SCOPIRA - A SYSTEM FOR THE ANALYSIS OF BIOMEDICAL DATA

Aleksander B. Demko, Nicolino J. Pizzi, Ray L. Somorjai
Institute for Biodiagnostics, National Research Council
435 Ellice Avenue, Winnipeg MB, R3B 1Y6
pizzi@nrc.ca

ABSTRACT

With the proliferation of high-dimensional biomedical data, an acute need exists for a comprehensive, user-friendly software suite that allows investigators, in the health care disciplines, to classify their data through the detection of discriminating features. Scopira is a software initiative that attempts to achieve these goals in addition to providing intuitive visual computation, logic construction and parallel execution. In this paper we describe the architecture of Scopira, and various design and implementation issues that surfaced during development.

Keywords: *Biomedical Data Analysis, Software Engineering, Parallel Computation.*

1. INTRODUCTION

As an algorithm builder, Scopira allows the placement of multiple algorithm modules and their connections to each other to form complex systems. An intuitive visual layout paradigm is employed to display and manipulate the network of modules and their connections. Inter-module connections may be quite complex, allowing for typical programming constructs such as loops and decision trees. All inter-module data are tightly typed in a hierarchical, object-oriented, data type tree. Configuration of module parameters is also performed visually with immediate feedback.

After testing, the system of modules may be mapped onto a cluster of computers to better utilize processing power while decreasing execution time. Modules need not be aware of this networking - Scopira transparently provides this service. Nevertheless, Scopira provides an interface for developers to optionally control parallelism at the intra-module level.

As a development environment for experienced data analysts, Scopira facilitates the creation of new modules, data types and functions that integrate seamlessly with existing services. Developers may quickly build graphical interfaces to their algorithms using Scopira's supplied user interface tool kit and simplified programming interface.

Scopira is written completely in C++, with major focus on ANSI compliance, proper design and maximum efficiency. Targeted at Linux desktops and clusters, Scopira is usable on many UNIX variants and may utilize many specialized high performance compilers.

1.1. Application

Scopira defines a general framework for modules and module interaction and does not specify an application domain.

Several modules were built to perform data classification and framework testing. These include: A genetic algorithm module to implement near-optimal region selection for feature space reduction[1]. A regularized multilayer perceptron, Profile Analysis, principle component analysis, linear discriminant analysis, fuzzy c-means[2] clustering, and half-space median[3] modules.

A set of general modules were also developed to deal with the bundled data types. These modules provide generic facilities such as data loading and saving, matrix splicing and merging, output generation and statistical functions.

2. USER FACILITIES

For the user (non-developer), Scopira focuses on giving the user maximum control over the module system while still maintaining a straightforward and consistent interface.

2.1. Maps and Modules

A Scopira map contains zero or more modules. Each module is defined as a self-contained algorithm and contains a collection of zero or more slots. Slots may further be divided into input slots and output slots, depending on whether they take data or produce it, respectively.

An input and output slot may be joined to form a slot connection. Each slot may be connected to any number of slots of the opposite type.

Slots exchange Scopira data objects. Each input slot has a queue of zero or more pending data objects. When an output slot sends a data object, that data object gets queued to each input slot that is attached to it. Certain input slots may be deemed necessary for module execution. Only when all of these input slots have at least one data object, will the module *fire* (execute). After a module fires, one data object is consumed from the data queues within each input slot.

2.2. Visual Display

To assist the user in map construction, a visual interface may be used. Modules are represented as titled icons, whereas connections between modules are represented by connecting lines (Fig. 1). The user interactively places and connects modules on the map. Various windows may be brought up to help debug and monitor modules and data as execution progresses.

Each modules may optionally define a set of properties. Properties are typically settings or parameters that affect how the module executes. Properties are implemented as slots, and may be connected to other modules as regular slots. In addition to regular connections however, properties may have their values set through a visual interface presented to the user. This allows the user to experiment with various properties interactively, with immediate feedback.

