A CLASSIFICATION CANVAS 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

ABSTRACT

With the rapid proliferation of complex high-dimensional biomedical data, an acute need exists for a comprehensive, knowledge-based, domain-specific, user-friendly software suite that allows investigators, in the health care disciplines, to classify their data through the detection of novel or discriminating features therein. The Classification Canvas is an attempt to achieve these goals in addition to providing intuitive visual computation and logic construction.

In this paper we describe various design and implementation issues such as: balancing user (novice) friendliness and developer (experienced) utility, performance versus modularity trade-offs, C++ and Java data sharing responsibilities, and creating graphical interfaces for (user-supplied) algorithm control.

1. INTRODUCTION

The Classification Canvas (CanClass) is a software tool and developer framework that allows investigators in the health care discipline to apply complex classification and other algorithms in various combinations, in a common, user-friendly environment.

As an algorithm builder, CanClass allows the placement of multiple algorithm modules (classifiers, pre- and post-processors) and their connections to each other to form more 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 is tightly typed in a hierarchical, object-oriented, data type tree. Configuration of module parameters is also performed visually, with immediate feedback.

As a development environment for experienced data analysts, new data types may be added to the type tree for automatic management by CanClass. Developers can quickly build graphical interfaces to their algorithms using CanClass' supplied *proponent* (property-aware components) graphic tool kit and simplified programming interface. CanClass is written in Java and C++ and offers a sophisticated data model and an extensible algorithm and proponent framework. Algorithms may be developed in Java or C++. Platform independence is maintained by strictly adhering to ANSI C++ for algorithms and Java for the rest of the application.

1.1. Application

CanClass 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. A genetic algorithm module was developed to implement near-optimal region selection for feature space reduction[1]. The genetic algorithm module connects to another module for fitness function evaluation. A linear discriminant analysis module was developed for classification. Fuzzy C-Means clustering[2] and half-space median[3] were implemented as modules, and validated for correctness against their original implementations. Finally, a set of general modules were 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), the Classification Canvas focusses on giving the user maximum control over the module system while still retaining a simple and consistent interface.

2.1. Maps and Modules

A CanClass 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 CanClass 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 in each input slot.

Maps not only contain modules, but may be treated as modules themselves. This allows the user to build and use compound modules from other modules and nest them within new maps. By packaging modules into new compound modules, the user is able to make more complex maps with little increase in apparent complexity.

2.2. Visual Display

To assist the user in map construction, CanClass presents the map visually to the user. Modules are represented as titled blocks, 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. 2).

2.3. Data Type Tree

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

Internally, CanClass 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*).

CanClass 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.

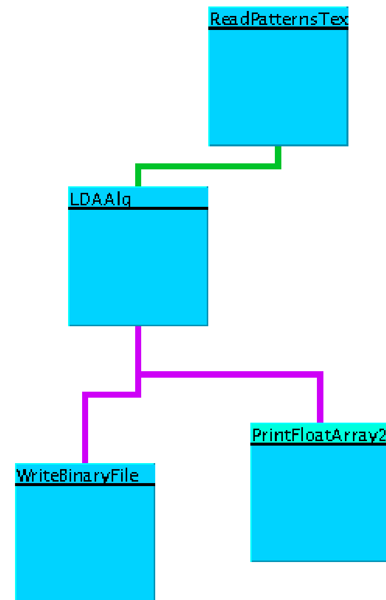

Fig. 1. Connected modules example

3. DEVELOPMENT FRAMEWORK

CanClass allows developers to extend the framework along three major areas; compound modules, types and proponents. By reusing generic components that already exist within CanClass, developers may focus on their application specific requirements.

3.1. Modules

A module in CanClass is defined as an algorithm that has zero or more data slots. With these slots, modules exchange data with other modules within the map (Fig. 3). However, everything except the module-specific core algorithms are common to all modules and naturally are abstracted from the module developer and handled by CanClass.

This abstraction lets the module developer focus on writing the algorithm specific code in their module. He uses several CanClass methods to extract input data and post output data. CanClass ensures that input slots marked as *required* have data before calling the core algorithm.

