed current density flux solution to displace the domain's boundary. This loop takes place at the VISSION level, in contrast to the previous one which was contained in a black box fashion within one component. Consequently, VISSION's dataflow engine will compute a new potential and current density flow solution, display them to the end user, displace the boundary, remesh the domain, reassemble a new global matrix, compute a new solution, re-read the user interface controls, and so on, until the end user locks a module of the loop or opens the loop by disconnecting one of the involved links.

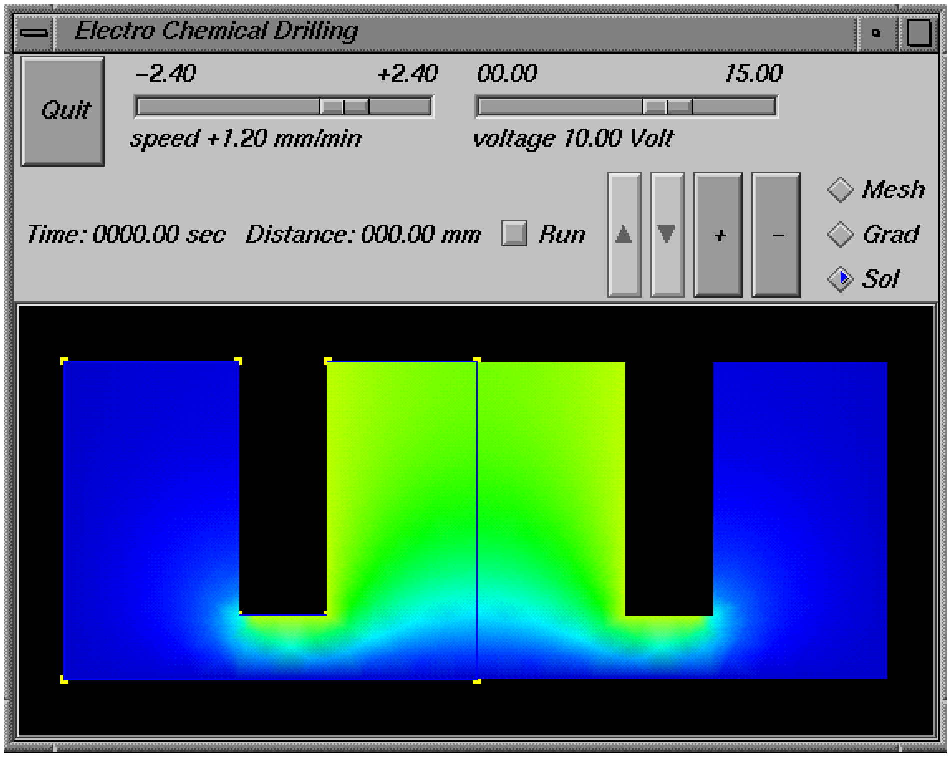


Figure 6.3: ECD simulation user interface

Building a computational loop with components at the VISSION network level has several advantages. First, the visualisation back-end is signalled by the dataflow engine whenever a new solution is produced, which enables the end user monitor the ECD simulation (mesh deformation, solution, and so on) interactively as they are computed. Secondly, the user can vary several parameters of the simulation as this one is running, such as the drill speed or potential difference. This models exactly the real physical process of creating various turbulator shapes by using several variations of the drill voltage and speed in time. The end user can thus easily investigate the inverse problem of determining a desired turbulator shape as a function of the potential and speed evolutions in time.

In contrast to this, we preferred to code the inner simulation loop in a single module (the `CG-`

`Solver`) for two reasons, as follows. First, for every time step several tenths of inner iterations are done. If we want to achieve an interactive simulation, i.e. a few time steps per second, it is better to code the inner loop iterations in highly optimised compiled code rather than to involve VISSION's dataflow engine, which in turn invokes the C++ interpreter and the visualisation back-end, at every inner iteration. Secondly, monitoring the convergence of the solution at a given time step is less important for the ECD end user, typically an engineer. Optimal results have been achieved in practice by using a small time step, in which case the user can control the ECD simulation in real time by interactively changing the drill speed and voltage. Figure 6.3 shows the graphics interface of the ECD simulation which offers various controls for the drill speed, voltage, and visualisation options, such as zoom, pan, and display the potential solution (shown in the actual figure), current flow density vector field, or triangle mesh.

Besides the end user interactive steering and visualisation features, the ECD simulation can be customised also at application design level, by connecting different modules in the VISSION network editor instead of the ones discussed above. The application designer can switch between various types of mesh generators, iterative solvers, and preconditioners with a few mouse clicks. Similarly, the produced datasets could be visualised in other ways than the discussed ones.

6.2.5 Conclusion

The electro-chemical drilling simulation shows the ability of VISSION to be used for constructing and steering computational applications of a turnkey-type, given the appropriate application domain components. Besides the end user interactive steering and visualisation features, the ECD simulation can be customised also at application design level, by connecting different modules in the VISSION network editor instead of the ones discussed above. Different types of mesh generators, iterative solvers, and preconditioners can be switched with a few mouse clicks. Similarly, the produced datasets could be visualised in other ways than the discussed ones.

6.3 The NUMLAB Numerical Laboratory

The previous section has described the implementation of a particular computational application into a component library and its interactive design and control in the VISSION system. The numerical components involved have, however, been specifically designed for the limited scope of the ECD application. It is hard, for example, to extend these components to address other problems, such as other PDEs, ODEs, to use different types of elements, or to implement finite difference computations.

However, it would be desirable to generalise the positive results experienced for the ECD application in terms of interactive design, control, and visualisation of numerical applications. The proposed goal is to achieve a system into which the same application design and use facilities are provided for a large range of numerical applications. Such applications should encompass ordinary differential equations (ODEs), partial differential equations (PDEs), image manipulation and data compression, nonlinear systems of equations, and matrix computations.

A wide range of applications addresses such problems. Following the classification introduced in chapter 2, these applications come as:

- *Libraries:* LAPACK [4], NAGLIB [73], or IMSL [48],
- *Turnkey systems:* Matlab [66], Mathematica [118],
- *Computational frameworks:* Diffpack [14], SciLab [49], Oorange [41].

Though efficient and effective, most existing computational frameworks are limited in several respects. These limitations concern the end user, application designer, and component developer, as overviewed by several authors [6, 88, 27, 110]. A short overview hereof follows.

First, few computational frameworks facilitate convenient interaction between visualisation (data exploration) and computations (numerical exploration), both essential to the end user. Secondly, from the application designer perspective, the visual programming facility, often provided in visualisation frameworks such as AVS or Explorer [113, 44], usually is not available for numerical frameworks. Conversely, it is quite difficult to integrate large scale computational libraries in visualisation frameworks [88]. From the numerical component developer perspective, understanding and extending a framework's architecture is still (usually) a very complex task, albeit noticeably simplified in object-oriented environments such as Diffpack or VTK.

Next to limitation with respect to the three types of users, many computational frameworks are constrained in a more structural manner: Similar mathematical concepts are not factored out into similar software components. As a consequence, most existing numerical software is heterogeneous, thus hard to deploy and understand. For instance, in order to speed up the iterative solution of a system of linear equations, a preconditioner is often used. Though iterative solvers and preconditioners fit into the same mathematical concept – that of an approximation $\mathbf{x}$ which is mapped into a subsequent approximation $\mathbf{z} \approx \mathbf{F}(\mathbf{x})$ – most computational software implements them incompatibly, so preconditioners cannot be used as iterative solvers and vice versa [14].

Another example emerges from finite element libraries. Such libraries frequently restrict reference element geometry and bases to a (sub)set of possibilities found in the literature. Because this set is hard coded, extensions to different geometries and bases for research purposes is difficult, or even impossible.

We have addressed the above limitations by designing the NUMLAB numerical library and integrating it into VISSION. NUMLAB is a numerical framework which provides C++ software components (objects) for the development of a large range of interdisciplinary applications (PDEs, ODEs, non-linear systems, signal processing, and all combinations). We integrated NUMLAB in VISSION to provide it with interactive application design/use facilities. Its computational libraries factor out fundamental notions with respect to numerical computations (such as evaluation of operators $\mathbf{z} = \mathbf{F}(\mathbf{x})$ and their derivatives), which keeps the amount of basic components small.

The remainder of this chapter presents the NUMLAB framework. In section 6.4, the mathematics that we wish to model in software is reduced to a set of simple but generic concepts. Section 6.5 shows how these concepts are mapped to software entities. Section 6.7 illustrates the above for the concrete case of solving the Navier-Stokes partial differential equation. Section 6.8 presents how concrete simulations combining computations and visualisation are constructed with the NUMLAB-VISSION combination.

6.4 The mathematical framework

In order to reduce the complexity of the entire software solution, we show how NUMLAB formulates different mathematical concepts with a few basic mathematical notions. It turns out that in general NUMLAB's components are either operators $\mathbf{F}$, or their vector space arguments $\mathbf{x}, \mathbf{y}$. The most frequent NUMLAB operations are therefore operator evaluations $\mathbf{F}(\mathbf{x})$ and vector space operations such as $\mathbf{x}+\mathbf{y}$. Important is the manner in which NUMLAB facilitates the construction of complex problem-specific operators (for instance transient Navier-Stokes equation with heat transfer), and related complex solvers. NUMLAB offers:

1. *Solvers for systems of linear equations*: Such systems are also operators $\mathbf{F}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$. Their solution is reduced to a sequence of operator evaluations and vector space operations.

2. *Solvers for systems of non-linear equations*: Such systems are operators, and their solution is reduced to the solution of a sequence of linear systems

3. *Problem-specific solvers for systems of ODEs*: Time-step and time-integration operators formulated with the use of (parts of) the problem-specific operators mentioned above. The former operators require non-linear solvers for the computation of solutions.

4. *Problem-specific operators*: Transient Finite Element, Volume, Difference operators $\mathbf{F}$ for transient boundary value problems (BVPs); Operators which formulate systems of ordinary differential equations (ODEs); operators which act on linear operators (for instance image filters). The operator framework is open, users can define customised operators $\mathbf{z} = \mathbf{F}(\mathbf{x})$.

The reduction from one type of operator into another proceeds as follows. Section 6.4.1, examines systems of (non-)linear equations and preconditioners, section 6.4.2 considers the reduction of systems of ODEs to non-linear systems, and section 6.4.3 deals with an initial boundary value problem. The presented mathematical reductions are de facto standards. The new aspect is NUMLAB's software implementation which maps one to one with these techniques.

6.4.1 Non-linear systems and preconditioners

This section presents NUMLAB's operator approach , and demonstrates how operator evaluations reduce to repeated vector space operations and operator evaluations. This is illustrated by means of examples, which include (non-)linear systems, and preconditioning techniques.

First, consider linear systems of the form $\mathbf{F}(\mathbf{x}) = \mathbf{f}$. Here $\mathbf{F}(\mathbf{x}) = \mathbf{A}\mathbf{x}$ is a linear operator, with an N by N coefficient matrix $\mathbf{A}$, and $\mathbf{f} \in \mathbf{R}^N$ is a right hand side. The NUMLAB implementation of the evaluation $\mathbf{z} = \mathbf{F}(\mathbf{x})$ is:

```
F.eval(z, x);
```

The actual implementation of `eval()` varies with the application type (for instance full matrix, sparse matrix, image, etc.). Though $\mathbf{z}$ is a resulting value, its initial value can be used for computations (for instance as an initial guess).

Next, NUMLAB formulates a linear system $\mathbf{F}(\mathbf{x}) = \mathbf{f}$ with the use of an affine operator $\mathbf{G}$:

$$\mathbf{G}(\mathbf{x}) = \mathbf{F}(\mathbf{x}) - \mathbf{f}. \tag{6.8}$$

The user constructs this NUMLAB system by providing $\mathbf{G}$ with the the linear operator and a right hand side vector:

```
G.setO(F);
G.setI(f);
```

The residual $\mathbf{z} = \mathbf{G}(\mathbf{x})$ is computed with:

```
G.eval(z, x);
```

Next, let $\mathbf{x} \in \mathbf{R}^N$ be a given vector, and focus on the solution(s) of $\mathbf{G}(\mathbf{z}) = \mathbf{x}$, i.e, on solution methods for affine operators. Assume that operator $\mathbf{R}$ approximates $\mathbf{G}^{-1}$:

$$\mathbf{G}(\mathbf{z}) = \mathbf{x} \Longleftrightarrow \mathbf{z} = \mathbf{G}^{-1}(\mathbf{x}) \Longleftrightarrow \mathbf{z} \approx \mathbf{R}(\mathbf{x}). \tag{6.9}$$

For the sake of demonstration, and without loss of generality, we assume that $\mathbf{R}$ is a left-preconditioned Richardson iterative solution method [67], with preconditioner $\mathbf{P}$. Such a method is based on a successive substitution process:

$$\mathbf{z}^{(k+1)} \quad = \quad \mathbf{z}^{(k)} - \mathbf{P}(\mathbf{G}(\mathbf{z}^{(k)}) - \mathbf{x}). \tag{6.10}$$

The process is terminated by a user-specified stopping criterion of the form $S\colon \mathbf{R}^n \mapsto \{0, 1\}$. The iterations stop as soon as $S(\mathbf{P}(\mathbf{G}(\mathbf{z}^{(k)}) - \mathbf{x})) = 0$. This recursion will converge if for instance $\mathbf{G}$ is as in (6.9) with $\mathbf{F}$ positive definite, and if $\mathbf{P}(\mathbf{x}) = h\mathbf{x}$ with h positive and small enough.

The related NUMLAB operator $\mathbf{R}$ for (6.10) is defined by its implementation of its `eval()` method:

```
R.eval(z, x)
{
 P.setO(G);
 repeat
 {
  G.eval(r, z);
  r -= x;
  P.eval(s = 0, r);
  z -= s;
 }
 while (S(s) > 0);
}
```

The system $\mathbf{G}(\mathbf{z}) = \mathbf{x}$ is solved with two lines of code:

```
R.setO(G).setP(T).setS(S);
R.eval(z, x);
```

Observe that solver $\mathbf{R}$ uses $\mathbf{z}$ both as initial guess $\mathbf{z}^{(0)} \in \mathbf{R}^N$ and as final approximate solution, whereas preconditioner $\mathbf{P}$ must use **0** as an initial guess. If a preconditioner is not provided, a default – the identity operator – is substituted. The stopping criteria are similarly dealt with. Note that operators can make use of operators: The preconditioner for Richardson's algorithm could have been Richardson's algorithm itself, a diagonal preconditioner, an (incomplete) LU factorisation, and so forth. Furthermore, the `eval()` methods of the solver $\mathbf{R}$, preconditioner $\mathbf{P}$ and system $\mathbf{G}$ are syntax-wise identical.

The pseudo code for $\mathbf{R}$ above executes `P.setO(G)`, so preconditioner $\mathbf{P}$ can use (has access to) $\mathbf{G}$ and its Jacobian. Further, the linear system $\mathbf{F}(\mathbf{z}) = \mathbf{f}$ could have been solved directly with $\mathbf{R}$:

```
R.setO(F);
R.eval(z, f);
```

NUMLAB formulates systems, solvers and preconditioners all with the use of `set-` and `eval()` syntax – though related mathematical concepts differ. Few other methods such as `update()` exist, and relate to data flow concepts, outside the current paper's scope.

A closer examination of the Richardson operator reveals more information of interest. NUMLAB implements all its operator evaluations with: (1) Vector space operations; and (2) all which remains: Nested operator evaluations. This is clearly demonstrated by $\mathbf{R}$'s implementation above:

$$\begin{array}{rcl} \mathbf{r} & \overset{(1)}{=} & \mathbf{G}(\mathbf{z}^{(k)}); \\ \mathbf{r} & \overset{(2)}{=} & \mathbf{r} - \mathbf{x}; \\ \mathbf{s} & \overset{(1)}{=} & \mathbf{P}(\mathbf{r}); \\ \mathbf{z}^{(k+1)} & \overset{(2)}{=} & \mathbf{z}^{(k)} - \mathbf{s}, \end{array} \tag{6.11}$$

where, $^{(1)}$ denotes *operator evaluation* and $^{(2)}$ *vector space operation*. This clear cut classification of operations thoroughly simplifies the mathematical framework.

Though NUMLAB regards preconditioning as approximate function evaluation – which simplifies its framework – this does not solve the problem of proper preconditioning. Specific iterative solution methods might require preconditioners to preserve for instance symmetry (such as the preconditioned conjugate gradient method PCG [10]) or at least positive definiteness of the symmetric part (for minimal residual methods, see [92] for GMRES and [7] for GCGLS). All iterative solvers have some requirements: Robust methods (e.g. [57]), multi-level methods (e.g. [9] and [62]), multi-grid methods (e.g. [43]), and incomplete factorisation methods (e.g. [42]).

The application designer should keep these mathematical restrictions in mind, when designing a suitable solver for the problem at hand.

Similar to linear systems, NUMLAB also formulates non-linear systems with the use of operators $\mathbf{G}$, and looks for solutions of $\mathbf{G}(\mathbf{z}) = \mathbf{x}$. The Jacobian (Frechet derivative) of a (non-linear) operator $\mathbf{G}$ at point $\mathbf{x}$ is denoted by $D\mathbf{G}(\mathbf{x})$ – or by $D\mathbf{G}$ if $\mathbf{G}$ is linear.

Related non-linear solvers are again formulated as operators. Non-linear operators $\mathbf{G}$ which do not provide derivative evaluation, can be solved with the use of a fixed point method (comparable to the Richardson method above), or with a combinatorial fixed point method [112] (a multi-dimensional variant of the bisection method). Non-linear operators $\mathbf{G}$ which provide derivative evaluation can also be solved with (damped, inexact) Newton methods (see [25] and [28]). A typical NUMLAB code for an undamped Newton method is:

```
Newton.eval(z, x)
{
 repeat
 {
  G.eval(r, z);
  r -= x;

  Solver.setO(G.getJacobian(z));
  Solver.eval(s, r);
  z -= s;
 }
 while (S(s) > 0);
}
```

and a system $\mathbf{G}(\mathbf{z}) = \mathbf{x}$ is solved by this method with:

```
Newton.setO(G).setSolver(R);
Newton.eval(z, x).
```

Here Richardson's method is used to solve the linear systems.

Again, the application designer should take care that the fixed point function is chosen properly, so it preserves properties of $\mathbf{F}$, such as symmetry and positive definiteness of the symmetric part.

In order to close this section on systems of equations and solvers, note that images are also treated as operators

$$\mathbf{F}(\mathbf{x}) = \mathbf{A}\mathbf{x}\,, \tag{6.12}$$

where $\mathbf{A}$ is a matrix (or block-diagonal) matrix of colour intensities. Thus, image visualisation reduces to Jacobian visualisation.

6.4.2 Ordinary differential equations

Standard discretisations of ordinary differential equations can also be formulated as operators whose evaluation reduces to a sequence of vector space operations and function evaluations. For instance, let $\mathbf{E}$ be an operator, and consider the initial value problem: Find $\mathbf{x}(t)$ for which:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{E}(t, \mathbf{x}(t)) \quad (t > 0), \qquad \mathbf{x}(0) = \mathbf{x}_0\,. \tag{6.13}$$

Let $h > 0$ denote the discrete time-step, and define $t_k = kh$ for all $k = 0, 1, 2, \ldots$. Provided with an approximation $\mathbf{x}^{(k)}$ of $\mathbf{x}(t_k)$, a fixed-step Euler backward method determines an approximation $\mathbf{x}^{(k+1)}$ of $\mathbf{x}(t_{k+1})$

$$\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} = h\mathbf{E}(t_{k+1}, \mathbf{x}^{(k+1)}), \tag{6.14}$$

which can be rewritten as

$$\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} - h\mathbf{E}(t_{k+1}, \mathbf{x}^{(k+1)}) = \mathbf{0}\,. \tag{6.15}$$

Define the operator $\mathbf{T}$ as follows:

$$\mathbf{T}(\mathbf{x}) = \mathbf{x} - \mathbf{x}^{(k)} - h\mathbf{E}(t_{k+1}, \mathbf{x}). \tag{6.16}$$

Then $\mathbf{x}^{(k+1)}$ is a solution of $\mathbf{T}(\mathbf{x}) = \mathbf{0}$. Of course, $\mathbf{T}$ depends on the user-provided values $\mathbf{x}^{(k)}$ and h. The NumLab evaluation code of $\mathbf{z} = \mathbf{T}(\mathbf{x})$ is:

```
T.eval(z, x)
{
 t[k+1] = t[k] + h;
 E.setT(t[k+1]);
 E.eval(z, x);
 z *= h;
 z += x;
 z -= x_1;
}
```

Because the approximate solution on t_{k+1}, i.e., $\mathbf{x}^{(k+1)}$ is a root of $\mathbf{T}$, it is computed with the code:

```
T.setT(t[k]).setX(x[k]).setH(t[k+1] - t[k]);
Newton.setO(G).setSolver(R);
Newton.eval(z, 0);
```

The operator formulation $\mathbf{x}^{(k+1)} = \mathbf{T}^{-1}(\mathbf{0})$ applies to all explicit methods such as Runge Kutta type methods [17], as well as to all implicit discretisation methods, such as Euler Backward and Backward Difference Formulas (BDF) [39].

It is obvious that the evaluation of $\mathbf{T}$ at a given $\mathbf{x}$ again only involves *vector space operations* and *operator evaluations.* Solving $\mathbf{T}(\mathbf{x}) = \mathbf{0}$ can thus be done by several methods: Successive substitution, Newton type methods, preconditioned methods, etc.

Naturally, a time-step integrator complements the time-step mechanism. NUMLAB provides standard fixed time-step methods and – required for stiff problems – adaptive time-step integrators of the PEC and PECE type [67].

6.4.3 Partial differential equations and initial boundary value problems

In order to show how partial differential equations (PDEs) are reduced to (non-)linear systems of equations, consider an initial boundary value problem. Let $\Omega \subset \mathbf{R}^d$ be the bounded region of interest, and let $\partial\Omega$ denote its boundary. We denote points in this region with $\mathbf{c} \in \Omega$ ($\mathbf{x}$ is reserved for vectors and related iterands). The problem of interest is: Find a solution u on $[0, \infty) \times \Omega$ which satisfies

$$\frac{\partial}{\partial t}u = \Delta u + f \qquad (t > 0), \tag{6.17}$$

subject to initial condition $u(0, \mathbf{c}) = u_0(\mathbf{c})$ for all $\mathbf{c} \in \Omega$, and boundary conditions $u(t, \mathbf{c}) = \gamma(\mathbf{c})$ for all $t \in [0, \infty)$ and $\mathbf{c} \in \partial\Omega$. For the sake of presentation, the boundary conditions are all assumed to be of Dirichlet type. With a method of Lines (MOL) approach, equation (6.17) fits into the framework (6.13), for a suitable operator $\mathbf{E}$, to be defined.

As an alternative, one can first discretise in time, and next discretise in space, or simultaneously discretise with respect to both (see for instance [8]).

For a MOL solution of (6.17), the region of interest Ω is covered with a grid of elements (with the use of a uniform, Delaunay, or bisection type [68] grid generator). Next, the static equation $-\Delta u = f$ is discretised with one of the available methods (standard conforming higher order finite elements, and non-forming elements as for instance in [51]).

A standard Galerkin approach assumes that the solution u is in a linear vector space V with basis $\{v_j\}_{j=1}^N$. For the method of lines approach applied to (6.17) one sets

$$u(t, \mathbf{c}) = \sum_i x_i(t) v_i(\mathbf{c}). \tag{6.18}$$

for all time t and $\mathbf{c} \in \Omega$. Functions in V are identified with their coefficient vectors in $\mathbf{R}^n$, so u is identified with $\mathbf{x}$. Multiplication of (6.17) with (test) functions $\{v_j\}_{j=1}^N$, followed by partial integration over Ω leads to a system of ODEs

$$\mathbf{M}\frac{d}{dt}\mathbf{x}(t) = -\mathbf{G}(\mathbf{x}(t)), \qquad (t > 0). \tag{6.19}$$

Here,

$$[\mathbf{G}(\mathbf{x})]_i = \int [\nabla \left(\sum_{j=1}^{N} x_j v_j \right) \nabla v_i - f v_i] \,, \tag{6.20}$$

if variable $i = 1, \ldots, N$ is not related to a Dirichlet point and

$$[\mathbf{G}(\mathbf{x})]_i = 0 \tag{6.21}$$

otherwise. Moreover

$$[\mathbf{M}]_{ij} = \int [v_j v_i], \tag{6.22}$$

if neither variable i nor variable j is related to a Dirichlet point, otherwise

$$[\mathbf{M}]_{ij} = \delta_{ij}, \tag{6.23}$$

where δ_{ij} is the Kronecker Delta. $\mathbf{M}$ is the so-called mass-matrix.