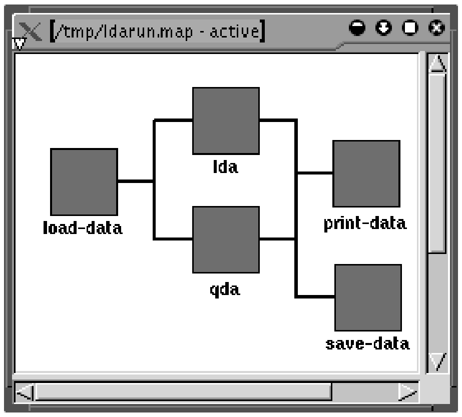


Fig. 1. Visual map display

2.3. Scripting

The more adept user may use the provided scripting language to automate the process of map construction and execution. The scripting language provides full access to the Scopira system, allowing the user to assemble modules, set properties and execute maps. Maps created with the visual interface may also be loaded into the scripting system. This allows the user to debug maps visually, and then automate their runs in batches, non-interactively.

For the application developer, Scopira may be embedded directly within an application. The Scopira core is itself a shared library with a straightforward, object-oriented interface than may be linked with any application. The visual display and scripting applications use Scopira in this manner. This allows unlimited customization possibilities with regard to visual front ends and information processing automation.

3. DEVELOPMENT FRAMEWORK

Scopira allows developers to extend the framework along four major areas; compound modules, data types, functions and proponents. Developers then package these extensions in a Scopira *kit* for distribution.

3.1. Modules

An algorithm kernel is the basic implementation of a particular algorithm or operation. A module contains a kernel, state information and data slots. With these slots, modules exchange data with other modules within a map.

This abstraction lets the module developer focus on writing the algorithm-specific kernels for their modules. He uses several Scopira methods to extract input data and post output data. Scopira handles data transport and ensures that input slots marked as *required* have data before calling the core algorithm.

Scopira comes with a bundle of useful and generic modules. This includes modules that load/save data in various formats, print data to log and interactively obtain user supplied data. Finally, some of these modules perform generic data manipulation (for example, array splicing and merging) and others perform common statistical and mathematical functions. All these modules operate on the bundled data types and user-defined descendants.

3.2. Data Types

Each input and output slot is associated with a data type. This denotes what type of data the slot produces (for output slots) or accepts (for input slots). Scopira will allow two slots to be connected only if their data types are *compatible* (see below).

Internally, Scopira maintains a *data type tree* (actually an acyclic, directed graph), with each node of this tree representing one data type. Each data type in this tree (excluding the top root node, *void*) has one or more parents.

In this tree, each node is considered a descendant of its parents, and as such, is said to extend these parents. This extension allows the data type to be treated as if it were any of its parent data types (*or any ancestor type for that matter*).

Scopira considers two data types to be compatible if, and only if, the two data types are identical, or if one is an ancestor of the other within the type tree. This fundamental concept of inheritance and polymorphism is borrowed from object-oriented methodology and allows modules to operate on data types that were developed after their inception. By simply defining what base data type a module needs, that module may then operate on all descendants of that type without modification.

Each data type presented to the user maps directly to a C++ class. Module developers that wish to introduce new types to the system, may simply *register* their type with Scopira by specifying where in the Scopira data type tree their type should be added. Scopira will then insure connection-time type checking, based on its location in the type tree.

Scopira takes advantage of the generic programming facilities of the C++ language by using template classes to make up the base of most of its core data types. Developers may easily create similar structures of new types by simply *instantiating* these template classes.

3.3. Micro Functions

Micro functions are standard C++ functions that are registered and managed by Scopira. An algorithm that does not require the interface or state-keeping features of Scopira modules may be implemented as a micro function. This is particularly beneficial to algorithms that will be called, perhaps very frequently, from within other modules. Calling a micro function is direct, with less overhead, than sending the data out through output slots.

The primary goal of micro functions is performance. By bypassing Scopira's data transport and event scheduling mechanisms, micro functions incur very little overhead. In fact, using a micro function within a module suffers only one pointer indirection of performance penalty, compared to calling a named function directly.