All CanClass data types are implemented as either Java interfaces or base classes. This common object-oriented practice allows the actual data type internals to be hidden from the module developer and permits polymorphism. Mod-

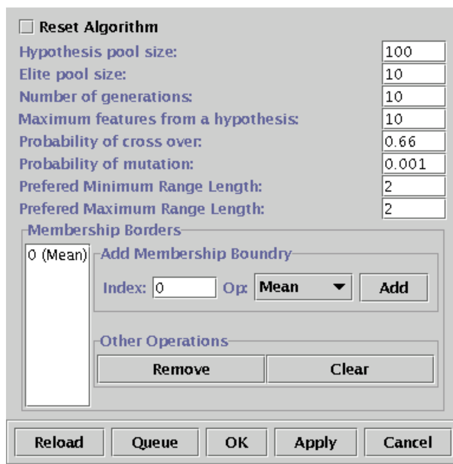

Fig. 2. Proponents built configuration screen example

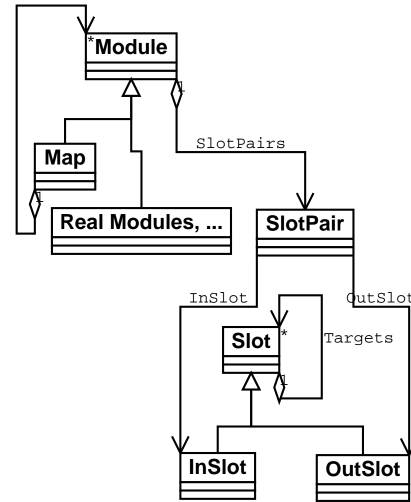

Fig. 3. Simplified class inheritance and ownership diagram of modules and maps

ule developers operate on their input and output data types via the prescribed methods for their respective interfaces.

CanClass comes with a bundle of useful and generic modules. This includes modules that load/save data in various formats (for example, in text for portability or binary for speed), 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 standard statistical functions. All these modules operate on the bundled data types and user-defined descendants.

In practice, it was determined that for computationally bound, numerical algorithms, Java proved to be too much of a bottleneck. Even if native code compilation was assumed, array access bounds checking, pointer access checking and lack of function in-lining proved to be, at times, several times slower when compared to an equivalent C++ implementation[4].

CanClass of course, does not restrict the module writer from calling *native* routines from their algorithms. Native routines are simply Java methods that are implemented in a native operating system library (often compiled from C or C++ source). The Java Native Interface (JNI) is a library binding standard, and C/C++ API that specifies how the virtual machine and JNI library files interact. In practice, we found the need to convert many of our numerical algorithms to C++ in order to realize acceptable module performance.

3.2. Types

CanClass structures data types as a tree, where each node of this tree represents one data type (Fig. 4). All data types are implemented as either Java interfaces or classes. Module writers add new types by extending an existing data type or implementing one of the defined interfaces.

Early in the development cycle, it was clear that the data type tree must support multiple inheritance. For example, a two-dimensional array of floats is both a descendant of the two-dimensional real-number array class and of the one-dimensional real-number vector class. In the interest of code reuse, however, the two-dimensional float array class inherits from the one-dimensional float vector class, which inherits only from the one-dimensional real-number array class. Since certain modules may need to accept the more generic two-dimensional real-number array class, the two-dimensional float array class must also inherit from that.

Module writers then need to *register* their type with CanClass by specifying where in the CanClass data type tree their type should be added. CanClass will then insure connection-time type checking, based on its location in the type tree.

3.3. *Proponents*

Proponents (“property aware components”) are visual, user interface components that are designed to control the properties of a module. They allow for objects to set or get their *current value* (which must be of a certain CanClass data type or any descendant of that type).

In Java, a proponent is defined as an interface with set and get methods, that operate on the base data type. We define it as an interface (and not as a base class), because all proponents will have to extend from an existing visual component class, but not necessarily the same one. Since Java does not have multiple inheritance, we define a proponent as an interface, since the visual components already have base classes.