The Jacobian $D\mathbf{G}$ of $\mathbf{G}$ is

$$[D\mathbf{G}]_{ij} = \int [\nabla v_j \nabla v_i] \tag{6.24}$$

if neither variable i nor variable j is related to a Dirichlet point, and $[D\mathbf{G}]_{ij} = \delta_{ij}$, the Kronecker Delta otherwise.

Functions in the linear vector space V do not need to satisfy the Dirichlet boundary conditions. Let $\alpha\colon \partial\Omega \mapsto \mathbf{R}$. Define the set (not necessarily a vector space)

$$V^{\alpha} = \{x \in V : x = \alpha \; at \; \partial\Omega\}. \tag{6.25}$$

Then V^0 (homogeneous boundary conditions) is a vector space, and V^{γ} is the set of all function which satisfy the Dirichlet boundary conditions.

The solution $\mathbf{z}$ of $\mathbf{G}(\mathbf{z}) = \mathbf{0}$ is obtained by application of Newton's method

$$\mathbf{z} -> \mathbf{z} + [D\mathbf{G}(\mathbf{z})]^{-1}[-\mathbf{G}(\mathbf{z})] \tag{6.26}$$

to an initial guess $\mathbf{z}^{(0)}$. The NumLab code for the above is:

```
Newton.setO(G).setSolver(R);
Newton.eval(z, 0);
```

In this code, Richardson's iterative solver R is used to solve $\mathbf{G}(\mathbf{x}) = \mathbf{0}$.

Note that operator $\mathbf{G}$ in (6.20) maps V onto V^0, and that its Jacobian matrix $D\mathbf{G}$ in (6.24) maps V^0 onto V^0. Assume that the (iterative) solver for the solution of the linear system maps (1) V^0 onto V^0 and (2) is the identity on $V - V^0$. Then, by induction, also (6.26) satisfies both assumptions. Because all common linear solvers such as PCG, GCGLS, CGS, etc. map V^0 onto V^0, all of NumLab's solvers map V^{γ} onto V^{γ}. This holds for the (non-)linear (iterative) solvers, as well as for the solvers of systems of ordinary differential equations.

For linear systems $\mathbf{G}(\mathbf{z}) = \mathbf{0}$ with $\mathbf{G}$ affine, as for instance in (6.20), the use of Newton's method (6.26) may seem an overkill. However, this is not the case: Under the assumption that all coefficients of $\mathbf{x}$ are degrees of freedom – including those related to Dirichlet points – the solution of $\mathbf{G}(\mathbf{z}) = \mathbf{0}$ requires the solution of (6.26) above.

In order to see this, define the linear system $\mathbf{F}(\mathbf{x}) = \mathbf{f}$ with

$$[\mathbf{F}(\mathbf{x})]_i = \int [\nabla \left(\sum_{j=1}^{N} x_j v_j \right) \nabla v_i] \,, \tag{6.27}$$

and

$$[\mathbf{f}]_i = \int [f v_i] \,, \tag{6.28}$$

for all i. The problem $\mathbf{F}(\mathbf{x}) = \mathbf{f}$ has no unique solution because $\mathbf{F}$ is singular. $\mathbf{F}(\mathbf{x}) = \mathbf{f}$ is below transformed in a standard manner, which results in the system $\mathbf{G}(\mathbf{x}) = \mathbf{0}$, and requires solution method (6.26).

First, define the projection $C\colon \mathbf{R}^N \mapsto \mathbf{R}^N$ by

$$\begin{array}{rcll} [\mathbf{C}(\mathbf{x})]_i & = & x_i & \text{for all related non-Dirichlet supports } \mathbf{c}_i \,, \\ & = & 0 & \text{elsewise} \,. \end{array} \tag{6.29}$$

Next, the vector $\mathbf{x}$ is coefficient-wise split into a vector which contains all Dirichlet related function values $\mathbf{x}^{(0)}$ and interior degrees of freedom $\mathbf{d}$, i.e., we set

$$\mathbf{x} = \mathbf{x}^{(0)} + \mathbf{d}. \tag{6.30}$$

Then $\mathbf{x}^{(0)}$ turns out to be the solution to

$$((I - \mathbf{C}) + \mathbf{C}D\mathbf{F}(\mathbf{x}^{(0)})\mathbf{C}^T)\mathbf{d} = \mathbf{C}(\mathbf{f} - \mathbf{F}(\mathbf{x}^{(0)})) \,. \tag{6.31}$$

This shows that for the solution of a linear boundary value problem $\mathbf{F}(\mathbf{x}) = \mathbf{f}$, we must solve (6.26), and in fact exactly solve $\mathbf{G}(\mathbf{x}) = \mathbf{0}$.

The standard splitting (6.30) for linear systems makes use of an $x^{(0)} \in V^0$ (in (6.31)) which is zero at all nodal points, except those at the Dirichlet boundary. This so called elimination of boundary conditions is a poor choice because x_0 has steep gradients near Dirichlet boundaries, whence the induced initial residual $\mathbf{r}^{(0)}$ for the iterative solver is large. Fortunately, from (6.30) it follows that we can also take different $x^{(0)} \in V^0$. In order to minimize the amount of iterative solution steps, one can use a smooth $x^{(0)}$.

Finally, we consider the NumLab formulation of (6.17). Under the assumption that the solution $\mathbf{x} \in V$, equation (6.17) is equivalent to the autonomous ODE

$$\mathbf{M}\frac{d}{dt}\mathbf{x} = -\mathbf{G}(\mathbf{x}) \qquad (t > 0), \tag{6.32}$$

where $\mathbf{x}(0)$ is the coefficient vector related to the initial condition. At its turn, (6.32) is equivalent to (6.13) for $\mathbf{E}(t, \mathbf{x}) = -\mathbf{G}(\mathbf{x})$ and $\mathbf{T}(\mathbf{x}) = \mathbf{M}(\mathbf{x} - \mathbf{x}^{(k)}) - h\mathbf{E}(t_{k+1}, \mathbf{x})$. Therefore, the initial boundary value problem (6.17) reduces to a sequence of systems of non-linear equations. Due to the particular choice of $\mathbf{T}$, its Jacobian is symmetric positive definite for small positive h, if $\mathbf{G}$'s Jacobian has this property.

6.4.4 Conclusions

The examples in sections 6.4.1, 6.4.2 and 6.4.3 have shown how mathematical problems with a seemingly different formulation can be reduced to the two basic operations of vector space computations and operator evaluation. Because of this, the NUMLAB software provides the basic notions as well as concrete specialisations of vector spaces V and operators $\mathbf{G}$ on V.

6.5 From the Mathematical to the Software Framework

In this section, we show how the notion of operators $\mathbf{F}$ and arguments $\mathbf{v}$ in (cross-product) spaces $\mathbf{V}$ map to a software framework. As outlined in the previous section, a large class of solution methods for problems of the form $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, can be reduced to a simple mathematical framework based on finite dimensional linear vector spaces and operators on those spaces. The software framework we propose will closely follow the mathematical model. As a consequence, the obtained software product will be simple and generic as well.

Consider the mathematical framework for spaces $\mathbf{V}$ and operators $\mathbf{F}$ in more detail. In general, let Ω be the bounded polygonal/polyhedral domain of interest, with smooth enough boundary $\partial\Omega$. The linear vector space $\mathbf{V} = V_1 \times \cdots \times V_n$ is a cross-product space of n spaces (n is the amount of degrees of freedom). Each space V_i is spanned by basis functions $\{v_{ij}\}_{j=1}^{N_i}$ where $v_{ij} : \Omega \mapsto \mathbf{R}$. An element $\mathbf{x} \in \mathbf{V}$ is a vector function from Ω to $\mathbf{R}^n$, and is written as $\mathbf{x} = [x_1, \ldots, x_n]$, a vector of component functions. Each component $x_i \in V_i$ is a linear combination of basis functions: for all $\mathbf{c} \in \Omega$

$$x_i(\mathbf{c}) = \sum_{j=1}^{N_i} x_{ij}(t) v_{ij}(\mathbf{c}). \tag{6.33}$$

Each element x_i is associated with a unique scalar vector $X_i = [x_{i1}, \ldots, x_{iN_i}] \in \mathbf{R}^{N_i}$. At its turn, $\mathbf{X}$ denotes the aggregate of these vectors: $\mathbf{X} = [X_1, \ldots, X_n]$, and $\mathbf{X}_{ij} = [X_i]_j$. Summarised, we have vector functions $\mathbf{x} = [x_1, \ldots, x_n]$ and related vectors of coefficient vectors $\mathbf{X} = [X_1, \ldots, X_n]$.

Whenever $n = 1$, we use a more standard notation. In this case, the space is V, spanned by basis functions $\{v_j\}_{j=1}^{N}$, and elements $x \in V$ are related to coefficient vector $\mathbf{x} = [x_1, \ldots, x_N]$.

For most finite element computations, the basis functions v_{ij} of V_i have local support. However,basis functions have global support in spectral finite elements computations. The local supports, also called elements, are created with the use of a triangulation algorithm.

The next subsections describe NUMLAB's software components related to the mathematical concepts discussed in this section: Grid generation in section 6.5.1, bases generation in section 6.5.2, vector functions in section 6.5.3, and related operators in section 6.5.4.

6.5.1 The `Grid` module

To be able to define local support for the basis functions v_{ij} later on, we need to discretise the function's domain Ω. This is modelled in the software framework by the `Grid` module, which covers the function's domain with elements e. This `Grid` module takes a `Contour` as input, which describes the boundary $\partial\Omega$ of Ω. The default contour is the unit square's contour.

In NUMLAB, the grid covers regions Ω in any dimension (e.g. 2D planar, manifold or 3D spatial), and consists of a variety of element shapes, such as triangles, quadrilaterals, tetrahedra, prisms, hexahedrals, n-simplices (see [68]), and so on. All grids implement a common interface. This interface provides a few services. These include: Iteration over the grid elements and their related vertices, topological queries such as the element which contains a given point. The amount of services is a minimum: Modules which use a grid generator and need more service must compute the required relations from the provided information.

Specific `Grid` generator modules produce grids in different manners. NUMLAB contains Delaunay generators, simplicial generators, and regular generators, and ”generators” which read an existing grid from a file.

6.5.2 The `Space` module

The linear vector space $\mathbf{V}$ is implemented by the software module `Space`. `Space` takes a `Grid` and `BoundaryConditions` as inputs. The grid's discretisation in combination with the boundary conditions are used to build the supports of its basis functions v_{ij}. The default boundary conditions are homogeneous Dirichlet type conditions for all solution components. In general, Dirichlet, Robin, Neumann, vectorial of no boundary conditions can be specified per boundary part. Recall that elements in $\mathbf{V}$ do not have to satisfy the Dirichlet boundary conditions. Recall that elements of $\mathbf{V}$ do not have to satisfy the essential boundary conditions.

Because `Grid` has a minimal interface, some information – required by `Space` for the construction of the basis functions – is not provided. Whenever this happens, `Space` internally computes the required information with the use of `Grid`'s services. A specific `Space` module implements a specific set of basis functions, such as constant, linear, quadratic, or even higher order polynomial degree, matched to the elements' geometry. The interface of the `Space` module follows the mathematical properties of the vector space $\mathbf{V}$ presented so far: Elements $\mathbf{x}, \mathbf{y} \in \mathbf{V}$ can be added together or scaled by real values. Furthermore, elements v_{ij} of $\mathbf{V}$ are functions, and $\mathbf{V}$ permits evaluation at points $\mathbf{c} \in \Omega$ of such functions and their derivatives.

It should be kept in mind that elements of $\mathbf{V}$ are functions, not linear combinations of functions. Therefore, the name SPACE is somewhat misleading. However, for the brevity of demonstration, the name SPACE will also be used in the sequel.

In most cases, the required basis functions have local support, also called element-wise support. The restriction of global basis function v_{ij} to support e is said to be local function v_{ir}. In software, this is coded as follows: For space component i (so V_i), element e, and local basis function r thereon, `j := j(i, r)` induces basis function v_{ij}. The software implementation is on element-level for efficiency purposes: Given a point $\mathbf{c} \in \Omega$, `Space` determines which support e contains $\mathbf{c}$ for the evaluation of $v_{ij}(\mathbf{c})$.

6.5.3 The `Function` module

As discussed, a vector function $\mathbf{x}\colon \Omega \mapsto \mathbf{R}^n$ in a space $\mathbf{V}$ generated by v_{ij} is uniquely related to a coefficient vector $\mathbf{X}$ with coefficients X_{ij}. Based on this observation, NUMLAB software module `Function` implements a vector function $\mathbf{x}$ as a block vector of real-valued coefficients $\mathbf{X}_{ij}$, combined with a reference to the related `Space` – which contains related functions v_{ij}.

The `Function` module provides services to evaluate the function and its derivatives at a given point $\mathbf{c} \in \Omega$. To this end, both $\mathbf{x}$'s coefficient vector X and the point $\mathbf{c}$ are passed to the `Space` module referred to by $\mathbf{x}$. At its turn, the `Space` module returns the value of $\mathbf{x}(\mathbf{c})$. This is computed following the definition $\mathbf{x}(\mathbf{c}) = [\sum_j x_{ij} v_{ij}(\mathbf{c})]$, as described in the previous section. The computation of the partial derivatives of a given function $\mathbf{x}$ in a point $\mathbf{c}$ follows a similar implementation.

Providing evaluation of functions $\mathbf{x} \in \mathbf{V}$ and of their derivatives at given points is, strictly speaking, the minimal interface the `Space` module has to implement. However, it is sometimes convenient to be able to evaluate a function at a point given as an element number and local coordinates within that element. This is especially important for efficiency in the case where one operation is iterated over all elements of a `Grid`, such as in the case of numerical integration. If the `Space` module allows evaluating functions at points specified as elements and local element coordinates, the implementation of the numerical integration is considerably faster than when point-to-element location has to be performed. Consequently, we also provided the `Space` module with a function evaluation interface which accepts an element number and a point defined in the element local coordinates.

6.5.4 The `Operator` module

As described previously, an operator $\mathbf{F}\colon \mathbf{V} \mapsto \mathbf{W}$ maps an element $\mathbf{x} \in \mathbf{V}$ to an element $\mathbf{z} \in \mathbf{W}$. The evaluation $\mathbf{z} = \mathbf{G}(\mathbf{x})$ computes the coefficients z_{ij} of $\mathbf{z}$ from the coefficients x_{ij} of $\mathbf{x}$, as well as from the bases $\{v_{ij}\}$ and $\{w_{ij}\}$ of $\mathbf{V}$ and $\mathbf{W}$ respectively. Next to the evaluation of $\mathbf{G}$, derivatives such as the Jacobian operator $D\mathbf{G}$ of $\mathbf{G}$ are evaluated in a similar manner. Such derivatives are important in several applications. For example, they can be used in order to find a solution of $\mathbf{G}(\mathbf{z}) = \mathbf{x}$, with Newton's method.

The software implementation of the operator notion follows straightforwardly the mathematical concepts introduced in Section 6.4. The implementation is done by the `Operator` module, which offers two services: evaluation of $\mathbf{z} = \mathbf{G}(\mathbf{x})$, coded as `G.eval(z,x)`, and of the Jacobian of $\mathbf{G}$ in point $\mathbf{y}$, $\mathbf{z} = D\mathbf{G}(\mathbf{y})\mathbf{x}$, coded as `G.getJ(y).eval(z,x)`. To evaluate $\mathbf{z} = \mathbf{G}(\mathbf{x})$, the `Operator` module takes two `Function` objects $\mathbf{z}$ and $\mathbf{x}$ as input and computes the coefficients z_j using the coefficients x_{ij} and the bases of the `Space` objects $\mathbf{z}$ and $\mathbf{x}$ carry with them. It is important that both the 'input' $\mathbf{z}$ and the 'output' $\mathbf{x}$ of the `Operator` module are provided, since it is in this way that `Operator`s determine the spaces $\mathbf{V}$, respectively $\mathbf{W}$.

To evaluate $\mathbf{z} = D\mathbf{G}(\mathbf{y})\mathbf{x}$, the `Operator` proceeds similarly. Internally, $D\mathbf{G}(\mathbf{y})$ is usually implemented as a coefficient matrix, and the operation $D\mathbf{G}(\mathbf{y})\mathbf{x}$ is a matrix-vector multiplication. However, the implementation details are hidden from the user ($D\mathbf{G}(\mathbf{y})\mathbf{x}$ may be computed element-wise, i.e. matrix-free), who works only with the `Function` and `Operator` mathematical notions.

Specific `Operator` implementations differ in the way they compute the above two evaluations. For example, a simple `Diffusion` operator $\mathbf{z} = \mathbf{G}(\mathbf{x})$ may operate on a scalar function and produce a function $\mathbf{z}$ where $z_i = x_{i-1} - 2x_i + x_{i+1}$. A generic `Linear` operator may produce a vector of coefficients $\mathbf{z} = \mathbf{A}\mathbf{x}$ where $\mathbf{A}$ is a matrix. A `Summator` operator $\mathbf{z} = \mathbf{G}_1(\mathbf{x}) + \mathbf{G}_2(\mathbf{x})$ may take two inputs $\mathbf{G}_1$ and $\mathbf{G}_2$ and produce a vector of coefficients $z_i = [\mathbf{G}_1(\mathbf{x})]_i + [\mathbf{G}_2(\mathbf{x})]_i$. Remark that the modules implementing the `Linear` and `Summator` operators actually have two inputs each. In both cases the function $\mathbf{x}$ is the first input, while the second is the matrix $\mathbf{A}$ for the `Linear` operator and the operators $\mathbf{G}_1$ and $\mathbf{G}_2$ for the `Summator` operator. These values could be as well hard-coded in the operator implementation. In both cases however, we see `Operator` as a function of a *single* variable $\mathbf{x}$, as described in the mathematical framework.

6.5.5 The `Solver` module

We model the solving of $\mathbf{G}(\mathbf{z}) = \mathbf{x}$ by the module `Solver` in our software framework. Mathematically speaking, `Solver` is similar to an operator $\mathbf{S}\colon \mathbf{V} \mapsto \mathbf{W}$, where $\mathbf{V}$ and $\mathbf{W}$ are function spaces. The interface of `Solver` provides evaluation at functions $\mathbf{x} \in \mathbf{W}$, similarly to the `Operator` module. The implementation of the `Solver` evaluation operation $\mathbf{z} = \mathbf{S}(\mathbf{x})$ should provide an approximation $\mathbf{z}$ to $\mathbf{z} \approx \mathbf{F}^{-1}(\mathbf{x})$. However, `Solver` does not provide evaluation of its Jacobian, as this may be undesirably complex to compute in the general case.

Practically, `Solver` takes as input an initial guess `Function` object $\mathbf{x}$ and an `Operator` object $\mathbf{G}$. Its output $\mathbf{z}$ is such that $\mathbf{G}(\mathbf{z}) = \mathbf{x}$. The operations done by the solver are either vector space operations or `Operator` evaluations, or evaluations of similar operators $\mathbf{G}(\mathbf{z})$. In the actual implementation, this is modelled by providing the `Solver` module with one or more extra inputs of type `Solver`. In this way, one can for example connect a nested chain of preconditioners to an iterative solver module.

The implementation of a specific `Solver` follows straightforwardly from its mathematical description. Iterative solvers such as Richardson, GMRES, (bi)conjugate gradient, with or without preconditioners, are easily implemented in this software framework.

The framework makes no distinction between a solver and a preconditioner, as discussed in Section 6.4. The sole difference between a solver and a preconditioner in this framework is semantic, not structural. A solver is supposed to produce an *exact* solution of $\mathbf{G}(\mathbf{z}) = \mathbf{0}$ (up to a desired numerical accuracy), whereas the preconditioner is *supposed* to return an *approximate* one. Both are implemented as `Solver` modules, which allows easy cascading of a chain of preconditioners to an iterative solver as well as using preconditioners and solvers interchangeably in applications. Furthermore, the framework makes no structural distinction between direct and iterative solvers. For example, an `ILUSolver` module is implemented to compute an incomplete LU factorisation of its input operator $\mathbf{G}$. The `ILUSolver` module can be used as a preconditioner for a `ConjugateGradient` solver module. In the case the `ILUSolver` is not connected to the `ConjugateGradient` module's input, the latter performs non preconditioned computations. Alternatively, a `LUSolver` module is implemented to provide a complete LU factorisation of its input operator $\mathbf{G}$. The `LUSolver` can be used either directly to solve the equation $\mathbf{G}(\mathbf{z}) = \mathbf{x}$, or as preconditioner for another `Solver` module.

6.6 An object-oriented approach to the software framework

So far, we have outlined the structure of the proposed numerical software framework. This structure is based upon a few basic modules which parallel the mathematical concepts of `Grid`, `Function`, `Space`, `Operator`, and `Solver`. These modules provide their functionality via interfaces containing a small number of operations, such as the `Operator`'s evaluation operation or the `Grid`'s element-related services previously outlined.

As stated in the beginning of this section, a large range of numerical problems can be modelled with these few generic modules. In order to capture the specifics of a given problem, such as the type of PDE to be solved or the basis functions of an approximation space, the generic modules have to be specialised. The specialised modules provide the interface declared by their class, but can implement it in any desirable fashion. For example, a `ConjugateGradient` module implements the `Solver` interface of evaluating $\mathbf{z} = \mathbf{G}^{-1}\mathbf{x}$ by using the conjugate gradient iterative method.

The above architectural requirements are elegantly and efficiently captured by using an object-oriented approach to software design [16, 91, 69, 12]. Consequently, we have implemented our numerical software framework as an object-oriented library written in the C++ language [102]. This design enabled us to naturally model the concepts of basic and specialised modules as class hierarchies. The software framework implements a few base classes `Grid`, `Function`, `Space`, `Operator`, and `Solver`. These base classes declare the interface to their operations. The interface is next implemented by various specialisations of these base classes. An overview of the implemented specialisations follows:

- `Grid:` 2D and 3D grid generators for regular and unstructured grids, and grid file readers;
- `Function:` Several specific functions v_ij are generated, such as sines, cosines, or piecewise (non-)conforming polynomial functions in several dimensions;
- `Space:` There is a single `Space` class, but a multitude of basis functions are implemented, as described further in Section 6.8;
- `Operator:` Operators for several ODEs, PDEs, and non-linear systems have been implemented, such as Laplace, Stokes, Navier-Stokes, and elasticity problems. Next, several operators for matrix manipulation and image processing have been implemented. For example, matrix sparsity patterns can be easily visualised, as in other applications like Matlab.
- `Solver:` A range of iterative solvers including bi-conjugate gradient, GMRES, GCGLS, QMR, etc. are implemented. Several preconditioners such as ILU are also provided as `Solver` specialisations, following the common treatment of solver and preconditioner modules previously described.

Besides the natural modelling of the mathematics in terms of class hierarchies, the object-oriented design allows users to easily extend the current framework with new software modules. Implementing a new solver, preconditioner, or operator usually involves writing only a few tens of lines of C++ to extend an existing one. The same approach also facilitates the reuse of existing numerical libraries such as LAPACK [4] or Templates [11] by integrating them in the current object-oriented framework.

6.7 Transient Navier-Stokes equations

This section examines the mathematical concepts at the foundations of a NUMLAB solver for transient Navier-Stokes equations. These concepts (1) – (4) in section 6.4, have been examined in sections 6.4.1 – 6.4.3 for small model problems, suited for presentation purposes. Here, these concepts are all worked out in relation to a single problem, the solution of transient Navier-Stokes equations. Section 6.8 discusses the design of a NUMLAB application with the NUMLAB operators discussed here.

The transient Navier-Stokes equations have been chosen since related finite element operators require a finite dimensional cross product vector space $\mathbf{V}$ of basis functions, and because the transient formulation leads to differential algebraic equations, and requires solution techniques related to ODEs. The DAE class of equations is non-trivial to solve, and common in industrial problems. Our claim is – see section 6.8 for details – that NUMLAB provides a sophisticated framework for the integration of complex Navier-Stokes solvers, not that NUMLAB provides solvers better than those found in the literature.

First we examine the static problem. For a particular discretisation, we show that there exists a straightforward and lucid relation between the mathematical formulas and the NUMLAB software implementation: The NUMLAB implementation of $\mathbf{F}$ accomplishes the finite element required (numerical) integration without space $\mathbf{V}$ exposing its basis functions and element geometries to $\mathbf{F}$.

The static case is followed with the mathematical formulation of the transient problem. We demonstrate that (components of) the static problem operator $\mathbf{F}$ can be used in combination with all suitable time-integrators $\mathbf{S}$ – suited for indefinite/stiff problems.

Due to the high degree of orthogonality between $\mathbf{F}$, $\mathbf{V}$ and time-stepper methods, NUMLAB can and does offer a range of finite element types – higher order, as well as non-conforming Crouzeix-Raviart (see [24]) – on rather arbitrary support geometries: simplices, parallelipipeda, prisms, etc. It facilitates and supports user-defined reference bases and geometries, as well as user-supplied geometries and grid generators. Existing applications do not have to be adapted for new bases and geometries, as long as all required mathematical conditions hold.