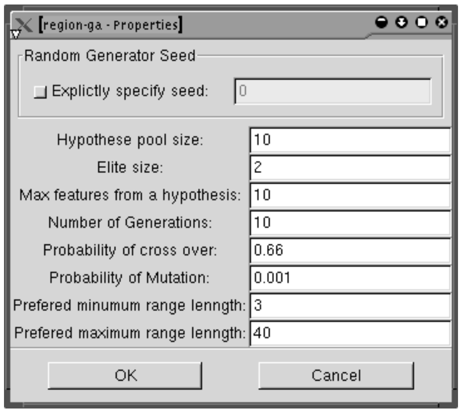
3.4. *Proponents*

Proponents ("property aware components") are visual, user interface components that are designed to control the properties of a module. They allow the user to inspect, visualize and possibly set the properties interactively, from within the graphical user interface.

Proponents operate on a module's data, and not the module itself. This decoupling allows proponents to be used on a variety of modules, and conversely, allows modules to select from a pool of proponents to build their configuration screens.

Scopira includes a set of generic proponents that operate on the bundled data types (Fig. 2), as well as proponents that visualize data in an output-only fashion.

Module writers are free to make their own proponents, either by extending an existing proponent or by building one from scratch. Proponent interface complexity is limited only by the graphical user interface toolkit provided by the system. This gives the module developer full power to make radically new and possibly complex interfaces.



Explictly specify seed:	0
Hypothese pool size:	10
Elite size:	2
Max features from a hypothesis:	10
Number of Generations:	10
Probability of cross over:	0.66
Probability of Mutation:	0.001
Prefered minumum range lenngth:	3
Prefered maximum range lenngth:	40

Fig. 2. configuration screen built with proponents

3.5. User Front Ends

The complete Scopira engine core is compiled into a shared core library. This library exposes a straightforward, object-oriented interface to map construction and manipulation.

This allows C and C++ developers to embed Scopira in their applications. This ability naturally extends to any language that may call C functions - such languages include Java, Perl and Python. The visual map editing interface and scripting system are both implemented in this manner.

4. SYSTEM ARCHITECTURE

The Scopira frame work is designed to be modular, portable, adaptable and distributed. This was achieved through various software engineering techniques.

4.1. Core Design

Scopira is divided into three large software components: the engine core, the front ends, and the back end computation kits.

The engine core is the central coordinator of a Scopira system. It is responsible for loading, maintaining and executing modules within maps. The engine manages intermodule data transport as well as data clean up.

The engine has three, run-time selectable event scheduling schemes. These schedulers decide which, where and when modules will run, possibly in parallel.

The uni-thread scheduler runs events sequentially, using only one operating system thread. The multi-thread scheduler attempts to maximize a multi-CPU machine by paralleling module execution on a single machine. The network-aware scheduler manages a collection of separate machines, connected via a network, each with any number of processors. This scheduler may partition a map over these network nodes transparently, without requiring any special programming by the module developer.

The engine core manages kits. A kit may contain any number of developer supplied module kernels, micro functions, data types and graphical proponents. Kits are implemented as shared code libraries, dynamically loaded at run-time and selected by the user. They may be developed and distributed independently from the engine and the rest of Scopira.

Finally, the front ends interact with the engine through its exposed, object-oriented interface. The interactive visual map editor and scripting systems both use this interface to manipulate and execute maps. Custom front ends may be built in a straight forward manner, with no need to rebuild any part of Scopira.

4.2. Portable

Scopira follows standard software engineering methods to maximize code portability.

Platform dependent code like the thread and network communication systems are encapsulated within objects. Any changes required to these systems for new platforms or compilers need only be made to these objects.

The only external routines used by the Scopira engine core and standard kits are those provided by the Standard C++ library and the system level C libraries. Because of this, it is quite straightforward to port the engine core and kits to other platforms.

The visual front end depends on the GTK+ graphical user interface library. This library is portable to all UNIX platforms - with a beta Win32 port. The script front end uses the GUILE/Scheme system, which is portable to all UNIX platforms. Portability to Win32 is assured, at least via the Cygwin tools.