CanClass includes set of generic proponents that operate

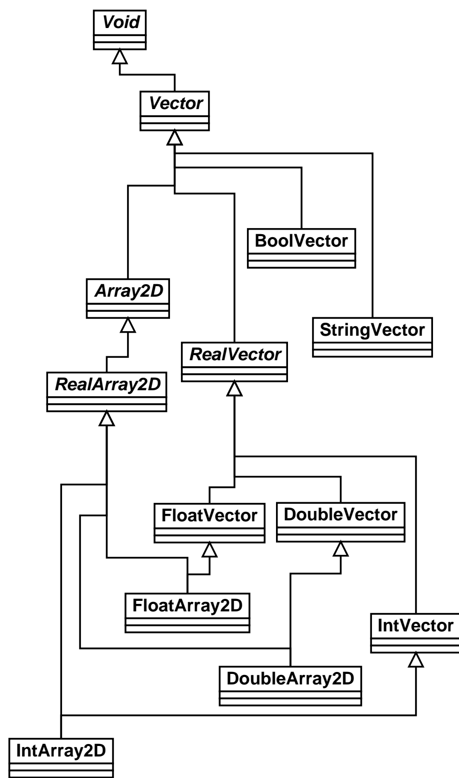

Fig. 4. Simplified CanClass module data type class hierarchy

on the bundled data types (Fig. 5). Module writers are free to make their own proponents based on these data types, either by extending an existing proponent or by building one from scratch.

If a module writer builds one from the ground up, he must implement the proponent interface, and if need be, convert the data type to a form needed by the visual component. Because proponents are also valid visual components, module writers may use them to compose larger, more complex proponents that operate on composite or simply more complex data types.

If a module writer introduces a new data type, he must also make at least one new proponent for it. If he does not, then that data type may not be used as a property of a module. In practice however, it was found that most new data types tended to be used for inter-module data transfer and not as properties.

Proponent interface complexity is limited only by the Java visual component toolkit (Swing). This gives the module developer full power to make radically new and possibly complex interfaces as required.

Because property slots are still regular slots at their core, they may be connected to slots in other modules in the usual manner. This allows controller-like modules to be built. These modules control the properties of one or more other modules, by taking data from a file or by building a unified property dialog screen from proponents and presenting them to the user.

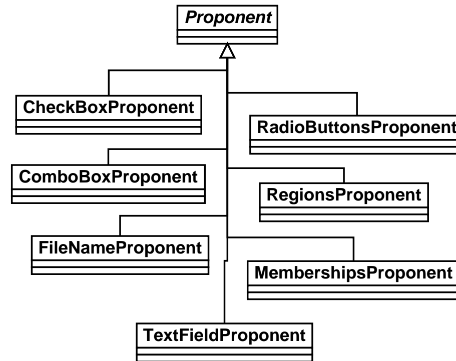

Fig. 5. Simplified proponent class hierarchy

4. DISCUSSION

CanClass attempts to balance the needs of the module user with those of the module developer. As a result, several design trade-offs had to be made.

4.1. User Interface

The concept of modules connecting their input and output slots to other modules on a map was taken from various internal applications. This scheme worked quite well in practice with most algorithm requirements. It was decided to keep this interface for CanClass. The interface presented to the user directly maps to objects managed by the system. This mapping allows the user to control and manipulate the map and modules in their natural form. Furthermore, we have concluded that the end user must be given more control over the firing algorithm. The firing algorithm controls modules execution, data propagation, and data caching. As maps get more complex, it was found that for every automated firing algorithm devised, a sample map may be found such that execution of the map would not run as expected under that algorithm. Exposing the user to the firing algorithm adds a little more complexity for the user, but in the end, it was required to be able to get the desired execution for all maps.

Proponents benefit both the map user and the module developer. The map users get used to seeing the same, consistent set of visual components in many modules. Module developers get a set of visual components that they may use to quickly assemble configuration screens for their modules.

Overall, the graphical interface presented to the user is functional and consistent. Module-dependent property configuration screens are made as consistent as possible with

proponents. All this was done with little or no impact on the developer's view of the system.