6.7.1 The Navier-Stokes equations

The incompressible Navier-Stokes equations describe an incompressible fluid subject to forces $\mathbf{f}$. For the sake of brevity, we assume the case of a finite 2D domain, $\Omega \subset \mathbf{R}^2$. We denote the fluid velocities by $\mathbf{u} = [u_1, u_2]$, and the pressure by p. To start with, we consider the stationary case. The classical problem is to find sufficiently smooth $(\mathbf{u}, p)$ such that in Ω:

$$\left\{ \begin{array}{rcl} -\epsilon\Delta\mathbf{u} + \mathbf{u}\nabla\mathbf{u} + \nabla p &=& \mathbf{f}, \\ \nabla\cdot\mathbf{u} &=& 0. \end{array} \right. \tag{6.34}$$

For the sake of exposition, all boundary conditions are presumed to be of Dirichlet type (parabolic in/outflow profiles and no-slip along walls).

Problem (6.34) is discretised by a finite element method. To this end, one first covers Ω by elements with the use of a grid generator module `Grid` (the construction and refinement of a suitable computational grid is a problem of its own (see for instance [32]). Then a triplet of finite dimensional (Hilbert) finite element spaces $\mathbf{V} := V_1 \times V_2 \times V_3$ is chosen such that $V_1 \times V_2$ and V_3 satisfy the L.B.B. condition [5]. The NUMLAB implementation creates one `Space` module $\mathbf{V}$, provided with three reference `Basis` modules. For the sake of presentation, quadratic conforming finite element bases are used for the velocities (V_1 and V_2), and a piecewise linear conforming finite element basis is used for the pressure (V_3).

Next, the equations (6.34) are multiplied by test functions $(\mathbf{v}, q) \in \mathbf{V}$, after which the first one is partially integrated. This procedure results in a variational problem: Find $\mathbf{x} = [u_1, u_2, p] = (\mathbf{u}, p) \in \mathbf{V}$ such that for all $(\mathbf{v}, q) \in \mathbf{V}$

$$\left\{ \begin{array}{rcl} \displaystyle\int \epsilon\nabla\mathbf{u} : \nabla\mathbf{v} - p\nabla\mathbf{v} + (\mathbf{u}\nabla\mathbf{u} - \mathbf{f})\mathbf{v} &=& \mathbf{0}, \\ \displaystyle\int \nabla\cdot\mathbf{u}\, q &=& 0. \end{array} \right. \tag{6.35}$$

Various other finite element discretisations of (6.34) are also possible. In order to facilitate the formulation of a NUMLAB application for our problem, system (6.35) is now reformulated into operator form: $\mathbf{F}(\mathbf{X}) = \mathbf{0}$.

The operator $\mathbf{F}$ related to the discrete variational formulation (6.35) has three components $\mathbf{F} := [\mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3]$, each related to one equation. For the definition of these components, first define $\mathbf{x} = [x_1, x_2, x_3] := [u_1, u_2, p] \in \mathbf{V}$, and set $\mathbf{z} = [z_1, z_2, z_3] \in \mathbf{V}$ (assume we use a Galerkin procedure). Recall that each vector function $\mathbf{x}$ is uniquely related to coefficients x_j, at their turn related to functions

v_{ij} from Ω to $\mathbf{R}$. The discrete Navier-Stokes operator, discretised in space, now is:

$$\begin{aligned}
Z_{1j} &= \mathbf{F}(\mathbf{X}) = [\mathbf{F}_1(X_1, X_2, X_3)]_j \\
&= \int \epsilon \nabla x_1 \nabla v_{1j} - x_3 \partial_x v_{1j} + (x_1 \partial_x x_1 + x_2 \partial_y x_1 - f_1) v_{1j} \\
Z_{2j} &= \mathbf{F}(\mathbf{X}) = [\mathbf{F}_2(X_1, X_2, X_3)]_j \\
&= \int \epsilon \nabla x_2 \nabla v_{2j} - x_3 \partial_y v_{2j} + (x_1 \partial_x x_2 + x_2 \partial_y x_2 - f_2) v_{2j} \\
Z_{3j} &= \mathbf{F}(\mathbf{X}) = [\mathbf{F}_3(X_1, X_2, X_3)]_j \\
&= \int (\partial_x x_1 + \partial_y x_2) v_{3j} \; .
\end{aligned} \tag{6.36}$$

Here, x_1 is the function related to coefficients X_1, and so forth. It is evident – as stated earlier – that $\mathbf{F}$ uses the coefficients of $\mathbf{x}$ as well as the bases functions in order to compute the coefficients of the result $\mathbf{z}$.

The integrals in (6.36) are computed support-wise, with the use numerical integration, involving integration points $\mathbf{x}_k$. As can be deduced from (6.36), required are the values $v_{i,r}(\mathbf{x}_k)$ and $\nabla v_{i,r}(\mathbf{x}_k)$ as well as the values $x_i(\mathbf{x}_k)$ and $\nabla x_i(\mathbf{x}_k)$. In the NUMLAB code below, these values are returned in arrays `v(i)(k)(r)`, `dv(i)(k)(r)`, `x(i)(k)`, respectively `dx(i)(k)`. The selection `dv(i)(k)(r)(dY)` returns the individual gradient component $\partial_y v_{i,r}(\mathbf{x}_k)$. Define `U1 = 0, U2 = 1, P = 2`. The NUMLAB evaluation of `z = F(x)` and `z = DF(x)*y` for support e (typeset to fit this layout) is:

```
Operator z = F(x):

 z(U1)(j(U1)(r)) += qw(k)* (eps*dx(U1)(k)*dv(U1)(k)(r) -
    x(P)(k)*dv(U1)(k)(r)(dX) + (x(U1)(k)*dx(U1)(k)(dX) +
    x(U2)(k)*dx(U1)(k)(dY) - f1(qp(k)) * v(U1)(k)(r)));
 z(U2)(j(U2)(r)) += qw(k)* (eps*dx(U2)(k)*dv(U2)(k)(r) -
    x(P)(k)*dv(U2)(k)(r)(dY) + (x(U1)(k)*dx(U2)(k)(dX) +
    x(U2)(k)*dx(U2)(k)(dY) - f2(qp(k)) * v(U2)(k)(r)));
 z(P)(j(P)(r)) += qw(k)* ((dx(U1)(k)(dX) + dx(U2)(k)(dY))*
    v(P)(k)(r));

Jacobian z = DF(x)*y:

 DF(U1)(U1)(j(U1)(r))(j(U1)(s)) += qw(k)* (dv(U1)(k)(s)*
    dv(U1)(k)(r) + v(U1)(k)(s)*dx(U1)(k)(dX) + x(U1)(k)*
    dv(U1)(k)(s)(dX));
       ...

 DF(P)(U2)(j(P)(r))(j(U2)(s))   += qw(k)*
    (dv(U2)(k)(s)(dY)*v(P)(k)(r));

 z = DF * y;
```

Both evaluation operations have an almost identical loop structure:

```
 V = x->getSpace();
 for (Integer e = 0; e < V->NElements(); e++)
```

```
V->fetch(e, j, v, dv, x, dx, .....);
for (Integer i = 0; i < j.size(); i++)
 for (Integer r = 0; r < j(i).size(); r++)
  for (Integer k = 0; k < x(i).size(); k++)
```

The Jacobian has an extra inner loop over trial functions `s`. With regard to this implementation, several observations come to mind:

- First, $\mathbf{F}$ does not have spaces $\mathbf{V}$ and $\mathbf{W}$ as input (i.e., as auxiliary variables). The spaces are obtained from the input/output variables. This technique simplifies computational networks.
- Secondly, because $\mathbf{F}$ performs numerical integration, it solely requires *the value* of (partial derivatives of) the basis functions at the quadrature points. The basis functions themselves are not required, so $\mathbf{F}$ operates orthogonal to $\mathbf{V}$ and $\mathbf{W}$.
- Finally, the NUMLAB operator models the discrete Navier-Stokes equations in (6.35) in a convenient fashion. The software implementation is one-to-one with the mathematical syntax, and can in fact be automated.

Finally, recall that the derivative operator acts as the identity operator on Dirichlet point related variables, which requires `fetch` to deliver the related information. This information is also required for non-homogeneous Neumann boundary conditions and Robin conditions.

6.7.2 The time discretisation

A transient version of the Navier-Stokes equations in the previous section can be formulated as so-called differential algebraical equations (DAEs):

$$\begin{cases} \frac{\partial}{\partial t}\mathbf{u} &= \epsilon\Delta\mathbf{u} - \mathbf{u}\nabla\mathbf{u} - \nabla p + \mathbf{f}, \\ \nabla\cdot\mathbf{u} &= 0, \end{cases} \qquad (t > 0) \tag{6.37}$$

with initial condition $\mathbf{u}(0,\mathbf{c}) = \mathbf{u}_0(\mathbf{c})$ on Ω and boundary conditions $u(t,\mathbf{c}) = u_1(\mathbf{c})$ for all $t \in [0,\infty)$ and $\mathbf{c} \in \partial\Omega$. We now construct a non-linear NUMLAB time-step operator $\mathbf{F}$ for a MOL discretisation of (6.37), which is implicit with respect to the constraint $\nabla\cdot\mathbf{u} = 0$.

For the sake of presentation, for a discretisation of the first vectorial equation in (6.37), we will use a rather basic time-step method: the θ-method – recall the constrained will be treated in an implicit way below. In practice, for stiff problems – high Reynolds number – one would rather use a backward difference method. For $\theta \in [0,1]$, the θ-method for a general non-linear system of ODEs

$$\frac{d}{dt}\mathbf{u}(t) = \mathbf{E}(t,\mathbf{u}(t)), \tag{6.38}$$

leads to the recursion:

$$\begin{aligned} u_0 &= \mathbf{u}(0), \\ \mathbf{u}_{k+1} - \mathbf{u}_k &= h\theta\mathbf{E}(t_k,\mathbf{u}_k) + h(1-\theta)\mathbf{E}(t_{k+1},\mathbf{u}_{k+1}). \end{aligned} \tag{6.39}$$

This all fits into the NUMLAB `Operator` style, if we define the time-step operator $\mathbf{T} := \mathbf{T}_{t_k,\mathbf{u}^{(k)},t_{k+1}}$ as follows:

$$\mathbf{T}(\mathbf{u}) := \mathbf{u} - \mathbf{u}^{(k)} - h\theta\mathbf{E}(t_k,\mathbf{u}^{(k)}) - h(1-\theta)\mathbf{E}(t_{k+1},\mathbf{u}). \tag{6.40}$$

In this way, the Jacobian of $\mathbf{T}$ is positive definite for small h, if the Jacobian of $\mathbf{E}$ is, and the approximation $\mathbf{u}^{(k+1)}$ of $\mathbf{u}(t_{k+1})$ is a root of

$$\mathbf{T}(\mathbf{u}) = \mathbf{0}. \tag{6.41}$$

For the solution of (6.37), we first discretise (6.37), in a manner similar to how (6.34) was discretised to obtain (6.36), with a discrete solution as formulated in (6.18). This leads to a discretised version of (6.37):

$$\left\{ \begin{array}{rcl} \mathbf{M}\frac{d}{dt}X_1(t) & = & -\mathbf{F}_1(\mathbf{X}(t)) \\ \mathbf{M}\frac{d}{dt}X_2(t) & = & -\mathbf{F}_2(\mathbf{X}(t)) \\ 0 & = & \mathbf{F}_3(\mathbf{X}(t)). \end{array} \right. \tag{6.42}$$

subjected to initial conditions on the two velocity components $X_1(0) = g_0$, $X_1(0) = g_1$. The operators $\mathbf{F}_i$ are those defined in (6.36), and $\mathbf{M}$ is the mass matrix. Define $\mathbf{Y}(t) = [X_1(t), X_2(t)]$, i.e., $\mathbf{X}(t) = [\mathbf{Y}(t), X_3(t)]$, and operator $\mathbf{E}(\mathbf{X}) := [-\mathbf{F}_1(\mathbf{X}), -\mathbf{F}_2(\mathbf{X})]$, then (6.42) reduces to

$$\left\{ \begin{array}{rcl} \mathbf{M}\frac{d}{dt}\mathbf{Y} & = & \mathbf{E}(\mathbf{X}) \\ 0 & = & \mathbf{F}_3(\mathbf{X}), \end{array} \right. \tag{6.43}$$

with related initial and boundary conditions. Thus, when we apply the θ-method (6.40), every approximate solution $\mathbf{X}^{(k+1)} = [\mathbf{Y}^{(k+1)}, X_3^{(k+1)}]$ must both solve $\mathbf{G}(\mathbf{X}) := [\mathbf{T}(\mathbf{X}), \mathbf{F}_3(\mathbf{X})] = [\mathbf{0}, 0]$, where

$$\mathbf{T}(\mathbf{X}) := \mathbf{Y} - \mathbf{Y}^{(k)} - h\theta\mathbf{E}(t_k, \mathbf{X}^{(k)}) - h(1-\theta)\mathbf{E}(t_{k+1}, \mathbf{X})\,. \tag{6.44}$$

Summarising (6.37) – (6.44), we have shown that each approximate solution $\mathbf{X}^{(k)}$ of $\mathbf{X}(t_k)$ must solve a non-linear system of equations $\mathbf{G}(\mathbf{X}) = \mathbf{0}$. The related non-linear operator $\mathbf{G}$ can be supplied to a NUMLAB non-linear solver.

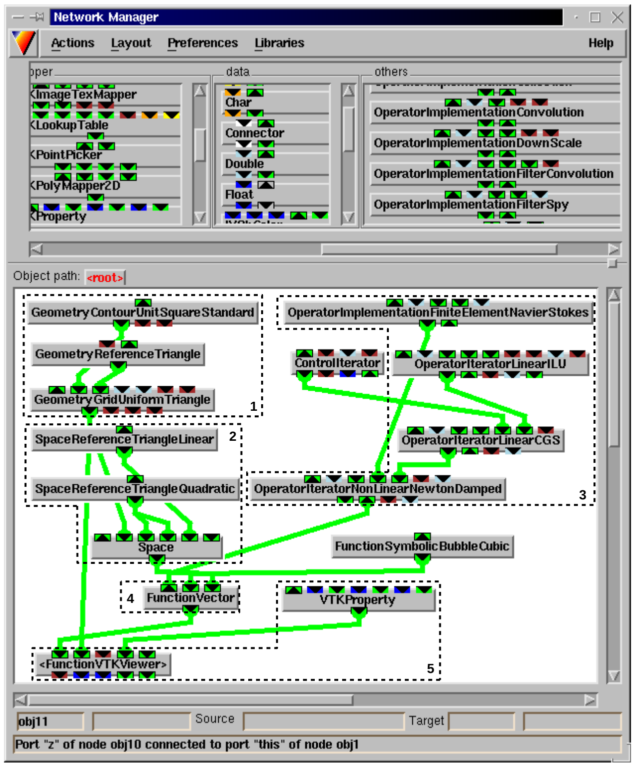
Finally, some remarks and observations. First, the value which operator $\mathbf{G}$ attains at $\mathbf{X}$, is composed of the values which $\mathbf{F}$ attains at related points. Therefore, the Jacobian $D\mathbf{G}(\mathbf{X})$ can be formulated in terms of $D\mathbf{F}$ at related points. The NUMLAB implementation of time-steps exploits this: The Jacobian $D\mathbf{G}(\mathbf{X})$ is a sequence of call-backs to the Jacobians $D\mathbf{F}$. Secondly, a NUMLAB discretised system of partial differential equations leads to an operator $\mathbf{F}(\mathbf{X})$ which can be *the* operator provided to an ODE step method. In this case, the PDE and ODE discretisations are used strictly orthogonal in the software implementation.

The saddle point problems related to (6.42) are hard to solve ([90], [32]).

6.8 Application design and use

The previous sections have presented the structure of the NUMLAB computational framework. It has been shown how new algorithms and numerical models can easily be embedded in the NUMLAB framework, due to its design based on few generic mathematical concepts. This section treats the numerical application construction and use with the NUMLAB system.

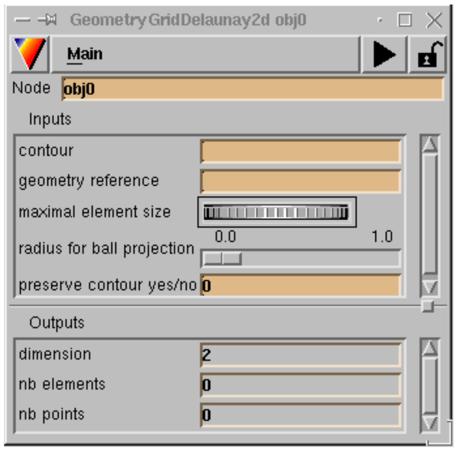
As stated in section 6.3, a numerical framework should provide an easy way to construct numerical experiments by assembling predefined components such as grids, problem definitions, solvers, and preconditioners. Next, one should be able to interactively change all parameters of the constructed application and monitor the produced results in a numerical or visual form. Shortly, we need to address the three roles of component development, application design, and interactive use for the scientific computing domain.


Network Manager
Actions
Layout
Preferences
Libraries
Help
oper
data
others
ImageTexMapper
LookupTable
PointPicker
PolyMapper2D
Property
Char
Connector
Double
Float
OperatorImplementationConvolution
OperatorImplementationDownScale
OperatorImplementationFilterConvolution
OperatorImplementationFilterSpy
Object path:
<root>
GeometryContourUnitSquareStandard
GeometryReferenceTriangle
GeometryGridUniformTriangle
1
SpaceReferenceTriangleLinear
2
SpaceReferenceTriangleQuadratic
OperatorImplementationFiniteElementNavierStokes
ControlIterator
OperatorIteratorLinearILU
OperatorIteratorLinearCGS
OperatorIteratorNonLinearNewtonDamped
3
Space
FunctionSymbolicBubbleCubic
4
FunctionVector
VTKProperty
<FunctionVTKViewer>
5
Source
Target

obj11

Port "z" of node obj10 connected to port "this" of node obj1

a)



Node	obj0
contour	
geometry reference	
maximal element size	
radius for ball projection	
preserve contour yes/no	0
dimension	2
nb elements	0
nb points	0

b)

Figure 6.4: a) Navier-Stokes simulation built with NUMLAB components. b) User interface for the grid generator module

We have approached the above by integrating the NUMLAB component library in the VISSION system. As NUMLAB is written as a C++ component library, its integration into VISSION was easy. Moreover, the structure of NUMLAB as a set of components that communicate by data streams in order to perform the desired computation matches well VISSION's dataflow application model. As no modification of the NUMLAB code was necessary, its integration in VISSION took only a few hours of work.

Once all the NUMLAB components were integrated into VISSION, constructing numerical applications with interactive computational steering and visualisation was easily achieved by using VISSION's visual network construction and end user interaction facilities described in chapter 4. We illustrate this next with the Navier-Stokes problem discussed in the previous section.

6.8.1 The Navier-Stokes simulation

As outlined previously, numerical applications built with the NUMLAB components are actually VISSION dataflow networks. Figure 6.4 a) shows such a network built for the Navier-Stokes problem. The modules in the Navier-Stokes computational network in Fig. 6.4 are arranged in five groups. The functionality of these groups is explained in the following.

6.8.2 The computational domain

The first group contains modules that define the geometry of the computational domain. These modules accomplish three functions:

1. definition of the computational domain's contour.
2. definition of the reference geometric element.
3. mesh generation

In our example, the computational domain is a rectangular region whose boundary is defined by the `GeometryContourUnitSquareStandard` module. This module allows the specification of the rectangle's sizes, as well as a distribution of mesh points on the contour. Next, the `Geometry-GridUniformTriangle` module produces a meshing of the rectangle into triangles. The reference triangle geometry is given by the `GeometryReferenceTriangle`. The mesh produced by the `GeometryGridUniformTriangle` module conforms both to the reference element supplied as input and to the boundary points output by the `GeometryContourUnitSquareStandard` module. Different combinations of contour definitions, mesh generators, and reference elements are easily achieved by using different modules. In this way, 2D and 3D regular and unstructured meshes of various element types such as triangles, quadrilaterals, hexahedra, or tetrahedra can be produced. The produced mesh can be directly visualised or further used to define a computational problem.

6.8.3 Function spaces

The second group contains modules that define the function space V over the computational domain. The modules in this group perform two functions:

1. definition of a set of basis functions v_i that span V.
2. definition of V from the basis functions and the discretised computational domain.

The first task is done by the `SpaceReferenceTriangleLinear` and `SpaceReferenceTri-angleQuadratic` modules, which define linear, respectively quadratic basis functions on the geometric triangles. The functions are next input into the `Space` module, which has already been discussed in the previous sections. The support of the basis functions is defined by the computational domain's discretisation which is also input into `Space`. In our case, `Space` uses the quadratic basis function module twice and the linear basis function module once, as the 2D Navier-Stokes problem has two velocity components to be approximated quadratically and one linearly approximated pressure component.

An important advantage of the design of NumLab is the orthogonal combination of basis functions and geometric grids. Several other (e.g. higher order) basis function modules are provided as well, defined on different geometric elements. By combining them as inputs to the `Space` module, one can easily define a large range of approximation spaces for various computational problems. In the case of a diffusion PDE solved on a grid of quadrilaterals, for example, one would use a single `Space-ReferenceQuadLinear` basis function input to the `Space` module.

6.8.4 Operators and solvers

The third group contains modules that define the function $\mathbf{F}$ for which the equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is to be solved, as well as the solution method to be used. This group contains thus specialisations of the `Operator` and
tt Solver modules described in the previous sections.

In our example, the discrete formulation of (6.36) discussed in the previous section is implemented by the `OperatorImplementationFiniteElementNavierStokes` module. The static Navier-Stokes problem is solved by a Newton solver implemented by the `OperatorIteratorNonLinearNewtonDamped` module. The linear system output by the Newton module is then solved by a conjugate gradient solver implemented by the `OperatorIteratorLinearCGS` module. The solution is accelerated by using an incomplete LU preconditioner `OperatorIteratorLinearILU` which is passed as input to the conjugate gradient solver.

Other problems can be readily modelled by choosing other operator implementations. Similarly, to use another solution or preconditioning method, a chain of `Solver` specialisations can be constructed. As solvers have an input of the same `Solver` type, complex solution algorithms can be built on the fly.

6.8.5 Functions

The fourth group contains specialisations of the `Function` module. These model both the solution of a numerical problem as well as its initial conditions or other involved quantities such as material properties. In our example, the `FunctionVector` module holds both the velocity and pressure solution of the Navier-Stokes equation. The solution is updated at every iteration, as this module is connected to the solver module's output. As explained in the previous sections, a function is associated with a space. This is seen in the `Function`'s input connection to the `Space` module.

The solution of the problem is initialised by connecting the `FunctionSymbolicBubble` module to the `FunctionVector`'s input. When the user changes the initial solution value, by changing an input of the `FunctionSymbolicBubble` signal or by replacing it with another function, the network restarts the computations from this new value.

6.8.6 Output monitoring

The last group of modules provides visualisation facilities to the computational network. The main module in here is the `FunctionVTKViewer` meta-group which takes as input the current solution of the Navier-Stokes equation and the grid upon which it is defined. In our example, the `FunctionVTKViewer` module inputs the velocity and pressure solution components into various visualisation algorithms, such as stream lines and hedgehogs for the vectorial, respectively colour plots and isolines for the scalar component. The visualisation facilities are provided in VISSION by a separate component library described in detail in chapter 7.

Several other visualisation methods can be easily attached to the Navier-Stokes simulation, by editing the contents of the `FunctionVTKViewer` meta-group. Keeping the visualisation back-end pipeline inside a single meta-group allows a natural separation of the computational network from the post-processing operations. This also helps to reduce the overall visual complexity of the network.

6.8.7 Navier-Stokes simulation steering and monitoring

Once the Navier-Stokes computational network is constructed, one can start an interactive simulation by changing the parameters of the various modules involved, such as mesh refinement, solver tolerance, or initial solution value. All the numerical parameters, as well as the parameters of the visualisation back-end are accessible via the module interactors automatically created by VISSION (Fig. 6.4 b).

Moreover, the evolution of the intermediate solutions produced by the Newton solver can be interactively visualised. This is achieved by constructing a loop which connects the output of the `Oper-`

`atorIteratorNonLinearNewtonDamped` module to its input. The module will then change the `FunctionVector`, and thus the visualisation pipeline downstream of it, at every iteration. This allows one to interactively monitor the improvement of the solution at a given time step, and eventually change other parameters to experiment new solvers or preconditioners.