To maintain clean, portable code throughout the various development phases of the project, Scopira is routinely compiled and tested under various compilers. Currently, this list includes GNU C++, PGI C++ and Intel C++.

4.3. Generic Programming

Scopira attempts to introduce as little overhead or indirection to the modules as possible, while still providing a convenient, object-oriented, type-safe interface. Unnecessary slow downs may have a notable cost when dealing with maps that run for long periods of time over a cluster of machines. Performance improvements may increase the throughput of a cluster and thus decrease the need to purchase more nodes.

One technique Scopira uses to achieve performance gains is to follow the C++ use of generic templates. This in-lining and specialized instantiations of constructs for exactly the types required gives the developer the exact objects he needs without the need for indirection, common base classes, or forcing everyone to use the same simple data types. For instance, with generic classes, developers are not forced to work with double types when they want the smaller float types.

4.4. Parallelism

Scopira supports parallel execution at two levels.

At the inter-module level, Scopira transparently schedules and executes modules simultaneously. Scopira does this by selecting a combination of modules to execute from the current run queue that would maximize the current state of free processors. All this is done transparently to the module developer.

At the intra-module level, Scopira provides an MPI-like interface that allows modules to request and use many processors within the context of their execution. This allows modules to be parallelized without having to be broken up into smaller units. Unlike MPI, Scopira may inherently *serialize* many of its data types, thus removing many of the error prone low level data transport calls that MPI requires.

Throughout both levels, Scopira constantly maintains and monitors the amount of processing resources, constantly attempting to maximize computational throughput.

5. DEVELOPMENT ISSUES

5.1. Module Granularity

When decoupling or partitioning a large algorithm into a set of connectable, logical modules, one will almost always trade some performance for the benefits of decoupling. This occurs because the new modules are made to be more general than their functional equivalents in the older, larger algorithms. As the communication between modules have to go through the more generic mechanism of Scopira's slots and data types, modules may no longer take advantage of being tightly coupled and passing messages with complete freedom.

The biggest challenge facing a module developer who wants to convert a large algorithm to a set of Scopira modules is the partitioning scheme. Partitioning an algorithm into many modules requires additional effort by the module developer. More importantly, however, the ratio of time that Scopira spends moving data between modules and the time actually spent running within each module increases. This is hardly desirable, but in exchange, users of the modules are now able to swap out certain modules without swapping out the whole algorithm.

At the other end of the partitioning spectrum, module developers break their algorithms into a few modules. This may be done relatively quickly and allows the module users to reassemble the full algorithm almost instantly. The users pay a price for this convenience though, as now they may not replace parts of the system without having to duplicate other parts. This occurs because the parts they want to replace are often bundled in the same module with parts they do not want to replace.

Ultimately, it is up to the module developer to decide on the level of algorithm partitioning. The proper balance must be met between flexibility and performance. In some cases, module developers may predict what parts of their algorithms users may want to replace. This may aid in the decision process, and may often be used as a guide for the level of partitioning with the other parts of the algorithm.

Alternatively, developers may choose to implement common, straightforward algorithms not as modules, but instead as micro functions.

5.2. Polymorphism and Generics

C++ offers both extensive polymorphic object-oriented facilities as well as extensive generic abilities via templates. Often thought as complementing each other, they are, in fact, opposing paradigms.

Run-time polymorphism is achieved through the use of virtual methods in pre-defined classes. If a method or object would like to operate on an unknown class, then it must at least specify the base interface of virtual methods that the class must implement. When it operates on this class, all virtual method calls will go through their virtual method table and be bounced to the real, final methods. This level of indirection is the cost of having run-time polymorphism.

Generic programming is a form of compile-time polymorphism. Entire classes and methods are developed to operate on unknown types - requiring these types to follow a certain *form*. This form dictates what methods, operators and other characteristics the unknown type must supply. However, unlike run-time polymorphism, these requirements may be fulfilled not only by virtual methods, but also by friend functions, operators and standard operators. This coupling of type and generic algorithm is done completely at compile time, resolving in no run-time indirections costs. The price of this performance and type-safety is of course, run-time polymorphism.