4.2. Overall Design

After deciding on the core class layout of maps, modules and slots, the rest of the system had to be designed. Based on this foundation, the developer framework had to be designed to promote code and design reuse.

Because of the popularity and success of object-oriented programming, and because the development of CanClass was done in an object-oriented language, it was decided that the CanClass' extensible areas should adhere to an object-oriented design.

The data type tree and base proponent collection are both in an object-oriented hierarchy. All the typical benefits of object-oriented programming; encapsulation, polymorphism and inheritance, are available to the module developer.

Decoupling the module configuration screens from the modules themselves was also deemed beneficial. This was done using proponents, and allowed reusing common visual components with many modules. The benefit was evidenced with the development of the first few modules that required the same basic proponents (edit box, file selector, etc).

Finally, all the common module functions are encapsulated in the common module class. Module developers are thus hidden from the details of data transport, allowing for different implementations in the future. For example, modules may be spread over many machines in a computational cluster, or they may all be on the same machine - neither methods require code change by the module writer.

4.3. Performance

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 CanClass' 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 CanClass 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 CanClass spends moving data between 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.

Some modules may be bottlenecks for a map. These may be modules that simply operate on large data sets, operate frequently (because of a loop) or simply implement a computationally intensive algorithm.

The Java language - even if we assume efficient run time byte code compilation - has some properties that make it ill-suited for such computationally bound modules[4]. Array access bounds checking, pointer access checking and lack of function in-lining may, at times, make a Java algorithm several times slower than the same algorithm implemented in C++.

In an attempt to reduce the Java performance hit in these modules, they should be implemented in native, standard C++ code. There is some inconvenience to the module developer as he has to do JNI marshaling between Java and C++, but after this common code bridge is set up, algorithm development proceeds normally, in C++.

The inter-module data-passing system of CanClass is implemented in Java. Therefore, if two separate, C++ computationally intensive modules are in a tight loop, all their data communication still goes to Java for transport. This may prove to be another bottleneck in a map, especially if the communication time to module execution time ratio is high.

There are several ways to deal with the performance hit if Java-based inter-module communication time becomes a bottleneck. One is to redo the inter-module communication using C++ and relegate Java only to the graphical user interface. Another is to perhaps introduce the concept of *micro-functions*, that may be thought of as very small modules that do not go through the normal slot connections and data type checking system as normal modules do.

5. CONCLUSION

In practice, the Classification Canvas has turned out to be very useful for our internal algorithms. Concepts introduced by CanClass, 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.

CanClass refinements are always ongoing. Two major areas of note however, are using proponents for visual data exploration and using computer clusters for parallel computation.

Proponents introduced the idea of data-aware visual components that may be separate from the modules and their associated slots. This concept may be expanded to include proponents that are more interactive with the host module. Proponents may then be used to give live output of the state of the module and of data as it passes through it. This would allow for more graphical representation of data, aiding users in their investigations.

Finally, modules in the map may be spread over a cluster of workstations, for instance a Linux Beowulf cluster. This would allow users to take advantage of inexpensive but powerful parallelization for their computationally intensive modules. Module developers would still have the same programming interface, and would not have to adjust their development in any way to accommodate the network. CanClass would be able to automatically move modules to various machines through the network to perform load balancing and parallel processing with users controlling this process as needed.

6. ACKNOWLEDGMENTS

We would like to thank Brion Dolenko of the Institute for Biodiagnostics (IBD) for sharing expertise with existing, internal classification systems and Alexandre Nikulin of IBD for his assistance with the implementation of the genetic algorithm module.

The National Sciences and Engineering Research Council of Canada (NSERC) is gratefully acknowledged for their financial support of this work.

7. REFERENCES

[1] Alexander E. Nikulin, Brion Dolenko, Tedros Bezabeh and Ray 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] Pizzi N., Vivanco R., Somorjai R. "EvIdent(tm): a Java-based fMRI data analysis application" *Proc SPIE Vol 3808, Applics Digital Image Proc XXII, Denver, USA* 761-770 (1999).