Figure 6.5 shows a snapshot from an interactive Navier-Stokes simulation. The simulation domain (Fig. 6.5 upper left) consists of a 2D rectangular vessel with an inflow and an outflow. The inflow and outflow have both parabolic essential boundary conditions on the fluid velocity. The sharp obstacle placed in the middle of the container can be interactively manipulated by the end user by dragging its tip with the mouse anywhere inside the vessel. Once the obstacle's shape is changed, the NUMLAB network re-meshes the new domain, recomputes the stationary solution for the Navier-Stokes simulation defined on this new domain, and displays the pressure and velocity solutions. Various other parameters, such as fluid viscosity, mesh refinement, and solver accuracy, can also be interactively controlled. The computational steering of the above problem proceeds at near-interactive rates. Consequently, such NUMLAB setups can be used for quick, interactive testing of the robustness and accuracy of various solvers, preconditioners, and mesh generators. For example, one can test the speed and robustness of an iterative solver for different combinations of obstacle size and shape, mesh coarseness, and fluid viscosity for the above problem.

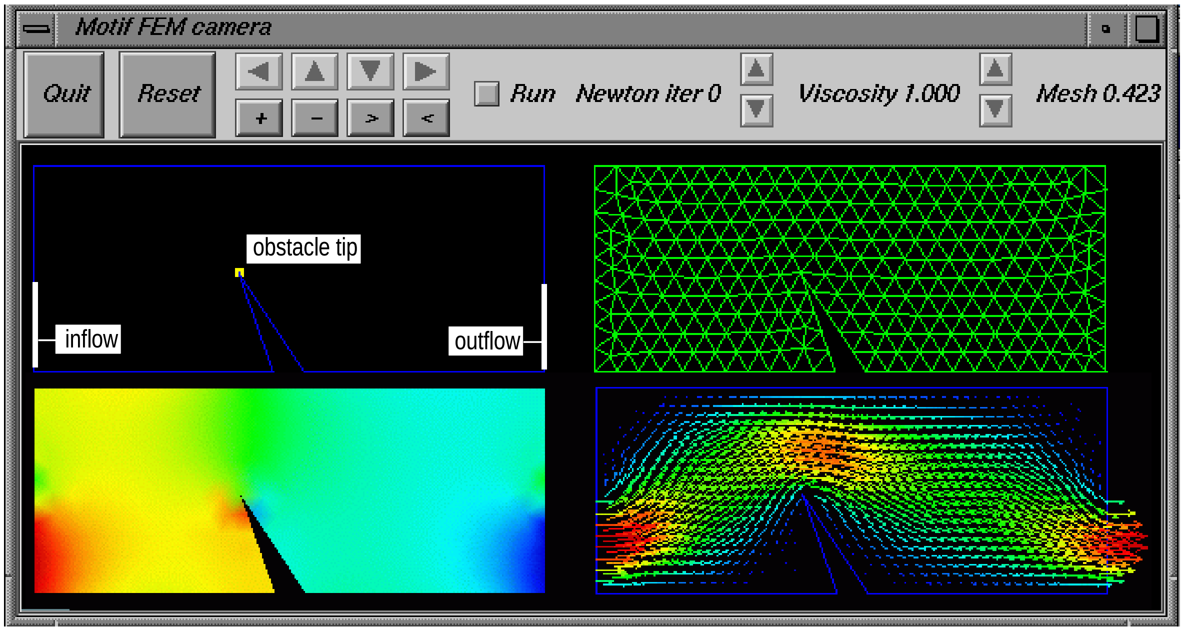


Figure 6.5: Interactive Navier-Stokes simulation: domain definition, mesh, pressure, and velocity solutions

NUMLAB can also be used for solving large computational problems. In the following example, glass pressing in the industry is considered. The process of moulding a hot glass blob pressed by a parison is simulated. The glass is modelled as a viscous fluid, subjected to the Navier-Stokes equations. The pressing simulation is a time-dependent process, where the size and shape of the computational domain is changed at every step, after which the stationary Navier-Stokes equations are solved on the new domain. The flow equations can be solved on a two-dimensional cross-section in the glass, since the real 3D domain is axisymmetric.

The simulation is analogous in many respects to the one previously presented. However, a mesh in the glass pressing simulation involves tens of thousands of finite elements, whereas the previous example used only a few hundreds. Consequently, the latter simulation can not be steered interactively. However, all computational parameters of the involved NUMLAB network can be interactively controlled at the beginning of the process, or between computation steps. Figure 6.6 shows several results of the glass pressing simulation. The first row depicts several snapshots of the 3D geometry of the moulded glass, reconstructed and realistically rendered in NUMLAB from the 2D computational domain. The second row in Fig. 6.6 shows fluid pressure snapshots taken during the 2D numerical simulation. The output of the NUMLAB visualisation pipeline can be connected to an MPEG movie creation module. In this way, one can produce movies of the time-dependent simulation which can be visualised outside the VISSION environment as well.

The above has presented two computational applications built with the NUMLAB library in the VISSION system. However different in terms of interactivity, computational complexity, and visualisation needs, these applications illustrate well the smooth integration of numerics, user interaction, and on-line visualisation that is achieved by embedding the NUMLAB library in the VISSION environment.

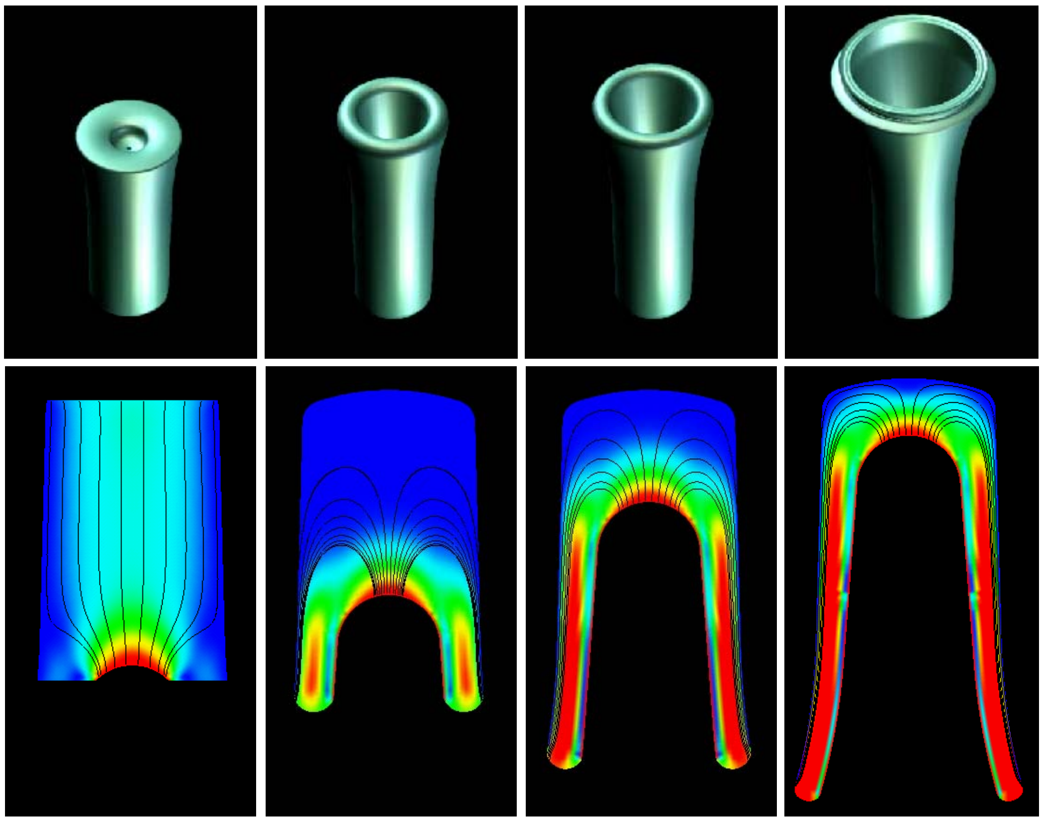


Figure 6.6: 3D visualisation of glass pressing (top row). Pressure magnitude in 2D cross-section (bottom row)

6.9 Conclusion

The numerical laboratory NUMLAB was designed to address two categories of limitations of current computational environments.

First, NUMLAB addresses the functional limitations of many computational systems by factoring out of a few fundamental mathematical notions: Vector functions $\mathbf{x}$, spaces $\mathbf{V}$, operators $\mathbf{F}$ on such spaces, and implementation of the evaluation of $\mathbf{z} = \mathbf{F}(\mathbf{x})$. Consequently, a large class of iterative solution methods, preconditioners, time steppers etc. are instances of approximate evaluations $\mathbf{x}^{(k+1)} = \mathbf{F}(\mathbf{x}^{(k)})$. This makes using and extending NUMLAB easy and close to the modelled mathematics.

Secondly, NUMLAB addresses the structural limitations of many computational environments that make them hard to extend, customise, and use for a large class of applications. The embedding of the C++ NUMLAB library in the generic dataflow environment VISSION provides interactive application construction, steering, and visualisation. These features are added to the numerical components without writing a single line of graphics user interface or dataflow synchronisation code. In VISSION, NUMLAB components can be freely intermixed with other visualisation, data processing, and data interchange components. There is a clear separation between the numerical, visualisation, user interaction and dataflow code, provided by the NUMLAB, VTK, and VISSION subsystems respectively. This makes their extension, maintenance, and understanding much easier than in systems where the above operations are amalgamated in the same (source) code. The NUMLAB computational code and VTK visualisation code can be used also outside of the VISSION environment, if the interactive application construction and steering facilities offered by VISSION are not required.

Chapter 7

Scientific Visualisation Applications

In the previous chapter we have presented the application of VISSION to the construction and use of computational applications. Visualisation of the computed data is an important component of these applications. In the previous chapter, we focused on the numerical components that enter the structure of these applications. In this chapter, we describe the construction and use of visualisation applications in VISSION . Section 7.1 states the requirements we formulate for the visualisation capabilities of VISSION. Section 7.2 presents the way these requirements are fulfilled by the integration of the VTK library in VISSION. Several limitations of VTK are removed by the integration in VISSION of the Open Inventor library, as discussed in Section 7.3. Finally, section 7.4 presents the use of VISSION in realistic rendering applications.

7.1 Introduction

A large amount of scientific visualisation software available exists, whether in the form of dedicated turnkey systems [79, 34, 6], or general purpose environments such as AVS [113], IRIS Explorer [44], or Oorange [41]. As VISSION is specifically designed for supporting generic software components from any application domain, the challenge is to provide it with general purpose visualisation capabilities similar to the aforementioned environments. We would thus like to offer to the visualisation researcher a set of visualisation primitives of a similar richness and domain coverage as, for example, the AVS system does.

For the above task, we need thus a visualisation component library that supports:

- several *dataset representations*, such as structured, unstructured, curvilinear, rectilinear, and uniform grids, with several types of values defined per node or per cell (scalar, vector, tensor, colour, etc). Support for image datasets should be provided as well. Besides these discrete datasets, the possibility of defining continuous datasets (e.g. implicit functions) should also be taken into account.

- several *dataset processing* tools, such as dataset readers and writers for various data formats, filters producing streamlines, streamribbons, isosurfaces, warp planes, slices, dataset simplifications, feature extraction, and so on. Imaging operations should also be supported, such as image filtering, Fourier transforms, image segmentation, colour processing, etc.

- several *visualisation primitives*, such as 2D and 3D rendering or objects with various shading models, mapping scalars to colours via various colourmaps, direct manipulation of the viewed objects, interactive data probing and object picking, hard copy options, animation creation, and so on.

A second requirement is that such a visualisation library should be open for extension or customisation, as researchers often need to extend, adapt, optimise, or experiment otherwise with various visualisation algorithms and data structures.

Writing such a general purpose library is clearly a task out of the scope of a single person. Moreover, such libraries exist, offering various degrees of application domain specificity and numbers of components. The simplest scenario would be thus to integrate such a library (or libraries) into VISSION. This should be an easy task, as VISSION's open architecture is especially designed for flexible, non intrusive code integration.

7.2 The Visualization Toolkit

In order to provide VISSION with the desired visualisation capabilities, we have integrated the Visualization Toolkit (shortly VTK) [94] into it. VTK is one of the most powerful freely available scientific visualisation libraries. VTK implements over 400 components for a wide range of applications, such as scalar, vector, and tensor visualisation, imaging, volume rendering, charting, and more. VTK has a classical white-box framework design, where components are implemented as C++ classes that specialise a few basic concepts such as datasets, filters, mappers, actors, viewers, and data readers and writers, as described in Chapter 2. Besides implementing the dataset and process object functionality, VTK also implements a sophisticated demand-driven dataflow execution model that supports data streamlining, caching, and multithreading transparently. Overall, the architecture of VTK follows the lines of the UML methodology (see Section 2.2), which confers it a sound, relatively easy to understand structure. VTK components can be used directly from compiled C++ programs, following the application library model described in Section 2.2.1. Most of VTK functionality for building and executing visualisation pipelines is provided also in an interpreted fashion by tcl [74] and Java [23] wrappers.

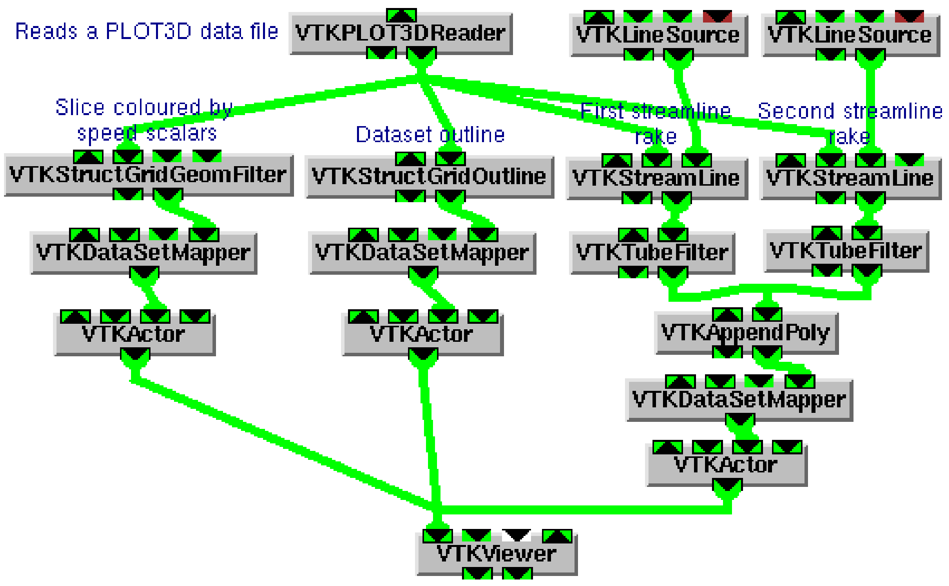


Figure 7.1: Visualisation network for the blunt fin dataset

A couple of scientific visualisation applications built with VTK in VISSION are presented next.

7.2.1 Blunt Fin Visualisation

A well known dataset for visualisation benchmarks is the blunt fin dataset. The dataset is produced by simulating the 3D air flow close to a blunt fin. For every cell in the domain discretisation, several quantities such as the air velocity, pressure, and density are computed. The data set is delivered with the VTK distribution.

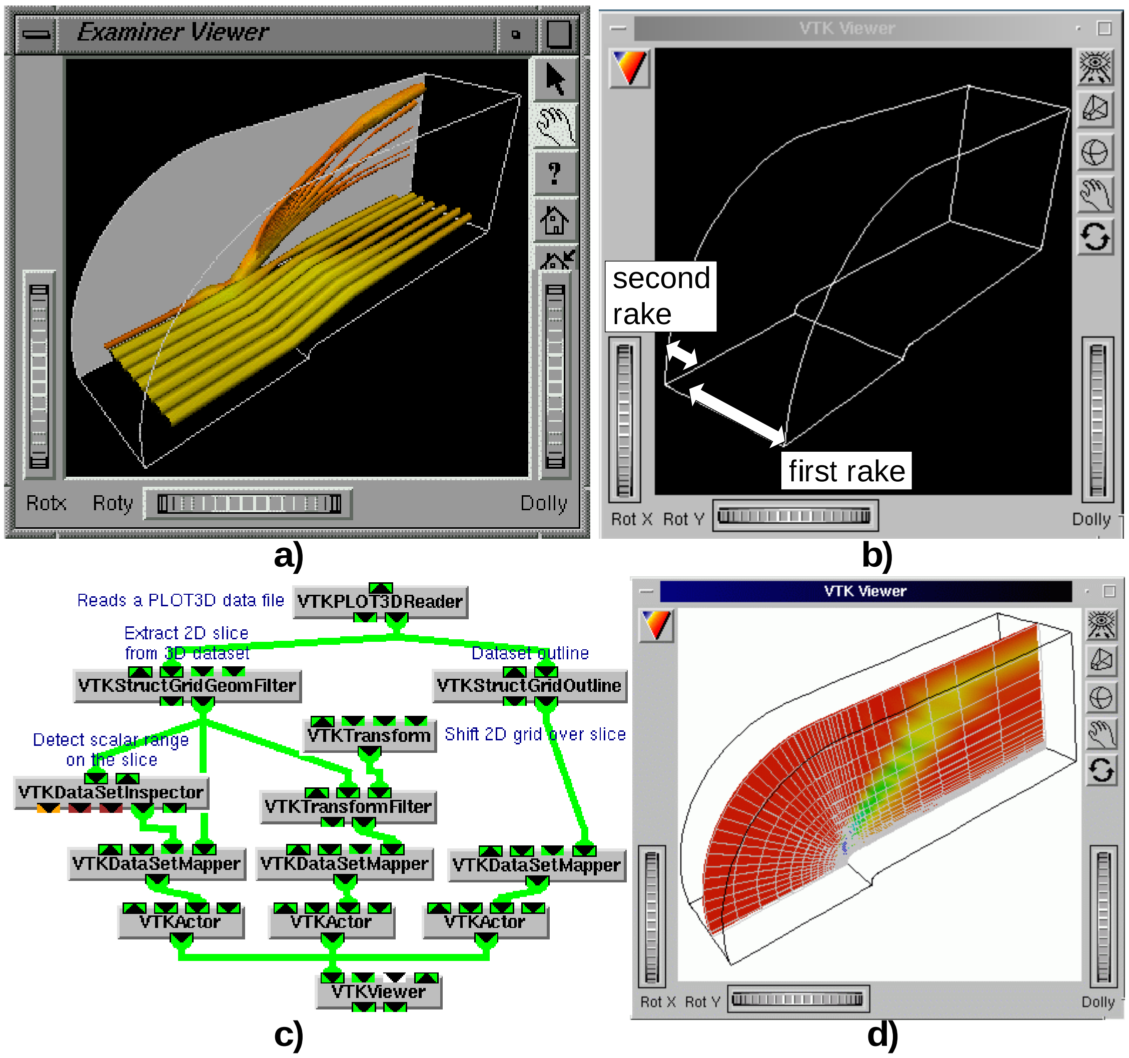
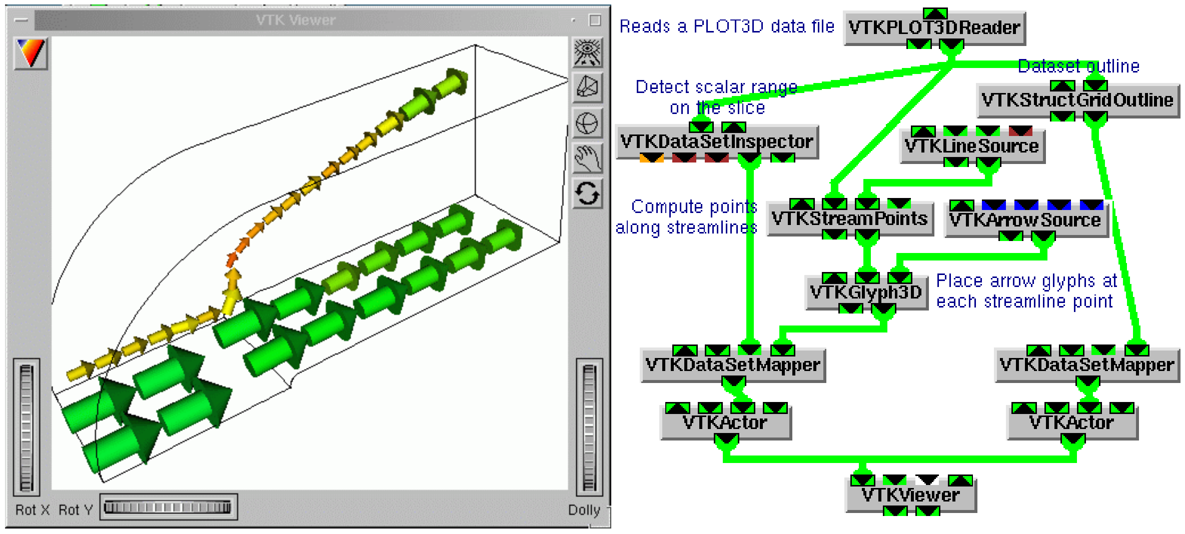


Figure 7.2: Blunt fin visualisation: a) streamlines; b) rakes; c) slice extraction network; d) slice

Figure 7.1 shows a VISSION network built for the blunt fin visualisation displayed in Fig. 7.2 a. The data set is first read from the VTK file delivered in PLOT3D format [114]. Next, two sets of stream tubes are traced from two linear rakes. A rake is a set of points from which stream lines are traced in a vector field. We use two sets of points uniformly distributed along two line segments by the `VTKLineSource` modules (Fig. 7.2 b). The first rake starts 15 streamlines equally spaced along the dataset's width (Fig. 7.2 a). These streamlines give a good overall insight in the flow. The second rake starts 8 streamlines concentrated close to the dataset's boundary depicted in gray. These streamlines show the appearance of a boundary layer flow with a different aspect than the quasi uniform flow within the domain's core. The streamlines are next transformed into tubes whose radius is controlled by the air pressure along the line. This conveys insight in two quantities, i.e. the flow velocity and pressure, in the same image.

In the second example (Fig. 7.2 c,d), a 2D slice is extracted from the middle of the dataset. The minimum and maximum of the pressure field on the slice is determined by the `VTKDataSetInspector` module and used to scale the slice's colours. The slice is displayed also as wireframe, shifted by the `VTKTransformFilter` so it is rendered in front of the coloured slice. By using a slider to control the slice position, end users can interactively slice the dataset and get an impression of the pressure variation in the 3D domain.

In the third example (Fig. 7.3), the blunt fin dataset is visualised by using arrow-shaped glyphs. First, successive points are computed by the `VTKStreamPoints` module, along three streamlines.



a) b)

Figure 7.3: Glyphs along streamlines in the blunt fin dataset

Next, the module `VTKGlyph3D` places 3D arrow geometries, defined by `VTKArrowSource`, at the computed points. The arrow glyphs are oriented and scaled according to the flow field direction, respectively velocity magnitude, and coloured to reflect the pressure scalars. The end user can control the number of initiated stream point sets and the density of the points along a stream line in the modules' GUIs. In this way, the glyph density can be easily adjusted to obtain a visually satisfying distribution. Figure 7.3 b illustrates the difference between the flow close to the vertical boundary, respectively close to the fin profile.

7.2.2 Tensor Field Visualisation

In this visualisation example, we consider a tensor field generated by the application of a point load on a semi-infinite 3D domain. The stress tensor is analytically computed and then sampled on a regular 3D grid by the `VTKPointLoad` module.