Scopira attempts a balance between these two paradigms. For performance, all data types and core algorithms are generic and template-based. However, the modules presented to the user must have be well defined, virtual interfaces. Therefore, many modules operate on the most popular data types and the generic ones are explicitly instantiated on similar popular data types. The *popular* data types serve as a higher level common ground. For systems where it is not possible to use or convert to these popular data types, developers may explicitly instantiate the algorithms as needed.

5.3. C++ Programming

The C++ programming language is relatively large and complex. Its flexibility and power, used improperly, may be a great source of errors and developer confusion. Modern C++ programs like Scopira take full advantage of many language facilities, many of which were only standardized or implemented in compilers quite recently.

Scopira allows *legacy* algorithms - that is, algorithms developed in traditional C, but not specifically in Scopira - to be used directly in Scopira. Each legacy algorithm needs converter code that converts Scopira data input and output to and from the format required by the algorithm. The Scopira scheduling system will also take care not to execute two instances of a legacy module in parallel. Scopira does this because the legacy algorithm may be using global variables for its state information, making parallel execution impossible.

Developers also have the option of keeping their data model and still take advantage of Scopira's inter-module parallelism. They do this by making sure all the state information for their algorithm may be encapsulated in a class, which in turn may be instantiated and controlled by Scopira.

Finally, developers may take the full plunge and convert their algorithms to use the Scopira data model and core algorithms.

Most core algorithms and data structures are generic structures within various C++ *name spaces* and encourage the use of *auto pointers* and *reference counting*. Name spaces allow for more logical code organization and reduce naming conflicts. Auto pointers and reference counting help to debug and manage dynamic memory. Rather than having direct access to the data, developers must use various access functions to read and manipulate the encapsulated data. Standard C++ method in-lining techniques reduce the cost of type-safe access methods to zero - that is, adding no overhead at all. During *debug* builds, these access methods do range checking. Developers accustomed to C arrays will find these classes fast, convenient and straight forward. Various vector and matrix classes also have methods that allow direct access to the C-array datum.

Internally, we found that non-C++ programmers made the transition to C++ gradually, but steadily. The key to this seems to be a good set of existing, similar code they may refer to, combined with straightforward documentation explaining the relevant parts of the data model.

6. CONCLUSION

In practice, the Scopira has turned out to be very useful for our internal algorithms. Concepts introduced by Scopira, such as the data type tree and proponent system demonstrated their utility early in development. As expected, these concepts and ideas were slightly refined as more modules and maps were created with the tool. Input from user testing also contributed to several refinements. On the whole however, the original object-oriented-like design of the system internals as presented to module developers has proved itself to be a solid foundation on which to build more algorithms.

The interface, as presented to the module users, also worked very well. This came as no surprise as the concept was borrowed from several past tools used internally[4].

Future plans for development include implementing more algorithms, optimizing those that currently exist and tuning the parallel scheduling engine. Algorithms and data types from other domains would also be worthwhile to test.

7. ACKNOWLEDGMENTS

We would like to thank Brion Dolenko, Alexandre Nikulin, Mark Alexiuk and Marina Mandelzweig (all from the IBD) for their contributions to development. NSERC is gratefully acknowledged for their financial support of this work.

8. REFERENCES

[1] A. E. Nikulin, B. Dolenko, T. Bezabeh and R. L. Somorjai "Near-optimal Region Selection for Feature Space Reduction: Novel Preprocessing Methods of Classifying MR Spectra" *NMR in Biomedicine* 11 1-8 (1998).

[2] Bezdek J., Ehrlich R., Full W. "FCM: the fuzzy c-means clustering algorithm" *Comput Geo Sci* 10 191-203 (1984).

[3] A. Struyf, P. Rousseeuw "High-dimensional computation of deepest location" *Computational Statistics and Data Analysis* 34 415-426 (2000).

[4] A. B. Demko, N. J. Pizzi, R. L. Somorjai "A Classification Canvas for the Analysis of Biomedical Data" *IEEE CCECE, Toronto, Canada, May 13-16* 015 (2001).