Tensor datasets can be visualised in a variety of ways. One possibility is to use the so-called *hyperstreamlines* (Fig. 7.4 a). The hyperstreamline's direction is computed by integrating through a tensor

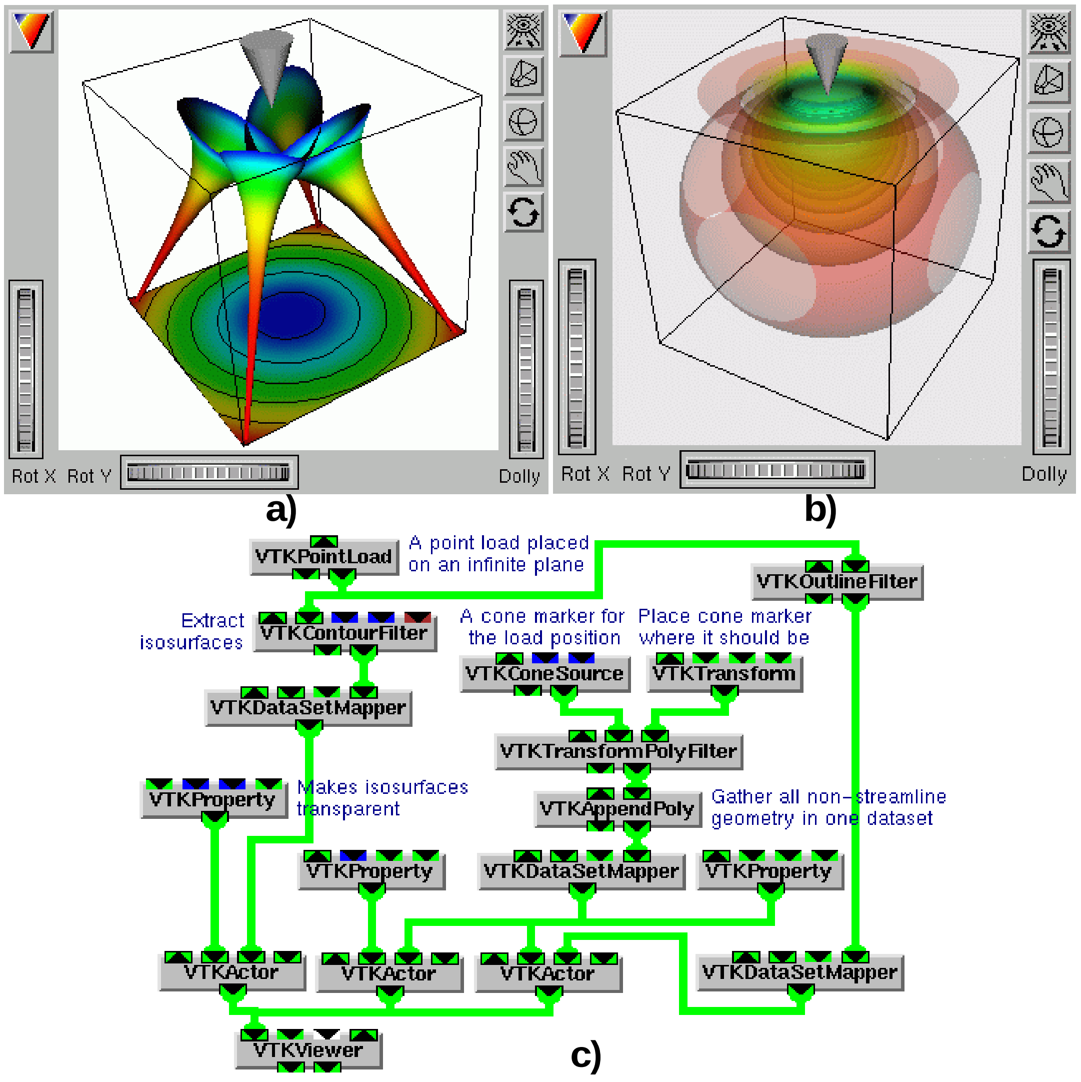


Figure 7.4: Tensor visualisation: a) hyperstreamlines; b) isosurfaces; c) network

field along the major eigenvector direction, i.e. the eigenvector that corresponds to the largest eigenvalue. The hyperstreamline cross-section is defined by the directions and sizes of the two other eigenvectors. Hyperstreamlines are thus tube-like structures with elliptic cross-sections. Because hyperstreamlines are often created near regions that contain singularities, their cross-sections can vary very rapidly over a small distance. To achieve a visually continuous rendering of the hyperstreamlines in such situations, we scale the tubes' cross-sections logarithmically.

Figure 7.4 a) visualises the point load tensor field produced with four hyperstreamlines. The position of the load is shown by a cone glyph. The four hyperstreamlines, computed by the four VTK modules with the same name, are placed such that they give a good visual coverage of the considered cubic domain. The hyperstreamlines are coloured to indicate the magnitude of the medium and minor eigenvectors. Next, a 2D horizontal slice is performed in the tensor dataset. The slice is coloured by the magnitude of the major eigenvector and displays isolines of the same quantity.

Next, the point load dataset is visualised by displaying the isosurfaces of the effective stress scalar

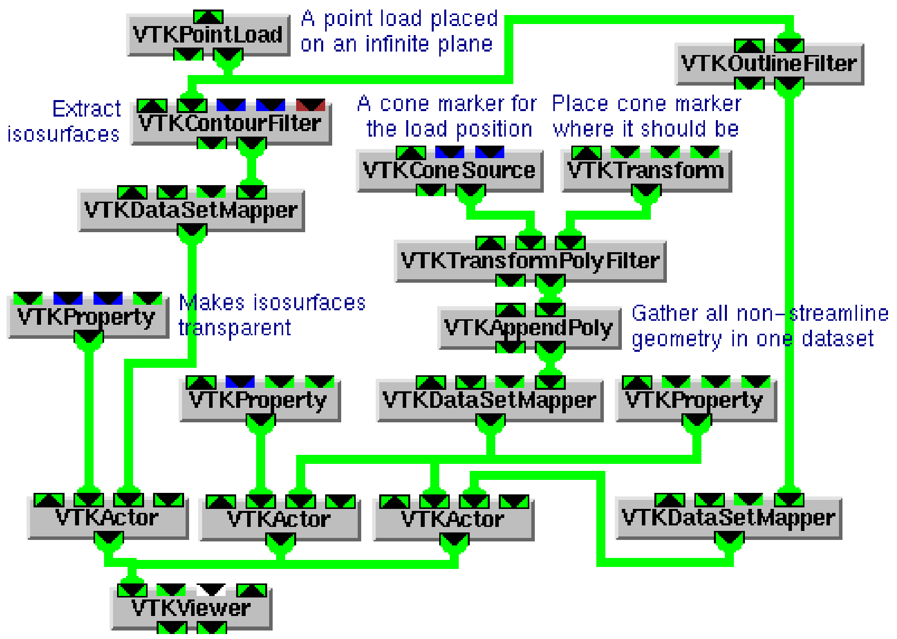


Figure 7.5: Network for point load isosurface visualisation

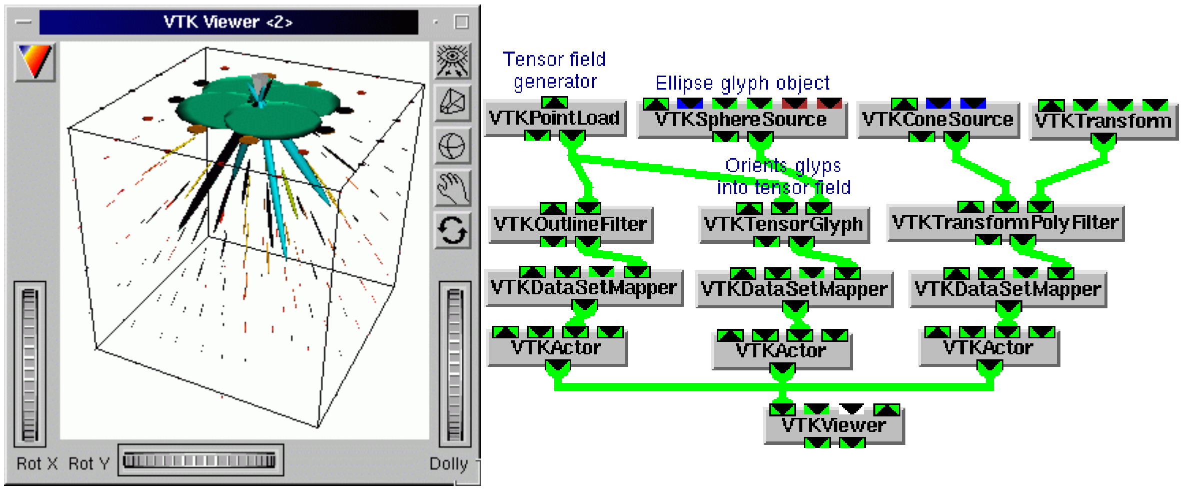
field (Fig. 7.4 b), with the network shown in Fig. 7.5. The module `VTKContourFilter` extracts several isosurfaces, coloured by their respective effective stress values. Since the isosurfaces are nested into each other, a `VTKProperty` module is used to render them half transparently.

A third way to visualise the point load tensor field is presented in (Fig. 7.6). Here, an elliptic glyph is placed in every dataset point and oriented along a local reference frame created by the major, medium, and minor eigenvectors of the tensor field at that point. The glyph is scaled along the axes of the reference frame according to the magnitudes of the eigenvectors, and coloured by the effective stress scalar value at the respective point. This type of visualisation is also frequently used for feature extraction [115].

7.2.3 Medical Visualisation

In this example, we address the visualisation of a medical computer tomography (CT) dataset of a human head. The featured dataset is the one included in the VTK distribution. The dataset is read into the visualisation network (Fig. 7.7 c) by the `VTKVolume16Reader` module from a volume file containing a set of consecutive data slices. Next, the `VTKMarchingCubes` module extracts an isosurface from the read dataset. To enhance the rendering speed, the isosurface's 3D polygonal representation is next converted into triangle strips by the `VTKStripper` module. This is needed in order to achieve near interactive visualisation of large isosurfaces on low-end graphics workstations. By using a slider for `VTKMarchingCubes`'s isosurface level input port, one can nearly interactively navigate through the skin (Fig. 7.7 a), muscular tissue, fat, and bone (Fig. 7.7 b) features of the dataset.

Next, we display a 2D sagittal slice through the dataset. The slice is extracted from the 3D volume by the `VTKExtractVOI` module. Figure 7.7 a) shows the slice's position by rendering it half transparently and by extracting its outline (`VTKFeatureEdges`) and rendering it as a tubular bevel (`VTKTubeFilter`). The same slice is extracted in Fig. 7.7 b and this time displayed by mapping the slice scalar data to colours via a contrast-enhancing colourmap. In this way, various structures related



a) **b)**

Figure 7.6: Tensor field visualisation with oriented glyphs

to specific scalar values become visible. Similarly to the isosurface case, one can interactively translate the slice through the dataset by means of a slider connected to one of `VTKExtractVOI`'s input ports.

7.2.4 VTK Limitations

However powerful, VTK is not intended as a full solution to building custom visualisations. This limitation is partly due to VTK's intent of being a programmer's toolkit rather than an application development environment, and partly to some architectural design decisions adopted in its construction. These limitations and the methods we used to overcome them are discussed below.

1. VTK provides no *end user interface* (e.g. GUIs) for its components. Application developers thus have to build such interfaces manually, using one of the GUI toolkits available on their platforms, such as Motif [33] or Windows' native GUI toolkit. One reason for the above is VTK's platform and interface policy independence. Another reason is that VTK is built upon a purely compiled C++ model, which offers no reflection facilities. Automatic building of GUIs from the components' interfaces requires however such facilities, as described in chapters 5 and 3.

2. Code development and integration in VTK is made difficult by a couple of factors. First, VTK components use a *restricted typing* for their input and output ports. Only basic types such as int, float, character string, and pointers to VTK datasets can be passed between components. Next comes VTK's restriction to single inheritance only. These limitations stem from VTK's white-box model (see Section 2.2.3) and its choice for a multiple language (compiled C++, interpreted tcl) design.

3. no *visual dataflow network editor* is provided. VTK applications are built by writing code, either in C++ compiled form, or in one of the supported interpreted scripting languages. VTK components are usually small, fine grained classes, so most VTK application networks tend to be very large, e.g. over 20 nodes. Constructing such networks by writing plain text is clearly more difficult than building them visually. Moreover, modifying a pipeline implies stopping the actual application, editing, and possibly recompiling the application script.

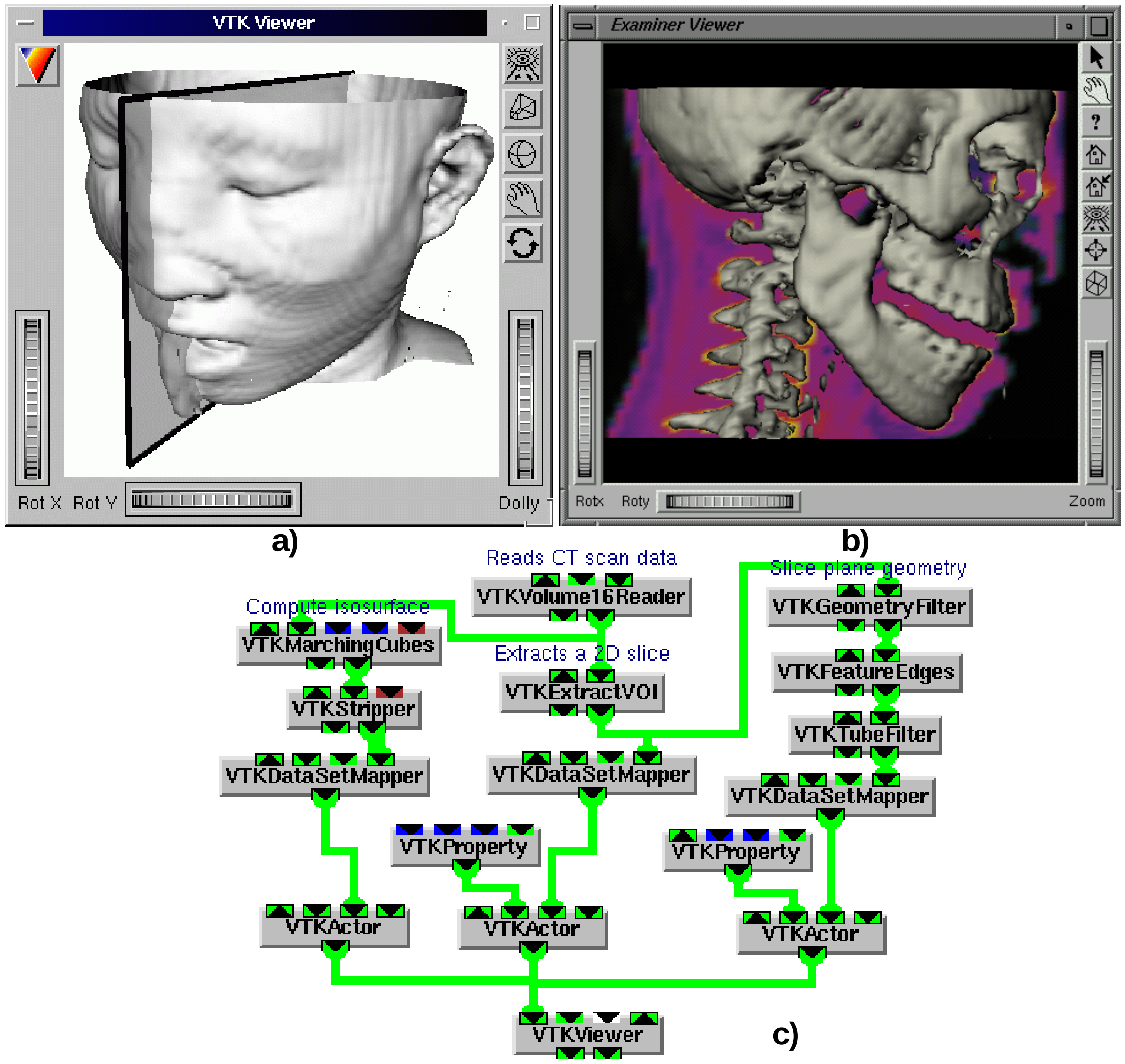


Figure 7.7: Medical visualisation

4. VTK offers only the simplest primitives for object *direct manipulation, picking, and location*. In contrast, most turnkey visualisation systems and even some application libraries such as Open Inventor [116] provide several simple to use, high end tools. Among these we note 3D viewers with integrated user interface controls, selectable viewing metaphors (fly, scene in hand, walk-through), and visual manipulators with built-in policies for object rotation, scaling, or other user defined transformations. Such tools are indispensable for most data exploration tasks, such as streamline seed placement, data slicing and probing, or computational steering.

Despite the above limitations, VTK provides an excellent basis for building a large palette of visualisation applications. The first three VTK limitations discussed above are naturally overcome by its integration in VISSION, as follows. Providing GUIs, visual component representation, and visual dataflow network editing are all operations that VISSION's design supplies by default with absolutely no effort from the part of the component developer, as shown in chapter 4. The difficulties posed to code development by VTK's restrained coding policies (no multiple inheritance and simple data typing) are overcome as well, since VISSION allows combining VTK components with other independently developed C++ components.

The next sections present how VTK's limitations concerning the direct manipulation and viewing tasks were addressed by the inclusion in VISSION of the Open Inventor component library.

7.3 Open Inventor

Open Inventor is a C++ component library for the creation of interactive 3D graphics applications. Originally Inventor was implemented atop of the IRIS 3D Graphics Library implemented on the Silicon Graphics platform. The Open Inventor follow-up is based on the OpenGL graphics library and is currently available on a range of UNIX-X Windows platforms. The main components of the Inventor library are presented below (see also Fig. 7.8).

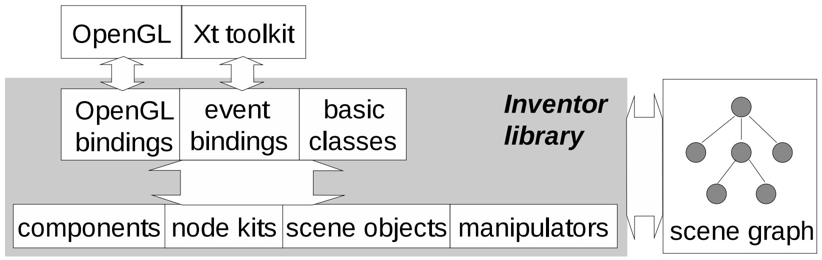


Figure 7.8: Open Inventor architecture

1. **Scene graph:** This is Inventor's representation of the modelled 3D universe. A scene graph is composed of nodes. According to their class, nodes represent various aspects of the modelled scene, such as geometries, material and shading properties, lights, level of detail, geometric transformations, camera parameters, and so on. Nodes are grouped hierarchically into *separator* nodes that model the assembly of subobjects into larger objects. For example, Figure 7.9 a)

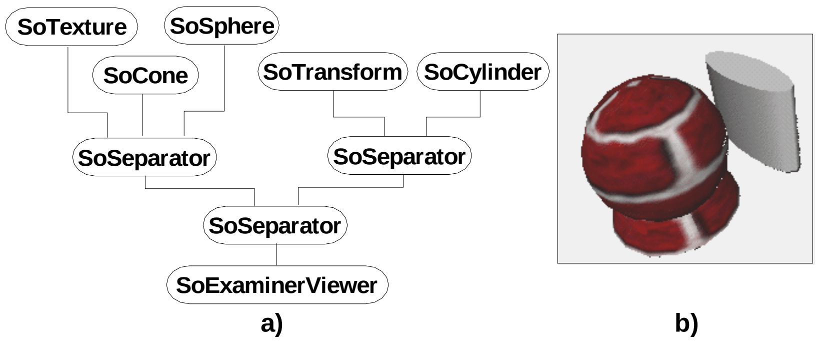


Figure 7.9: Open Inventor scene graph example (a) and its rendering (b)

shows a scene graph containing three objects: a cone `SoCone`, a sphere `SoSphere`, and a cylinder `SoCylinder`. The cone and sphere are both textured with a brick-like image by the texture node `SoTexture`. The cylinder is deformed by a `SoTransform` node. The three objects are viewed with a `SoExaminerViewer` interactive viewer tool, producing the image in Fig. 7.9 b.

2. **Scene objects:** These are the various node classes presented above. Every node has several parameters (or fields) that conceptually correspond to the input ports of a module in the dataflow model. When a node parameter is modified, Inventor updates the node and possibly propagates the change to other nodes in the graph, by traversing it from the modified node in a left-to-right, bottom-to-top order. Inventor has thus a built-in event driven dataflow mechanism.

3. **Manipulators:** Manipulators are special nodes that react to user events (dispatched by the underlying Xt library) and can be used to directly interact with the displayed objects, as outlined in Section 7.2.4.

4. **Node Kits:** Node kits are customisable subgraphs with imposed structure but with editable parameters. They are similar to VISSION's node groups (see Chapter 4) or AVS's macro modules [113].

5. **Xt Components:** Xt Components are interactive 2D and 3D viewers with a GUI that controls various viewing parameters and provides a built-in direct manipulation interface, and material and colour editor GUIs similar to the ones provided by VISSION (see Section 4.4.2).

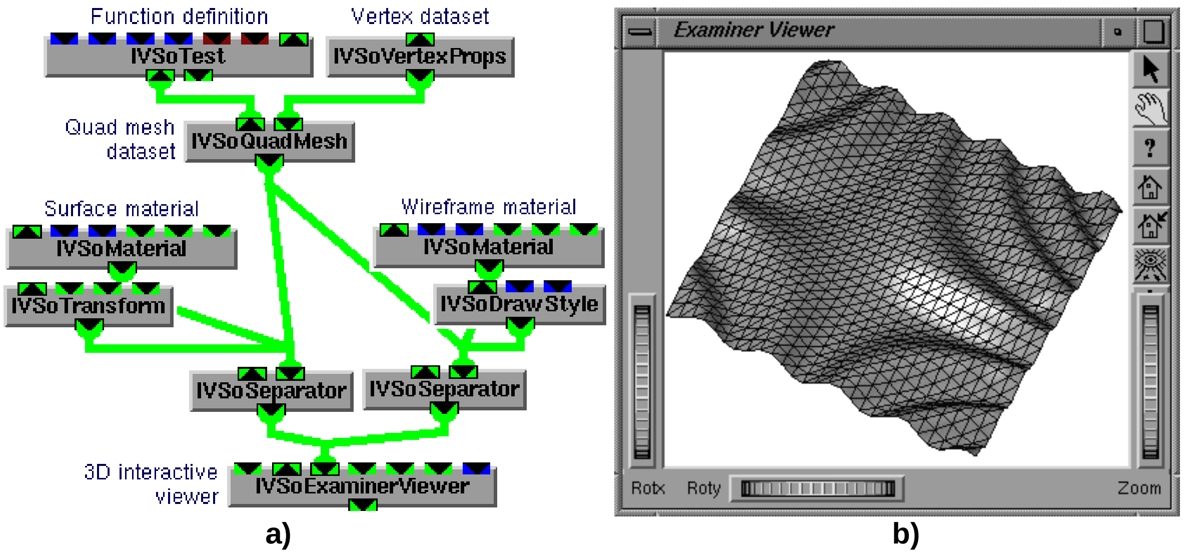
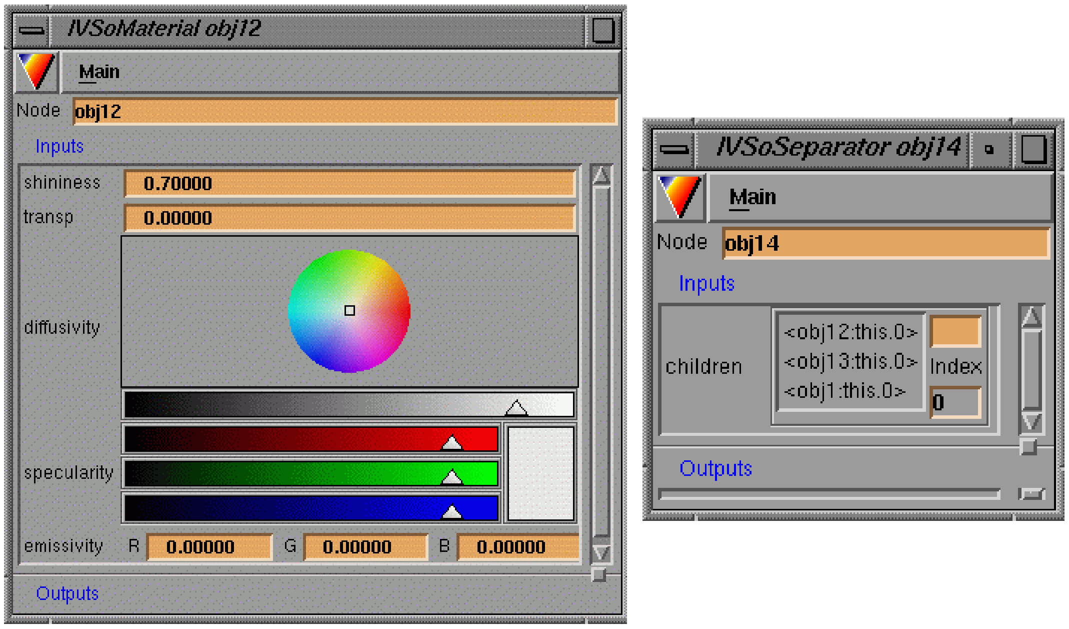


Figure 7.10: Inventor-based carpet plot visualisation network (a) and its rendering (b)

We have integrated over seventy Inventor components in VISSION by writing the required metaclasses. Similarly to VTK's integration, we didn't have to change Inventor's source code. The latter would not have been possible anyway as Inventor is distributed only in its compiled version. Several visualisations that combine VTK and Inventor functionality are presented in the following.

7.3.1 Inventor visualisations

Visualisation of scalar functions $y = f(x, y)$ is required in many application areas. An ubiquitous tool for such visualisations is the 3D carpet plot, provided by environments such as Matlab [66] and Mathematica [118]. A VISSION network for plotting such functions is shown in Fig. 7.10 a. Figure 7.10 b shows the plot of the function $f(x, y) = sin(x * y)$. The network starts with a module `IVSoTest` which provides a text field for editing the function's symbolic definition and constructs a carpet plot by sampling the function on a regular mesh of quadrilaterals. The mesh is represented by the Inventor module `IVSoQuadMesh` and visualised as a smooth shaded surface overlayed with a wireframe representation (Fig. 7.10 b). In order to do this, two separator nodes are constructed. The first one groups the mesh with the shading material properties that render the smooth shaded surface. The second separator models the wireframe rendered over the smooth shaded plot. The wireframe's black colour, drawing mode, and its position in space are specified by the separator's `IVSoMaterial`, `IVSoDrawStyle`, and `IVSoTransform` children respectively. The carpet plot is finally visualised by an interactive 3D viewer `IVSoExaminerViewer`.


IVSoMaterial obj12

Main

Node: obj12

Inputs

shininess: 0.70000

transp: 0.00000

diffusivity

specularity

emissivity R 0.00000 G 0.00000 B 0.00000

Outputs

IVSoSeparator obj14

Main

Node: obj14

Inputs

children: <obj12:this.0> <obj13:this.0> <obj1:this.0>

Index: 0

Outputs

a) **b)**

Figure 7.11: Interactors constructed for the Inventor components

The network coincides with the familiar Inventor scene graph. All its modules, except the independently developed `IVSoTest` one, are exactly the Inventor C++ components with the same names. The ports' data types are the original C++ types of the Inventor class fields. GUI interactors are automatically constructed for these ports. Figures 7.11 a) and b) show the interactors built for the `IVSoSeparator` and `IVSoMaterial` modules. The `IVSoMaterial` interactor shows several widgets, such as text type-ins, a colour wheel, and colour sliders, for editing the material's the ambient, diffuse, and specular properties. The interactor offers the same functionality as Inventor's own Xt-based `SoMaterialEditor` GUI. In contrast to Inventor's GUI, VISSION's interactor can be fully user customisable, as explained in Section 4.4. The `IVSoSeparator` interactor shows a widget for a multiple input port, i.e. a port that can be connected to several other ports. This port, of type `WRPortDynArr` (Section 3.3.4), perfectly models Inventor's multiple field type, such as the `IVSoSeparator` or the `IVSoExaminerViewer` input in the carpet plot network. The port's GUI is an editable list with the names of all ports connected to it. Using this widget, the user can e.g. group components in the desired order into a separator node. The connection order is important, as Inventor will render the components in this order when traversing the scene graph. For example, the material, draw style, and transform property nodes must be connected before the mesh node in the two separators described above in order to be used when rendering the mesh.

7.3.2 Inventor direct manipulation

We can easily add direct manipulation to the previous carpet plot visualisation by using Inventor's manipulator components. Figure 7.12 b shows two manipulators added to the carpet plot network. The `IVSoSpotLightManip` inserts a spot light in the scene graph and the possibility to control its direction, position, and angular aperture parameters by directly manipulating a 3D actor with the mouse. Directly controlling the lighting of a 3D plot is useful for emphasising the plot's 3D appearance. The second object (`IVSoTabBoxDragger`) is an Inventor dragger which comes as a wireframe box with resize handles. The box can be translated and scaled by dragging its handles with the mouse. The dragger's output can be connected to the `IVSoTest` module's input that defines the function's domain, as the dotted line in Fig. 7.12 a shows. When the dragger is interactively activated, the sizes of the new domain and the carpet plot's vertical scaling factor are passed to the `IVSoTest` module. The function plot, sampled on the new domain, is redisplayed in the viewer. The whole process takes place in real time, so the user has the feeling he actually drags a rectangular 'data lens' over the function's 2D definition domain.

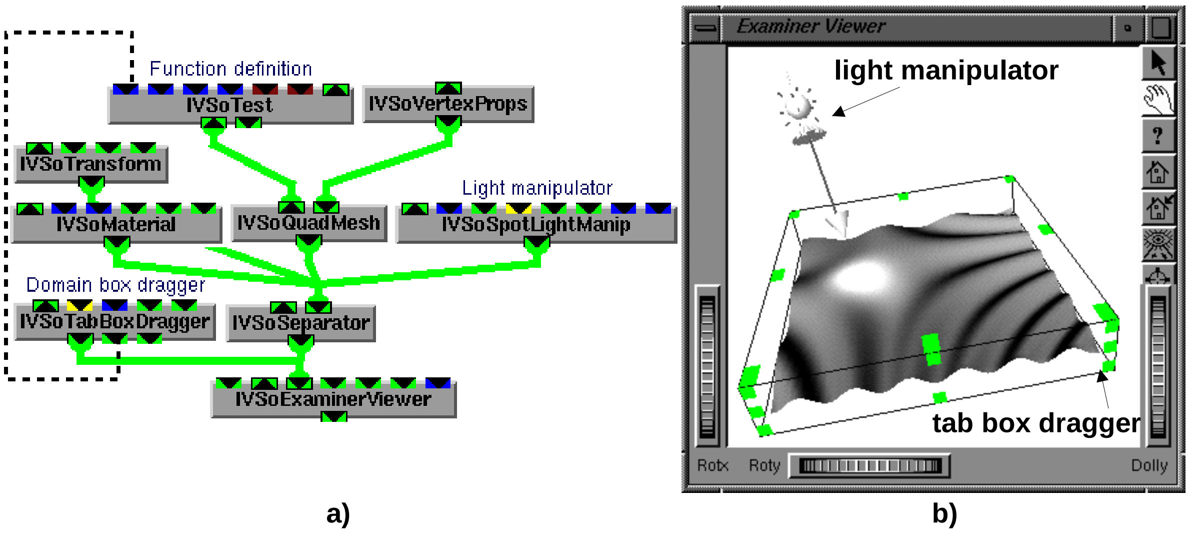


Figure 7.12: Direct manipulation with Inventor components

Direct manipulation is convenient in the cases when the meaning of the controlled parameters is visually coupled with other displayed entities. Inventor's 25 manipulators and draggers offer a large selection of direct interaction tools with 1D, 2D, and 3D rotation and translation degrees of freedom such as trackballs, linear, circular, and planar sliders, general affine transformation boxes, and so on. By incorporating these tools, VISSION offers the possibility to directly steer simulation pipelines in a variety of ways. For precise parameter control, users can revert to the classical GUI interactors.

7.3.3 VTK and Inventor Combination

We have presented in the previous sections how Inventor and VTK components can be combined in various visualisations in VISSION. The two toolkits exhibit a certain resemblance, especially in their dataflow and object-oriented architecture, as well as in the 3D rendering components provided. However, they are designed with different application domains in mind. Inventor's goal is to provide general-purpose 3D scene modelling and animation. VTK focuses mostly on support for scientific visualisation applications. The main differences between the two toolkits are summarised as follows.

- Inventor's data model covers the description of a 3D scene. The provided elements are geometric objects, transforms, and rendering and viewing properties. VTK's data model covers more data types common in the scientific visualisation community. It consists of datasets and process objects, which represent respectively various data types and data processing operations encountered in scientific visualisation.

- Inventor offers high-level, specialised components for 3D viewing and direct manipulation, as well as some GUI components. In contrast, VTK provides only some basic primitives such as 3D cameras and simple point-on-surface pick functions, and no GUI components.

- Inventor has a special dataflow network traversal policy in which several actions that share a global state are applied over the whole scene graph. Unconnected nodes can still influence each other by modifying the global state during the traversal, such as material or transform nodes which affect all nodes encountered after them during a traversal. The overall Inventor network traversal mechanism is quite complicated. Several action types can be applied during a traversal, and the encountered nodes can respond in several ways to these actions [116]. In contrast, VTK has a simpler dataflow mechanism in which no global state is maintained. Nodes communicate only explicitly via data connections. There is a single traversal action, which consists in calling the components' update operation in a demand-driven fashion.

- Inventor's rendering mechanism is highly tuned for speed by using several internal scene graph transformations and caches. In contrast, VTK's rendering components directly call the underlying graphics library, e.g. OpenGL. VTK has a lower overall rendering pipeline throughput, especially for large scenes.

- Inventor is commercial software which has to be separately purchased for every new platform. In contrast, VTK is freely available on a wide range of systems.

VISSION integrates both Inventor and VTK, so application designers may freely decide whether to use Inventor or VTK components for a specific task. On high-end graphics workstations that have an Inventor installation, one could use the fast Inventor components for the 3D rendering and direct manipulation, instead of the similar VTK components. On machines where Inventor is not available, one would use VTK to construct slower but functionally similar pipelines.

We shall illustrate the above by a visualisation provided in the VTK standard example set. In this example, a scalar quadric function defined on a 3D domain is visualised by means of several isosurfaces. The network (Fig. 7.13 a) starts with a module `VTKQuadric` which defines a quadric function of the form $f(x, y, z) = ax^2 + by^2 + cz^2 + dxy + eyz + fxz + g$, where the coefficients $a - g$ are input by the user. Next, `VTKSampleFunction` samples the function onto a regular grid. Isosurfaces are extracted from the sampled dataset by using the marching cubes algorithm [60]. The extracted isosurfaces are next input in two different rendering pipelines. The left pipeline uses Inventor components and end with the already described `IVSoExaminerViewer`. (Fig. 7.14 a). The right pipeline uses only VTK components and ends with an interactive 3D viewer `VTKViewer` (Fig. 7.14 b) which we have built by enhancing the basic VTK viewer with GUI controls similar to the ones provided by the Inventor viewer. As seen in Figs. 7.13 a and 7.14, the VTK and Inventor rendering pipelines and viewers are almost identical in structure and functionality. End users and application designers can thus construct and utilise visualisation applications with the rendering pipeline which best suits their current platform or availability constraints.

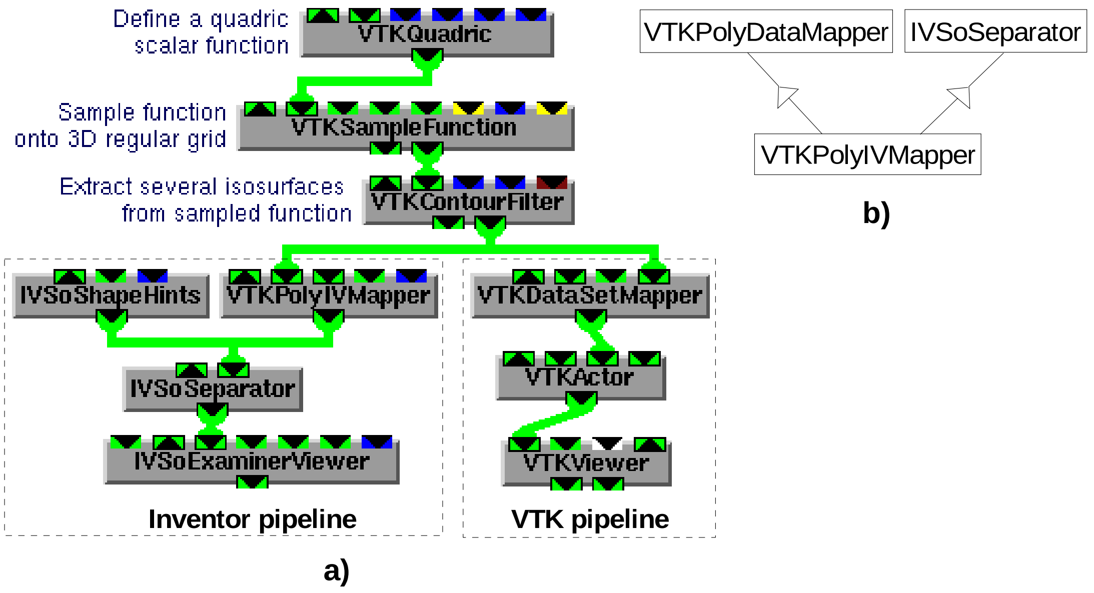


Figure 7.13: Visualisation combining Inventor and VTK. Network (a) and adapter class (b)

In order to combine the VTK and Inventor pipelines, we had to write a simple data adapter class `VTKPolyIVMapper` of around 120 C++ lines which translates VTK's geometric dataset into the equivalent Inventor representation. The design of this adapter is a good example of situation in which multiple inheritance is needed (Fig. 7.13 b). On one hand, the adapter inherits from the Inventor `IVSoSeparator` component, which makes it connectable to the Inventor scene graph, thus viewable. On the other hand, the adapter inherits from the VTK components `VTKPolyDataMapper`, which makes it take part to the VTK dataflow mechanism by reading a VTK data set. In order to merge the two toolkits, `VTKPolyIVMapper` defines its own update routine which translates the VTK polygonal data set delivered by its `VTKPolyDataMapper` parent, translates it to an Inventor representation, and inserts this representation into its `IVSoSeparator` parent. As pointed out by Gamma et al. [37], class adapters are the mechanism of choice for connecting independently developed class hierarchies, such as the VTK and Inventor toolkits. Constructing such an adapter would however not have been possible without multiple inheritance , if we do not wish to modify any of the two involved toolkits. In our case, VISSION natively supports multiple inheritance, so the coupling of the VTK and Inventor toolkits was done without any problems.

Combining VTK and Inventor components in VISSION is not limited to having alternative rendering back-ends. One can for example steer the input of a VTK computational pipeline with the output of a manipulator acting on its Inventor rendering back-end, similarly to the network depicted in Fig. 7.12 a. The VTK and Inventor components may have their own data communication mechanisms and update policies among themselves and still perfectly cooperate and be freely intermixed in a VISSION network. For example, in the visualisation described above the Inventor components use their own built-in event-driven update mechanism and special scene graph traversal, while the VTK ones use VTK's own demand-driven dataflow policy. The two mechanisms are merged transparently as both VTK and Inventor components interface with VISSION via the same metaclass layer described in Chapter 3. Combining Inventor and VTK components would not have been possible without the presence of the meta-layer abstraction (e.g. directly in compiled C++) as no common component dataflow interface would have been available.

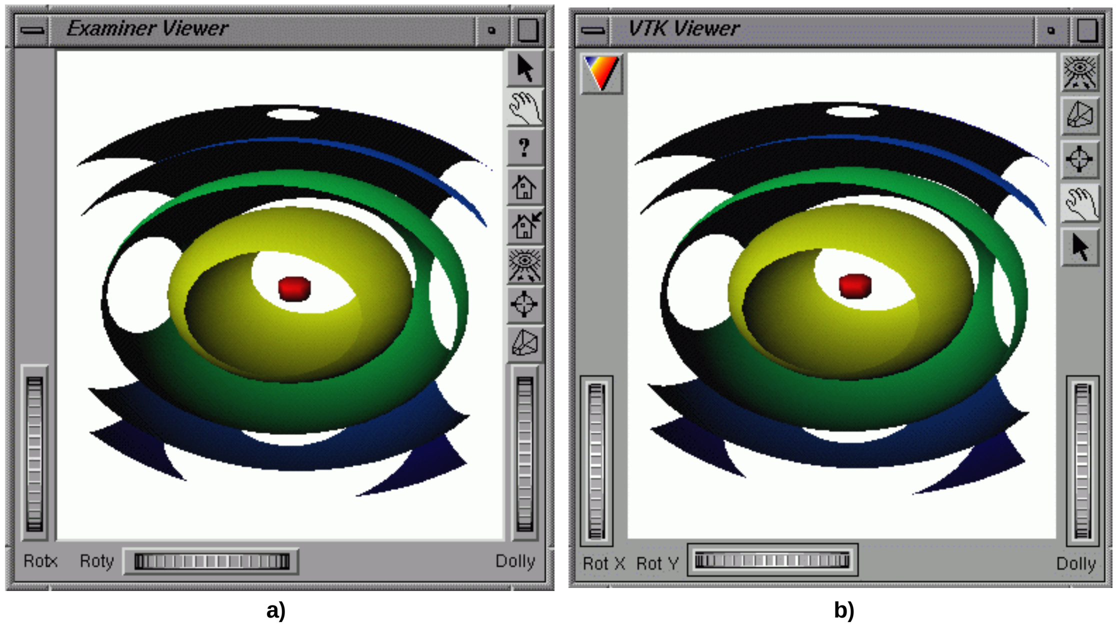


Figure 7.14: Inventor (a) and VTK (b) 3D interactive viewers

7.4 Realistic rendering

Radiosity techniques [19, 98] are an important class of realistic image synthesis methods, due to their ability to simulate indirect lighting and visually convincing shadows.

Radiosity simulation software often requires delicate tuning of many input parameters, and thus can not be used as black box, monolithic application pipelines. Testing new algorithms requires also the configurability of the radiosity pipeline. These options are however rarely available to non-programming experts in current radiosity software. In contrast, software for other realistic rendering techniques, such as raytracing, is much simpler to use and requires less parameter control by the end users. Consequently radiosity applications, even though capable of producing high quality renderings, are mostly employed only by specialists.

We addressed this problem by including into VISSION a radiosity component library written in C/C++ by us [108] before VISSION was conceived. Similarly to the other integration cases we described, making the radiosity toolkit available in VISSION required the writing of the metaclasses and the creation of a few dataset adaptors.

Application designers can now easily build various radiosity renderer setups by visually assembling the provided radiosity components in VISSION's network editor. Figure 7.15 shows a radiosity pipeline built on the progressive refinement model [20]. The pipeline starts with a 3D scene reader that creates the scene dataset, followed by an initial *a priori* scene subdivision into coarse elements, or patches. Next an octree spatial partitioning is built from the subdivided scene. The algorithm core consists of the iterative radiosity shooter based on raytracing which produces a progressively refined rendering of the scene. During the progressive refinement, adaptive subdivision of the initial scene mesh is performed based on various solution characteristics. The method maintains thus a two-level hierarchy of non-uniform element meshes, and is thus similar with other hierarchical numerical methods known in the literature. The form factors governing the energy transfer are computed on the fly, by adaptively sampling the source and target elements. Testing for spatial occlusion during the energy shooting is done by raytracing, using the computed octree to speed up the ray-polygon intersections. The 3D scene illumination solution is next visualised by being input into a VTK visualisation pipeline (Fig. 7.16 a,b).

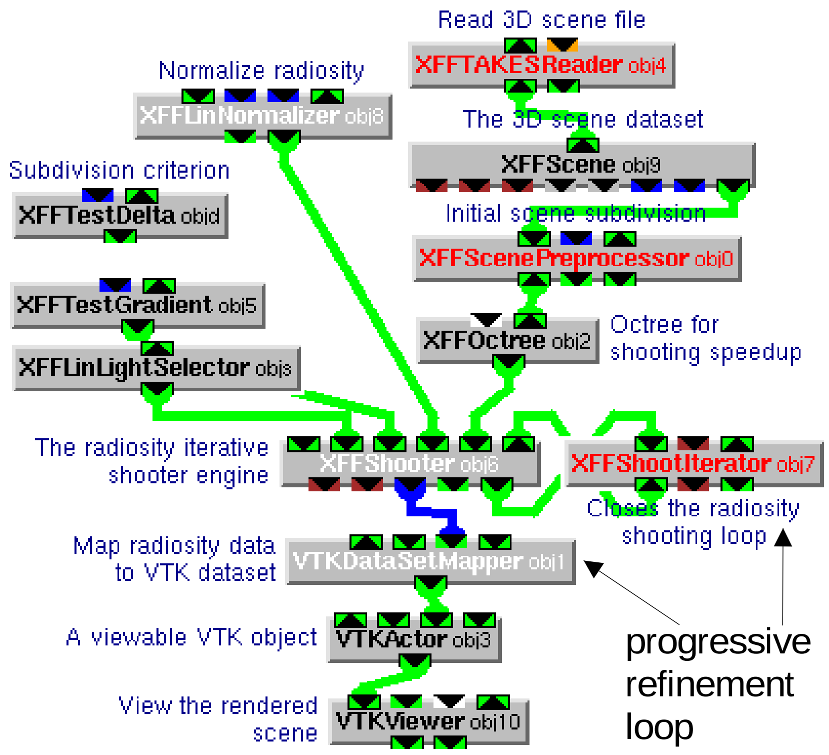


Figure 7.15: Radiosity pipeline in VISSION

Using the dataflow application model for radiosity rendering has several advantages as compared to a classical monolithic application setup. End users can now change all the 'hidden' parameters along the radiosity pipeline, such as refinement thresholds, maximum number of iterations, or normalisation factors, and notice the rendering improvements almost instantly. This allows to monitor and time the progressive improvement of a radiosity solution online, as the loop in Fig. 7.15 keeps iterating and producing new solutions. Application designers can change a pipeline component even during its execution. This allows one to render a scene using a certain refinement algorithm up for e.g. 50 iterations and then interactively switch to another algorithm by connecting its icon to the pipeline. For example, Fig. 7.15 shows two possible mesh subdivision criteria `XFFTestDelta` and `XFFTestGradient` that decide an element's subdivision based on the solution's first or second derivative respectively. Component developers can write new radiosity algorithms or integrate existing ones by subclassing the existing radiosity toolkit's C++ base classes. Finally, results are available for interactive visualisations or further postprocessing using the VTK components. For example, VTK's 2D imaging components were used to interactively process 2D snapshots produced by a running radiosity pipeline in order to monitor the solution quality and detect illumination anomalies. Several problems of the original ra-

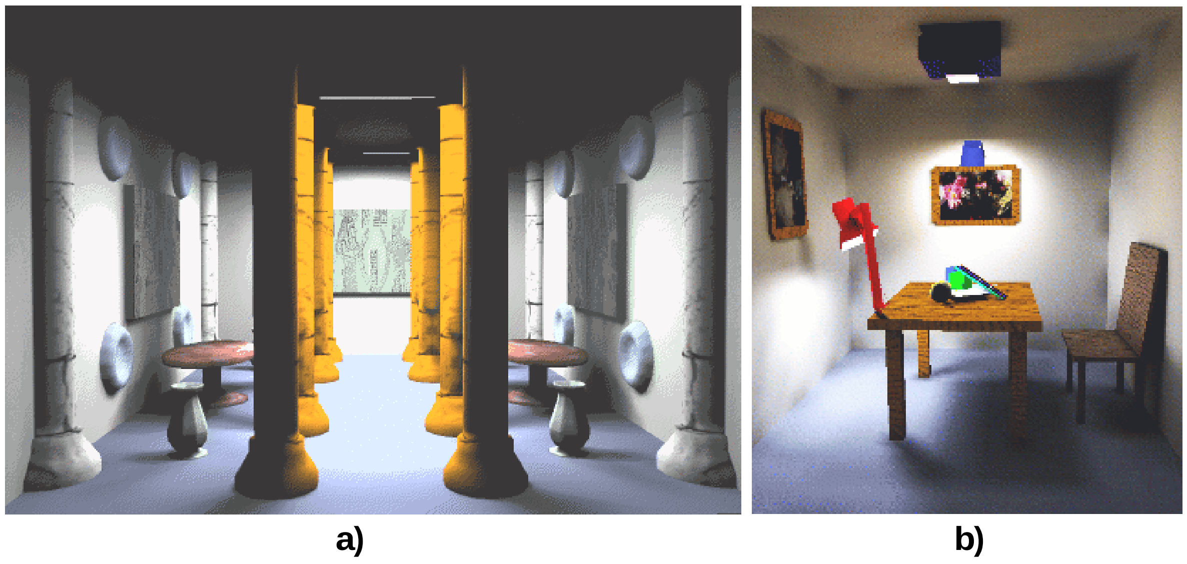


Figure 7.16: Radiosity renderings: a) pillar room (60,000 elements, 50 iterations). b) office room (13000 elements, 60 iterations)

diosity code left undiscovered in the non-interactive, batch mode setup, were quickly detected in the VISSION based setup.

7.5 Discussion

In this and the previous chapter we have presented several application areas in which VISSION was successfully used. In the following several observations on the flexibility and limitations of VISSION are presented which address VISSION's suitability as a framework for domain specific application construction and steering.

One of the most challenging problems SimVis researchers and software designers are confronted with is the choice of the software platform or environment to suit a given application. In most cases, a perfectly suited application does not exist, so one has to adapt an existing one or build a new one. In research environments, coding and adapting new components and gluing existing ones together to produce several application setups is a routine task. It is, however, also this task that is the most tiresome and error prone part of the research and development process. This is mainly due to the heterogeneity of existing software components that range from monolithic applications to frameworks and libraries, ending up with small, self developed algorithms. A frequent question SimVis software practitioners asks themselves is: which software environment should I adopt such that I can easily combine my own code with other people's code, and with the powerful available large SimVis libraries?

In this respect, VISSION's non intrusive code integration policy proved to be a powerful concept. Code components which were developed on a totally independent basis, ranging from two very large C++ libraries (VTK and Inventor) containing over 500 components each, to small libraries (under 20 components) and one-component code fragments independently developed by several researchers could be integrated and coupled together. The only programming effort involved writing the corresponding metaclasses having a size of approximately 6 MC++ lines per metaclass. This process could be done gradually. First, only the base classes of VTK and a few concrete filters, mappers, and a 3D viewer were integrated to test the concept. Next, the remaining VTK classes were integrated. No modification to the original VTK code was done. In a few cases when we detected an erroneous or otherwise undesired behaviour of the integrated VTK classes, we provided some simple C++ wrappers around the original VTK code. These wrappers maintain the same interfaces and delegate their operations to the underlying VTK implementation, but apply the desired checks or adaptation to the original operations. As the original VTK code was not modified, we could use it as it evolved through several releases with practically no change to our metaclasses. Only a few minor interface changes were needed when the new VTK releases changed some component signatures.

other minor changes were sometimes needed, e.g. when 'wrapping' an algorithm present as a fully independent, console-based C program, into a C++ class declaration that communicates with the outside world via method calls instead of the standard input and output. In most cases this actually made the original code even more reusable and context independent. Each component set preserved its own typing policies, as there is no need to e.g. force all components to use the same dataset structures. The same holds for execution mechanisms, as, for example, VTK uses internally a demand driven dataflow, while Inventor has an event driven one. Components from the two libraries can, however, be freely intermixed in VISSION pipelines.

The above facts lead to the following strategy for a typical VISSION user:

1. identify the application domain of interest (e.g., finite difference computations)

2. find an existing software library that addresses the respective domain. If such a library exist, bring it to the form of a C++ class library. In virtually all cases, this should be relatively easy, as C++ offers a clearly more powerful set of concepts and interface mechanisms than other languages such as C or FORTRAN.

3. test the library offline, e.g. in a classical console-based program.

4. write the meta-library for the existing C++ library. This is trivial in most cases, as MC++ was especially designed to cope with C++ interfaced components.

5. use the library in VISSION, possibly by intermixing its components with other components already present in VISSION. In this way, one has to provide only the components particular to his application domain, as many general-purpose visualisation and graphics primitives are already present in VISSION via the VTK and Inventor libraries.

The above scenario was applied successfully to building over 50 visualisations and graphics applications by integrating the VTK and Inventor libraries in VISSION. Several other more specific application domains such as numerical computations and realistic rendering were addressed in basically the same way. In most cases, applications used components from two or three different libraries, which proves that component reuse and interoperability takes indeed place in VISSION. The fact that several libraries with over 600 components were made available in VISSION, either by adapting existing code or by writing it from scratch proves that the system scales well. VISSION's combination of visual dataflow programming, supported by a flexible code integration mechanism proves to be an effective alternative for SimVis researchers to the classical paradigm of writing a standalone program doing computations, interaction, and visualisation for every new application.

One has to admit that classical programming is undoubtedly more flexible than visual dataflow programming. If one has total freedom over the way code statements are written, it is obvious that more control can be achieved over the shape of the final application than when combining a few tenths of blocks controlled by a few parameters and a fixed dataflow execution model. However, one should not forget which are the prices to pay for this freedom:

- Writing, debugging, understanding, and maintaining classical code is orders of magnitude more difficult than doing the same to a visual network. Building a visual pipeline involves *no* programming knowledge, while building a similar application in a classical way involves familiarity with a compiler and a programming language, component libraries, GUI libraries, windowing systems, and possibly more. The time it takes to do the same operations in a visual environment is much shorter, even for e.g. expert C++ programmers.

- Sharing data between peer researchers has taken place since long. Sharing applications and code is however rarely done, as people have different coding styles. Visual dataflow programming and the distribution of code as VISSION meta-libraries makes the interchange of code modules or full applications much easier, as these now obey a standard form.

Chapter 8

Conclusions

8.1 Introduction

This thesis addresses the development of interactive scientific simulation and visualisation (SimVis) applications in academic research environments. Such environments pose major challenges to the development and use of SimVis software applications. Research applications usually have an experimental nature. They are built to test new concepts, algorithms, data structures, and so on. The nature of the research process implies a continuously changing structure of the SimVis software applications. Insight or new ideas acquired during the (interactive) experimentation with the application often urge the need to add or remove numerical or visualisation software components, recode components, or even write new components from scratch.

8.2 SimVis Application Construction and Use

Visual programming environments are one of the most successful solutions to the development of interactive SimVis applications in the research community. Such environments provide an architecture that supports rapid and intuitive application construction and interactive steering in the form of visual dataflow networks. The user can focus on steering the final application or on the development of custom numerical or visualisation code.

However flexible, most existing SimVis visual programming environments have important limitations, especially with respect to the development of new software components or the integration of independently developed components. In particular, we focus on object-oriented (OO) software techniques, which offer well established tools in the software engineering community for large-scale code organisation and reuse. Both object orientation and visual dataflow programming are new techniques. Consequently, the combination of their advantages in the field of SimVis software design has been addressed only partially.

The aim of this thesis is to provide a SimVis system that offers a close combination of the above two techniques. We approach this by first designing an abstract architecture that combines the dataflow and object-oriented modelling principles. This architecture extends the object-oriented constructions of the C++ language, such as typing, inheritance, and encapsulation, with the basic elements needed by dataflow modelling, namely input-output ports and the update operation. The combination of the two results in the MC++ component specification language, described in chapter 3.

MC++ provides a non-intrusive manner for extending C++ class libraries with the dataflow elements that make them eligible for use in dataflow network construction. We use a black-box, non intrusive code integration approach, in contrast to existing solutions for dataflow component design. Rather than modifying the semantics of the C++ language by adding new constructions to its syntax or designing a new language from scratch, as done by various SimVis systems, we defined MC++ as a 'shell' language, or meta-language. This implies two things. First, the MC++ specification files, or meta-files, are separately analysed (e.g. parsed and interpreted) from the C++ files they extend. This makes the MC++ language implementation simple and easy to extend. Secondly, the original C++ code stays completely unchanged, which keeps it directly usable in other contexts, e.g. outside our SimVis environment.

The second step of our work takes the OO-dataflow architecture and the MC++ language to a concrete implementation, the VISSION simulation and visualisation environment. Due to the syntactic and semantic separation of concerns of C++ and MC++ described above, the implementation of VISSION was relatively simple, as compared to other similar SimVis environments. The key element of the implementation was the use of a C++ interpreter, or virtual machine, responsible for the dynamic C++ component loading and code execution. The concern separation mentioned above enabled us to completely reuse an existing C++ virtual machine coming as about 80,000 lines of C/C++ source code. In contrast, the MC++ virtual machine we implemented ourselves is not larger than 20,000 C++ lines. The total of 100,000 lines of code is to be compared with similar SimVis system kernels of over 400,000 lines [113, 44, 118]. The design of VISSION is the object of chapter 5.

Similarly to existing visual SimVis application builders, VISSION also offers a visual programming interface for application construction and component interfaces for interactive monitoring and steering of applications. The architecture of VISSION, based on the C++ and MC+ virtual machines, enables the automatic construction of both component iconic representations and component user interfaces out of the component specification written in MC++. Practically, this means that all SimVis user interface related aspects are provided in VISSION with no programming effort. The construction and use of VISSION's visual interfaces is described in chapter 4.

Our intention was not only to provide a better theoretical architectural framework for SimVis environments and an eventual prototype implementation. The success of any software environment is ultimately measured by its effectivity and efficiency when facing its users. Consequently, we have extensively used VISSION for the whole pipeline of SimVis component development, and application design and use, both for numerical simulation, as well as for visualisation applications. In the area of numerical simulation, we used VISSION for the computational steering of an electro-chemical drilling (ECD) numerical simulation, as described in Section 6.2. The ECD problem illustrates well the use of VISSION for producing turn-key applications, in which end users can intuitively control and monitor a process via a simple interface.

A more extensive numerical application design and integration in VISSION is represented by the NUMLAB component library. NUMLAB offers a comprehensive set of C++ components for modelling a large class of numerical applications, such as ordinary and partial differential equations discretised and solved with various methods, on 2D and 3D domains. NUMLAB was developed as an independent C++ library. Its integration in VISSION and coupling with various visualisation components demonstrates well the claim of easy application-independent component integration. Moreover, the object-oriented design of the NUMLAB library enables the modelling of a large class of numerical problem with a small set of software components. A major aim in the design of NUMLAB was orthogonality, which allows the application designer to easily interchange one component type with another one of a similar type, such as the interchange of numerical solvers, preconditioners, or mesh generators. As all these actions are done via VISSION's visual interface, numerical experimentation is greatly simplified as compared with other similar software applications. NUMLAB is presented in section 6.3.

The second application class for VISSION concerns scientific visualisation. The integration in VIS-

SION of the over 500 components of the VTK visualisation library demonstrates again the flexibility of the proposed architecture. Several tens of visualisation applications were built using VTK components in VISSION, such as scalar, flow, and image data visualisation. These are described in Section 7.2.

Finally, we look at another application domain, namely realistic rendering. Similarly to the ECD application, we have integrated a C++ library for radiosity computations in VISSION. Using this library, one can easily set up a 3D scene, initiate its rendering, and experiment interactively with the various parameters of the radiosity pipeline. Compared to the classical file input-output radiosity applications, this allows a more intuitive understanding and control of the convergence issues involved in the radiosity process. This application is the object of Section 7.4.

8.3 Directions for Future Work

Interactive construction and use of SimVis applications is a growing field. New application domains are continuously coming into the focus of the researchers. New methods are devised to address existing computational problems. Finally, more efficient or more generic software implementations appear for existing application domains. The work presented in this thesis on SimVis application construction can evolve in several directions, as follows.

First, VISSION can be used to provide interactive application construction, visualisation, and steering facilities for new application domains. For this, the domain-related functionality has to be provided as a set of components, which are next to be integrated in VISSION by writing the corresponding MC++ specifications. The integration mechanism has been extensively put to test for the large-scale examples described in the previous section. We therefore expect that covering new application domains in VISSION will be relatively easy. Object orientation is a well proven methodology for modelling a large range of problem domains. We expect that VISSION will scale well with the number and diversity of application domains. In particular, the ongoing development of the NUMLAB numerical component library makes VISSION a very interesting environment for numerical computation and experimentation. Ongoing work in scientific visualisation done in VISSION is described in [111, 38].

Secondly, there are promising research directions in extending the mechanisms of VISSION's kernel itself. These directions are presented next from the perspective of the three user roles discussed in this thesis.

From the *end user*'s perspective, we could imagine the development of new application steering and monitoring mechanisms. Such mechanisms would be more convenient and natural for specific SimVis domains than the generic graphics user interfaces described in chapter 5. For example, multimodal (e.g. voice or gesture based) control of a simulation could augment a standard graphics user interface. A concrete scenario for such interfaces is a 3D data investigation application, in which the user visualises the output of some simulation in 3D (e.g. isosurfaces) and in the same time controls some simulation input parameters. Examining a 3D dataset usually involves the direct manipulation of the visualised object in a graphics window by means of a mouse. Steering the input parameters involves manipulating some widgets in another window. Performing both operations requires the user to switch his attention from one window to the other, which is a disruptive process. By introducing a voice or gesture based input in VISSION, the input steering and output examination actions could be performed concurrently. For example, one could assign the right hand to the 3D mouse-based manipulation and the voice or the left hand's gestures to the input parameter steering. The implementation of such alternative input devices in VISSION would be greatly facilitated by the modular structure of the input devices, described in chapter 5.

Another aspect pertaining to the end user role concerns the construction of the end user graphics interface. Many SimVis systems offer sophisticated tools to design such interfaces visually. So far, VISSION offers the automatic construction of component interactors and a programmatic way to construct custom end user interfaces, both presented in chapter 4. Replacing the programmatic user interface construction by a visual GUI editor would make the construction of custom user interfaces a much easier task, especially for non programmers. Due to the modular design of VISSION's views, this task should also be relatively easy to accomplish.

Finally, integrating an interactive documentation system within VISSION would greatly help the end user and application designer's roles. SimVis systems such as AVS provide such a facility in terms of per-module browsable hypertext pages. Work to provide a similar facility in VISSION in terms of automatically generated HTML documentation from the MC++ and C++ component information is under progress.

From the *application designer*'s perspective, new techniques could be imagined to make the dataflow network construction easier. An important limitation of the scalability of the dataflow model is the large number of components from which the designer has to choose the right ones for the problem at hand. Documentation systems are of limited usability, since they offer general information, not related to the specific context of the network under construction. One technique for assisting the network construction on which we have worked is a smart agent that learns the application designer's preferences [109]. The agent assists the construction of a new network by providing the application designer with hints for connecting modules based on information learnt from previous network constructions. This concept could be extended with new heuristics or learning strategies to further simplify the task of building dataflow networks. The development of visual techniques for managing large component libraries is a promising research direction, related to the emerging information visualisation discipline. One such technique would be to replace the 'flat' presentation of component libraries as a scrollable list of components by a two-dimensional graph of components, where similar or logically related components would be placed close to each other. Browsing such a graph would provide a faster way to understand the organisation of a large component library that the actual list-based interfaces.

Finally, VISSION could be extended from the *component developer*'s perspective as well. An important direction would be to extend VISSION's component implementation language to other languages than C++, such as Java or Fortran. This would allow a direct integration of application libraries written in these languages in VISSION and the transparent intermixing of components written in different languages in the same network. This task would be facilitated by VISSION's architecture which separates the dataflow (MC++) and implementation (C++) concerns, as presented in chapters 4 and 6. However, an appropriate component model for VISSION's context has to be found in these languages, similarly to the class concept in C++. Another research direction would be to extend the dataflow concept to *code flow*. In code flow networks, source code, as well as data, could be produced and consumed by modules. This would allow a network to change its operational semantics dynamically depending on the user input, and thus allow a greater modelling power than the actual fixed-semantics dataflow networks. Implementing code flow in VISSION would be facilitated by the existence of the C++ interpreter in its kernel.

We conclude by remarking again that the success of any software system is ultimately determined by two factors: its coverage of the user requirements and its ease of use.

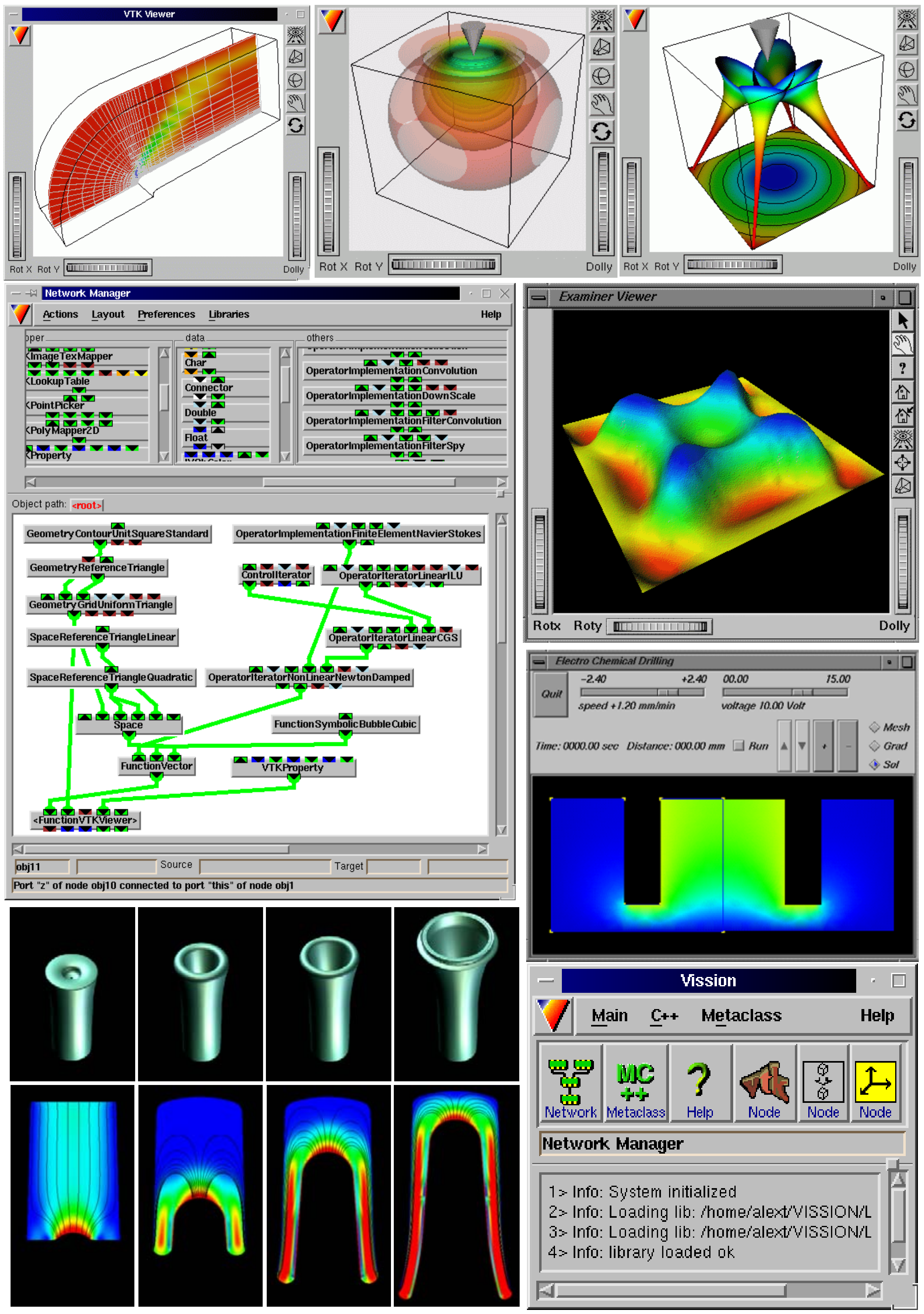
Concerning the first factor, we should not forget that the SimVis application domain is extremely broad, and that the requirements of its three user categories (end users, application designers, and component developers) are often conflicting. The VISSION system presented in this thesis improves some already existing mechanisms, such as visual dataflow and object-oriented programming, and removes some of the technical limitations of similar systems. However, there will always exist other systems which are better suited for a narrow application domain than a general-purpose environment like VIS-

SION.

Concerning the second factor, VISSION is a software product essentially built by a single programmer over a few years. Its ease of learn and use, measured in terms of its fault tolerance, end user interface, and documentation, is thus hard to be compared with those of commercial products built by tens of programmers over longer time periods. However, given a larger programming effort to be further invested in the current implementation, we believe that VISSION has a sound architecture to grow and compete with most similar SimVis environments.

Appendix A

Plates



Port "z" of node obj10 connected to port "this" of node obj1

1> Info: System initialized
2> Info: Loading lib: /home/alext/VISSION/L
3> Info: Loading lib: /home/alext/VISSION/L
4> Info: library loaded ok

Figure A.1: Simulation and visualisation applications in the VISSION environment (chapters 6-7)

Appendix B

MC++ Syntax

This appendix presents the syntax of the MC++ specification language used for C++ component integration in the VISSION system. This summary of the MC++ syntax is intended to be an aid to comprehension. Similarly to the presentation of the C++ language grammar by Stroustrup [102], the MC++ grammar described here accepts a larger set of MC++ constructs than the semantically valid ones. This is unavoidable, as, although syntactically simple, MC++ is a semantically rich language which covers constructs such as scoping and inheritance. As pointed out by several authors [2, 47, 102, 100], capturing such context-dependent semantics in a context-independent grammar is an almost impossible task.

Consequently, several context-dependent and other semantics-related rules are used by the actual MC++ parser implemented in VISSION to accept syntactically valid out meaningless constructs. As described in chapter 5, the MC++ parser uses information from the C++ interpreter and meta-entity table subsystems to provide the context needed for resolving the parsed lexical elements. The MC++ language analysis system of VISSION follows the classical structure encountered in most compilers and interpreters. First, a lexical analysis stage is performed to segment the MC++ input stream into the lexical elements described in section B.1. Next, these elements are passed to the MC++ parser which implements the MC++ language grammar described in sections B.2 and B.3.

B.1 Lexical Elements

As described in chapter 3, the MC++ language enhances the C++ language with dataflow semantics. However, MC++ does not define notions that already exist in C++, such as data types. Consequently, all MC++ lexical elements, or tokens, are imported form C++. Similarly, several more complex C++ constructs are directly used within MC++, such as C-style and C++ comments, expressions, type specifiers, code blocks, and function definitions.

This section introduces the lexical elements on which MC++ is built. There are nine such elements, as follows:

1. *Integer:* any valid positive integer, e.g. `0`, `1`, `100`.

2. *String:* any valid C++ string enclosed between double quotes, e.g. `"abcd"`, `"this is a text"`. As in C++, newline characters can be used to define strings on multiple lines.

3. *Identifier:* any valid C++ identifier, containing letters, digits, and the underscore, e.g. `abc01`, `diff_23`.

4. *Comments:* any valid C-style or C++ comment, e.g. `/* comment */`, `// comment`. Comments can occur anywhere in a MC++ source file. They are eliminated during the lexical analysis stage.

5. *Types:* any valid C++ type specifier, such as `int`, `const Object*`. The specified types must be meaningful in the framework of the types already loaded by the C++ interpreter.

6. *Expressions:* any valid C++ expression, or r-value. Simple string `"abcd"`, numerical `-12.34`, string `"abcd"` expressions, and so on, are accepted. Expressions involving constructors, e.g. `RGBColor(1.0,0.0,1.0)` are accepted as well, provided they are correct in the framework of the types loaded by the C++ interpreter.

7. *Code:* any valid block-level C++ code, such as { `int a=1; func(1.2,a+3);` }. The code may contain declarations and statements, e.g. function calls and loops, provided that it uses only symbols known by the C++ interpreter in the current scope.

8. *Function definitions:* any valid global-scope C++ function definition, e.g. `int f(float x)` { `return x+12.3;` }. The function definitions are subject to the same restrictions as the code definitions described above.

In the following sections, we shall use the symbols *integer*, *identifier*, *cpptype*, *cppexpr*, *cppcode*, and *cppfunc* to denote the corresponding elements described above. Strings are printed as **bold** text, while `courier` text will denote lexical constructs that appear verbatim in the MC++ source. Italic text denotes grammar rules. The construct $element_{opt}$ denotes zero or one occurrence of the grammar rule $element$. The construct $element^*$ denotes zero or more occurrences of the grammar rule $element$. When several right-hand expressions are listed on different lines for the same left-hand production name, any of them can be selected for that production name.

B.2 The Meta-Library

The program unit in MC++ is the meta-library (Sec. 3.6). The grammar rules describing the library follow.

$library$:

$header \quad entity^*$

$header$:

$libname \quad implem_{opt} \quad guilib^* \quad includelib^* \quad initcode_{opt} \quad finalcode_{opt}$

$libname$:

`library:` **sofilename**

$implem$:

`implementation:` **sofilename**

$guilib$:

`gui` $\{ \quad nodelib^* \quad portlib^* \quad editor^* \quad \}$

$nodelib$:

`nodelib` **sofilename**

$portlib$:

`portlib` **sofilename**

$editor$:

`editor` $cppeditorclassname \quad edtype \quad \{ \quad cpptypelist \quad \}$

$cppeditorclassname$:

$identifier$

$cpptypelist$:

$cpptypelist$ `,` $cpptype$

$cpptype$

$includelib$:

`include:` **mhfilename**

$initcode$:

`initialization:` $cppcode$

$finalcode$:

`finalization:` $cppcode$

B.3 The Meta-entity

Besides the header, described in the previous section, a MC++ library contains three main constructs: the metaclass (Sec. 3.3), the meta-group (Sec: 3.5), and the meta-type (Sec: 3.4). The grammar rules for these are described in the next sections.

B.3.1 The Metaclass

$entity$:

$module$

$group$

$type$

$module$:

`module` $modulename$ $basespec_{opt}$ $abstract_{opt}$ $\{$ $body$ $\}$

$basespec$:

`:` $baselist$

$body$:

$input\_spec^*$

$output\_spec^*$

$update\_spec_{opt}$

$category\_spec_{opt}$

$info\_spec_{opt}$

$modulename$:

$identifier$

$input\_spec$:

`input:` $port\_decl^*$

$output\_spec$:

`output:` $port\_decl^*$

$update\_spec$:

`update:` $update\_decl$

$category\_spec$:

`category:` **categoryname**

$info\_spec$:

`info:` **infotext**

$port\_decl$:

$porttype$ `*`$_{opt}$ **portname** $cpptype$ `(` $arglist$ `)`
`required`$_{opt}$ `optional`$_{opt}$ `no_icon`$_{opt}$ `preferred_ed`$_{opt}$

$porttype$:

`RDPortMem`

`RDPortMeth`

`WRPortMem`

`WRportMeth`

`RDPortSignal`

`WRPortSignal`

`RDPortDynArr`

$arglist$:
  $arglist$ `,` $cppmethodname$
  $arglist$ `,` $cppmembername$
  $cppmethodname$
  $cppmembername$

$cppmethodname$:
  $identifier$

$cppmembername$:
  $identifier$

$preferred\_ed$:
  `editor:` $cppeditorclassname$

$updatedecl$:
  $\{$ $cppmethodname$ $\}$
  $cppfunc$

$baselist$:
  $baselist$ `,` $modulename$ $hidden_{opt}$
  $modulename$ $hidden_{opt}$

$hidden$:
  `(` $hidden\_inports$ $hidden\_outports_{opt}$ `)`

$hidden\_inports$:
  $portnamelist$

$portnamelist$:
  $portnamelist$ `,` **portname**
  portname

$hidden\_outports$:
  `;` $hidden\_inports$

$abstract$:
  `=0`

B.3.2 The Meta-group

$group$:

`group` $groupname$ $basespec_{opt}$ { $groupbody$ }

$groupbody$:

$groupinput\_spec^*$
$groupoutput\_spec^*$
$children\_spec^*$
$connect\_spec^*$
$category\_spec_{opt}$
$info\_spec_{opt}$

$groupname$:

$identifier$

$groupinput\_spec$:

`input:` $groupportdecl^*$

$groupoutput\_spec$:

`output:` $groupportdecl^*$

$groupportdecl$:

$childname$. **portname** $port\_rename_{opt}$

$port\_rename$:

`as` **port_newname**

$children\_spec$:

`children:` $childdecl^*$

$childname$:

$identifier$

$childdecl$:

$modulename$ $childname$
$groupname$ $childname$

$connect\_spec$:

`connect:` $connectdecl^*$

$connectdecl$:

$childname$. **portname** idx_{opt} `to` $childname$. **portname** idx_{opt}

idx :

. $integer$

B.3.3 The Meta-type

$$
\begin{aligned}
type &: \\
&\texttt{type} \quad cpptype \quad \{ \quad typebody \quad \} \\
typebody &: \\
&store\_decl_{opt} \\
&default\_decl_{opt} \\
store\_decl &: \\
&\texttt{store:} \quad cppfunc \\
default\_decl &: \\
&\texttt{default:} \quad cppexpr
\end{aligned}
$$

Bibliography

[1] G. Abram, L. Treinish, *An Extended Data-Flow Architecture for Data Analysis and Visualization*, Proc. IEEE Visualization 1995, ACM Press, pp.263-270.

[2] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] G.S. Almasi and A. Gottlieb, *Highly parallel computing*, Benjamin/Cummings Publishing Company Inc., 1994.

[4] E. Anderson, Z. Bai, C. Bischof et al., *LAPACK user's guide*, SIAM Philadelphia, 1995.

[5] D.N. Arnold, F. Brezzi and M. Fortin, *A stable finite element for the Stokes equations*, Calcolo 21(1984), 337-344

[6] P. Astheimer *Sonification tools to supplement dataflow visualization*, Scientific Visualization: Advances and Challenges, Academic Press, 1994, pp. 251-263.

[7] O. Axelsson, *A generalized conjugate gradient, least square method*, Numerische Mathematik, 51(1987), 209-227

[8] O. Axelsson and J. Maubach, *Global space-time finite element methods for time-dependent convection diffusion problems*, Advances in Optimization and Numerical Analysis, 275(1994), 165-184

[9] O. Axelsson and P.S. Vassilevski, *Algebraic multilevel preconditioning methods I*, Numerische Mathematik, 56(1989), 157-177

[10] O. Axelsson and V.A. Barker, *Finite Element Solution of Boundary Value Problems*, Academic Press, Orlando, Florida, 1984

[11] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Domine, H. van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*, SIAM, Philadelphia, 1994.

[12] G. Booch, *Object-Oriented Analysis and Design*, Benjamin/Cummings, Redwood City, CA, second edition, 1994.

[13] J.C. Browne, *Parallel architectures for computer systems*, Physics Today, 37(5):28-35, 1984.

[14] 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.

[15] R. BRUN, S. RADEMAKERS *ROOT - An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in See also http://root.cern.ch/.

[16] T. BUDD, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1997.

[17] J.C. BUTCHER, *The numerical analysis of ordinary differential equations : Runge-Kutta and general linear methods*, Wiley, 1987

[18] T. CATARCI, M. F. COSTABILE, S. LEVIALDI, *Advanced Visual Interfaces*, Proc. International Workshop AVI '92, Rome, Italy, May 27-29, 1992, WSCS Press, 1992.

[19] M. F. COHEN, J. R. WALLACE, *Radiosity and Realistic Image Synthesis*, Academic Press, San Diego CA, 1993.

[20] M. F. COHEN, J. R. WALLACE, S. E. CHEN, D. P. GREENBERG, *A Progressive Refinement Approach to Fast Radiosity Image Generation*, Computer Graphics (SIGGRAPH '95 Proceedings), vol 22, no 4.

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

[22] INTRODUCTION TO ALGORITHMS, *T. H. Cormen, C. E. Leiserson, R. L. Rivest*, MIT Press, 1996.

[23] G. CORNELL AND C. S. HORSTMANN, *Core Java*, SunSoft Press, 1999.

[24] M. CROUZEIX AND P.A. RAVIART, *Conforming and nonconforming finite element methods for solving the stationary Stokes equations, I*, R.A.I.R.O., 3(1973), 33-76

[25] R.S. DEMBO, S.C. EISENSTAT AND T. STEIHAUG, *Inexact Newton methods*, SIAM Journal on Numerical Analysis, 19(1982), 400-408

[26] S. DEMEYER ET AL., *Design Guidelines for 'Tailorable' Frameworks*, Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 60-65.

[27] A. M. DUCLOS, M. GRAVE, *Reference models and formal specification for scientific visualization*, Scientific Visualization: Advances and Challenges, Academic Press, 1994, pp. 251-263.

[28] S.C. EISENSTAT AND H.F. WALKER, *Globally convergent inexact Newton methods*, SIAM Journal on Optimization 4(1994), 393-422

[29] ELECTRONICS STAFF, *Computer technology shifts emphasis to software: a special report*, Electronics, 8:142-150, May 1980.

[30] M. ELLIS, B. STROUSTRUP, *Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[31] A. ENZMANN, L. KRETZSCHMAR, C. YOUNG, *Ray Tracing Worlds With Pov-Ray*, Waite Group Press, 1993.

[32] V. ERVIN, W. LAYTON AND J. MAUBACH, *A Posteriori error estimators for a two level finite element method for the Navier-Stokes equations*, Numerical Methods for PDEs, 12(1996), 333-346

[33] P. M. FERGUSON, *The Motif Reference Manual*, O'Reilly and Asociates, 1994.

[34] T. FRÜHAUF, M. GÖBEL, K. KARLSSON, *Design of a Flexible Monolithic Visualization System, Scientific Visualization: Advances and Challenges*, Academic Press, 1994, pp. 265-286.

[35] R. P. GABRIEL, *Patterns of Software - Tales from the Software Community*, Oxford University Press, New York, 1996.

[36] G. GAINES, D. CARNEY, J. FOREMAN, *component-Based Software Development/COTS Integration*, Software Technology Review, Software Engineering Institute, Carnegie Mellon University, 1997.

[37] E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[38] H. GARCKE, T. PREUSSER, M. RUMPF, A. TELEA, U. WEIKARD, J.J. VAN WIJK, Proceedings of IEEE Visualization 2000, ACM Press, 2000.

[39] C. W. GEAR, *Numerical initial value problems in ordinary differential equations*, Prentice-Hall, 1971

[40] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The Language and its Implementation*, Addosin-Wesley, 1983.

[41] 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

[42] W. HACKBUSCH AND G. WITTUM (EDS.), *ILU algorithms, theory and applications, Proceedings Kiel 1992, in Notes on numerical fluid mechanics 41*, Vieweg, 1993

[43] W. HACKBUSCH AND G. WITTUM (EDS.), *Multigrid Methods, Proceedings of the European Conference in Lecture notes in computational science and engineering*, Springer Verlag, 1997

[44] M. A. HALSE, *IRIS Explorer User's Guide*. Silicon Graphics Inc., Mountain View, California, 1993.

[45] W. HIBBARD, D. SANTEK, *The Vis5D System for Easy Interactive Visualization*, Proc. IEEE Visualization '90, ACM Press 1990, pp. 129-134.

[46] D.H. HILS, *Visual languages and computing survey: data flow visual programming languages*, Journal of Visual Languages and Computing, 3:69-101, 1992.

[47] J. HOLMES, *Object-Oriented Compiler Construction*, Prentice Hall, 1994.

[48] IMSL, *FORTRAN Subroutines for Mathematical Applications, User's Manual*, IMSL, 1987.

[49] INRIA-ROCQUENCOURT, *Scilab Documentation for release 2.4.1*, `http://www-rocq.inria.fr/scilab/doc.html`, 2000

[50] 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

[51] V. JOHN, J.M. MAUBACH AND L. TOBISKA, *A non-conforming streamline diffusion finite element method for convection diffusion problems*, Numerische Mathematik 78(1997), pp. 165-174

[52] R. E. JOHNSON, *Frameworks = (Components + Patterns)*, Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 39-42.

[53] KHORAL RESEARCH INC., *Khoros Professional Student Edition*, Khoral Research Inc, 1997.

[54] G. KICZALES, J. DES RIVIERES, D. G. BOBROW, *The Arto of the Metaobject Protocol*, MIT Press, 1991.

[55] L. KLANDER, *Core Visual C++ 6.0*, Prentice Hall, 1999.

[56] G. E. KRASNER, S. E. POPE, *A cookbook for using the model view controller user interface paradigm in Smalltalk-80*, Journal of Object-Orientated Programming, 1(3):26-49, August/September 1988.

[57] W. LAYTON, J.M. MAUBACH AND P. RABIER, *Robust methods for highly non-symmetric problems*, Contemporary Mathematics, 180(1994), 265-270

[58] S. LIPPMAN, *Inside the C++ Object Model*, Addison-Wesley, 1996.

[59] B. LISKOV, S. ZILLES, *Programming with Abstract Data Types*, SIGPLAN Notices, vol. 9, no. 4, April 1974, pp. 50-59.

[60] W. LORENSEN, H. CLINE, *Marching cubes: A high -resolution 3D surface construction algorithm*, Proc. SIGGRAPH '91, *Computer Graphics*, no. 25, ACM Press, 1991, pp. 285-288.

[61] MARC ANALYSIS RESEARCH CORPORATION, *On the MARC Newsletter*, MARC Analysis Research Corporation, 1991-1999. Also online on `http://www.marc.com`.

[62] S.D. MARGENOV AND J.M. MAUBACH, *Optimal algebraic multilevel preconditioning for local refinement along a line*, Journal of Numerical Linear Algebra with Applications 2(1995), 347-362

[63] R. MARTIN, *Design Patterns for Dealing with Dual Inheritance Hierarchies in C++*, C++ Report, SIGS Publications, April 1997.

[64] R. MARTIN, *The Open Closed Principle*, C++ Report, SIGS Publications, January 1996.

[65] R. MARTIN, *The Liskov Substitution Principle*, C++ Report, SIGS Publications, March 1996.

[66] MATLAB, *Matlab Reference Guide*, The Math Works Inc., 1992.

[67] R. M. M. MATTHEIJ AND J. MOLENAAR, *Ordinary differential equations in theory and practice*, Wiley, 1996

[68] J. MAUBACH, *Local bisection refinement for n-simplicial grids generated by reflections*, SIAM Journal on Scientific Computing, 16(1995), 210-227

[69] B. MEYER, *Object-oriented software construction*, Prentice Hall, 1997

[70] MICROSOFT, *The Component Object Model Specification*, Seattle WA, 1995.

[71] J. D. MULDER, *Computational Steering with Parametrizable Geometric Objects*, PhD thesis, 1998, Universiteit van Amsterdam, Wiskunde en Informatica, The Netherlands

[72] J. D. MULDER, J. J. VAN WIJK, *3D computational steering with parametrized geometric objects*, Proceedings of the 1995 Visualization Conference, eds. G. M. Nielson and D. Silver, p.304-311, 1995.

[73] NAG, *FORTRAN Library, Introductory Guide, Mark 14*, Numerical Analysis Group Limited and Inc., 1990.

[74] C. NELSON, *Tcl/Tk: Programmer's Reference*, McGraw-Hill, 1999.

[75] O. NIERSTRASZ AND L. DAMI, *Component-Oriented Software Technology*, in *Object-Oriented Software Composition*, eds. O. Nierstrasz and D. Tsichritzis, Prentice Hall, 1995.

[76] M. J. NOOT, A. C. TELEA, J. K. M. JANSEN, R. M. M. MATTHEIJ, *Real Time Numerical Simulation and Visualization of Electrochemical Drilling*, in *Computing and Visualization in Science*, No 1, 1998

[77] A. NYE, *The Xlib Programming Manual*, O'Reilly and Asociates, 1990.

[78] A. NYE, *The X Toolkit Programming Manual*, O'Reilly and Asociates, 1990.

[79] H. G. PAGENDARM, *HIGHEND, A Visualization System for 3D Data with Special Support for Postprocessing of Fluid Dynamics Data*, in *Visualization in Scientific Computing*, edited by M. Grave, Y. LeLous, W. T. Hewitt, pp. 87-98, Springer, 1994.

[80] D. L. PARNAS, *On Criteria to be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No.12, Dec. 1972, pp. 1053-1058.

[81] S. G. PARKER, D. M. WEINSTEIN, C. R. JOHNSON, *The SCIRun computational steering software system*, in E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 1-40, Birkhaeuser Verlag AG, Switzerland, 1997

[82] E. PEETERS, *Design of an Object-Oriented, Interactive Animation System*, Ph.D. thesis, Eindhoven University of Tehcnology, Mathematics and Computing Science, 1995.

[83] L. PINSON, R. WIENER, *Objective-C : Object-Oriented Programming Techniques*, Addison-Wesley, 1991.

[84] PRINCIPIA MATHEMATICA INC. *The Visualization Studio Pipeline Editor*, online product documentation: `http://www.principiamathematica.com`

[85] A. RASHID, D. PARSONS, A. SPECK, A. TELEA, *A "Framework" for Object Oriented Frameworks Design*, Proceedings of TOOLS'99 Europe, eds. R. Mitchell, A.C. Wills, J. Bosch, B. Meyer, ACM Press, 1999.

[86] 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

[87] W. REISIG, *Petri Nets: An Introduction*, Springer-Verlag, 1985

[88] W. RIBARSKY, B. BROWN, T. MYERSON, R. FELDMANN, S. SMITH, AND L. TREINISH, *Object-oriented, dataflow visualization systems - a paradigm shift?*, in *Scientific Visualization: Advances and Challenges*, Academic Press (1994), pp. 251-263.

[89] F. M. RIJNDERS, *A Visual Programming Environment for Scientific Applications: Possibilities and Limitations*, PhD thesis, Vrije Universiteit Amsterdam, 1995.

[90] H.G. ROOS, M. STYNES AND L. TOBISKA, *Numerical Methods for Singularly Perturbed Differential Equations*, Springer Verlag, 1996

[91] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, W. LORENSEN. *Object-Oriented Modelling and Design*, Prentice-Hall, 1991

[92] Y. SAAD AND M.H. SCHULTZ, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7(1986), 856-869

[93] B. SCHNEIDERMANN, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, second edition, 1992

[94] W. SCHROEDER, K. MARTIN, B. LORENSEN, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1995

[95] J. SIEGEL, *CORBA Fundamentals and Programming*, John Wiley & Sons Inc. New York, 1996.

[96] SILICON GRAPHICS INC., *ShowCase User Guide*, Mountain View CA, 1993.

[97] SILICON GRAPHICS INC., *CaseVision Software Developer Guide*, Mountain View CA, 1995.

[98] F. X. SILLION, C. PUECH, *Radiosity and Global Illumination*, Morgan Kaufmann Publishers Inc., San Francisco, 1994.

[99] P. SLUSALLEK, *Vision: An Architecture for Physically-Based Rendering*, PhD thesis, University of Erlangen, Germany, 1995.

[100] C. SMINCHISESCU, A. TELEA *An Object-Oriented Approach to C++ Compiler Technology*, PhDOOS'99 Workshop Report, in *Object-Oriented Technology: ECOOP'99 Workshop Reader*, eds. A. Moreira, S. Demeyer, Springer, 1999, pp. 116-135.

[101] I. SOMMERVILLE, P. SAWYER, *Requirements Engineering: a Good Practice Guide*, John Wiley and Sons, 1997.

[102] B. STROUSTRUP, *The C++ Programming Manual*, Addison-Wesley,1993.

[103] J. SHARP, *Data flow computing*, Ellis Horwood Ltd., Chichester, 1985.

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

[105] C. SZYPERSKI, *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[106] A. TELEA, *Combining Object Orientation and Dataflow Modelling in the* VISSION *Simulation System*, Proceedings of TOOLS'99 Europe, eds. R. Mitchell, A.C. Wills, J. Bosch, B. Meyer, ACM Press, 1999.

[107] A. TELEA, C.W.A.M. VAN OVERVELD, *An Object-Oriented Interactive System for Scientific Simulations: Design and Applications*, in *Mathematical Visualization*, H.-C. Hege and K. Polthier (eds.), Springer Verlag 1998

[108] A. TELEA, C. W. A. M. VAN OVERVELD, *The Close Objects Buffer: A Sharp Shadow Detection Technique for Radiosity Methods*, the *Journal of Graphics Tools*, Volume 2, No 2, ACM Press, 1997.

[109] A. TELEA, J. J. VAN WIJK, SMARTLINK*: An Agent for Supporting Dataflow Application Construction*, in Proc. IEEE VisSym 2000, eds. R. van Liere, W. Ribarsky, Springer, 2000.

[110] A. TELEA, J. J. VAN WIJK, VISSION*: An Object Oriented Dataflow System for Simulation and Visualization*, Proceedings of IEEE VisSym '99, Springer, 1999.

[111] A. TELEA, J. J. VAN WIJK, *Simplified Representation of Vector Fields*, Proceedings of IEEE Visualization '99, ACM Press, 1999.

[112] M. J. TODD, *The Computation of Fixed Points and Applications*, Lecture Notes in Economics and Mathematical Systems 124, Springer Verlag, 1976

[113] 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.

[114] P. WALATKA, P. BUNING, L. PIERCE, P. ELSON, *PLOT3D User's Manual*, NASA Technical Memorandum 101067, March 1990.

[115] T. VAN WALSUM, F. H. POST, D. SILVER, F. J. POST, *Feature Extraction and Iconic Visualization*, IEEE Transactions on Visualization and Computer Graphics, vol. 2, no. 2, June 1996, pp. 111-119.

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

[117] 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

[118] S. WOLFRAM, *The Mathematica Book*, 4th edition, Cambridge University Press, 1999.

[119] M. WOO, J. NEIDER, T. DAVIS, D. SHREINER, *The OpenGL Programming Guide*, 3rd edition, Adison-Wesley, 1999.

[120] E. YOURDON, L. CONSTANTINE, *Structured Design: fundamentals of a Discipline of computer Program and Systems Design*, Prentice Hall, 1979.

Summary

Better insight in complex physical processes requires the integration of scientific visualization and numerical simulation in a single interactive software framework. Typical simulation and visualisation (SimVis) frameworks for research applications distinguish between three user roles. End users steer and monitor a simulation via virtual cameras, direct manipulation, and graphics user interfaces. Application designers construct applications by assembling domain-specific components visually in a dataflow network. Component developers write or extend components using a programming language.

Although numerous SimVis frameworks exist, few of them provide sufficient freedom for all three user roles and allow the user to easily switch between these roles. Their most important limitations regard intrusive code integration, limited automation for visual interface construction, and limited support of custom (object-oriented) data types.

This thesis describes the design and implementation of VISSION, a general-purpose environment for VIsualization and Steering of SImulations with Object-oriented Networks. VISSION addresses the above SimVis systems' limitations by introducing a new method for combining object-oriented application code and dataflow code non intrusively. The application code, provided as C++ class libraries, is not constrained by the VISSION system, as in most other SimVis frameworks. The interface between these libraries and VISSION is provided at a meta-language level, by the MC++ language. MC++ adds dataflow notions such as data inputs, outputs, and update operations to the C++ classes in a non intrusive, white box manner. The original code remains unchanged, which makes it easy to maintain, extend, or replace. The component-based architecture of VISSION provides also the automatic construction of the component icons used for visual network editing and graphics user interfaces used for application steering and monitoring. VISSION's architecture, based on a single component development language and automatic visual interface construction, ensures an easy role transition from component development to application design and use.

A large number of SimVis applications built with VISSION is presented. They cover several domains such as finite elements and finite difference simulations, scientific visualization, animation, and realistic rendering. These applications are based on the integration in VISSION of several existing class libraries, such as Open Inventor and VTK, or newly designed ones, such as the NUMLAB computational library. Overall, the VISSION system proves to be as flexible and easy to use for its end users and application designers as similar commercial systems, while being much more versatile and extendable for its component developers.

Samenvatting

Beter inzicht in complexe physische processen vraagt om de integratie van wetenschappelijke visualisatie en numerieke simulatie in een interactief software framework. Typische simulatie en visualisatie (SimVis) frameworks voor onderzoek maken een onderscheid tussen drie gebruikersroles. Eindgebruikers besturen en monitoren een simulatie met behulp van virtuele cameras, direkte manipulatie, en graphische user interfaces. Applicatieontwerpers construeren de toepassingen door het assembleren van gebied-specifieke componenten in een dataflow netwerk op een visuele manier. Componentontwikkelaars maken gebruik van een programmeringstaal om componenten te schrijven of uit te breiden.

Hoewel er een groot aantal van SimVis frameworks bestaat, weinigen leveren genoeg vrijheid op voor alle drie gebruikersroles en brengen de gebruiker in stand om makkelijk van role te wisselen. Hun meest belangrijke beperkingen betreffen de intrusieve codeintegratie, beperkte automatisering van de constructie van visuele interfaces, en beperkte ondersteuning voor gebruikerspecifieke (objekt-georiënteerde) datatypen.

Dit proefschrift beschrijft de ontwerp en implementatie van VISSION, een algemene omgeving voor VIsualisatie en beStuur van SImulaties met Objekt-georiënteerde netwerken. VISSION beantwoort de bovenstaande beperkingen van SimVis systemen door de introductie van een nieuwe methode om objekt-georiënteerde applicatie-code met dataflow-code te combineren op een niet intrusieve manier. De applicatie-code, ingeleverd als C++ classenbibliotheken, wordt niet beperkt door het VISSION systeem, zoals het gebeurt met meest andere SimVis frameworks. De interface tussen deze bibliotheken en VISSION wordt uitgevoerd op een meta-taal niveau, met behulp van de MC++ taal. MC++ voegt dataflow noties, zoals in- en uitvoeren en update operaties, bij de C++ classen, op een niet intrusieve, *white-box* manier. De oorsprongelijke code blijft onveranderd, dus makkelijk te onderhouden, uit te breiden, of vervangen. De component-gebasseerde architectuur van VISSION levert ook de automatische constructie van componenteniconen op die worden gebruikt voor de visuele netwerkseditering, evenals de graphische user interfaces die gebruikt worden voor het besturen en monitoren van applicaties. De architectuur van VISSION, die gebasseerd is op een enkele taal voor componentontwikkeling en automatische constructie van de visuele interfaces, bepaalt een makkelijke roltransitie van componentontwikkeling naar applicatieontwerp en gebruik.

Een groot aantal SimVis toepassingen gemaakt met VISSION wordt voorgesteld. Deze toepassingen bedekken verscheidene gebieden zoals eindige elementen en eindige differenties simulaties, wetenschappelijke visualisatie, animatie, en realistische rendering. Deze toepassingen zijn gebasseerd op de integratie van verschillende bestaande classenbibliotheken in VISSION. zoals Open Inventor en VTK, of nieuwe bibliotheken, zoals de NUMLAB computationele bibliotheek. Alles samengesteld bewijst VISSION zich om net zo flexibel en makkelijk te gebruiken te zijn voor zijn eindgebruikers en applicatieontwerpers als similaire comerciële systemen en om meer versatiel en uitbreidbaar te zijn voor zijn componentontwikkelaars.

Curriculum Vitae

Alexandru Telea was born in Bucharest, Romania, on July 16th, 1972. After completing his pre-university education at the Informatics Highschool in Bucharest, he started in the same year to study at the Polytechnical University of Bucharest. During the five years of faculty study, he received three three-months study grants at the Eindhoven University of Technology (EUT). He graduated in September 1996 in electrical engineering and computer science on the subject of radiosity lighting simulations, carried at EUT under the supervision of dr. C.W.A.M. van Overveld.

From September 1996, he worked as a trainee research assistant (AIO) on an interdisciplinary project in the groups Scientific Computing and Technical Applications at the faculty of Mathematics and Computing Science of the TUE. The research on scientific simulation and visualisation and object-oriented software construction carried out in the period September 1996 - June 2000 led to this thesis.

The work on the VISSION system described in this thesis began in September 1997, inspired both by the author's discussion with the Oorange development team at the VisMath '97 symposium in Berlin and his previous experience with the AVS system. Development of VISSION and growth of its user group continues in the present, after its first stable release in spring 1998.