Programming Techniques – Project Report

Candidate # 81222 (Ken Webb)
January 6, 2004

Specification

The purpose of this project is take an existing object-oriented (OO) system (Webb & White, 2003) written using a proprietary software development system, Rational Rose RealTime (RRT), and convert key parts of it to generic C and C++. The system is an executable model of a biological cell. It contains an inheritance hierarchy (see Figure 1), as well as a containment hierarchy (see Figure 2).

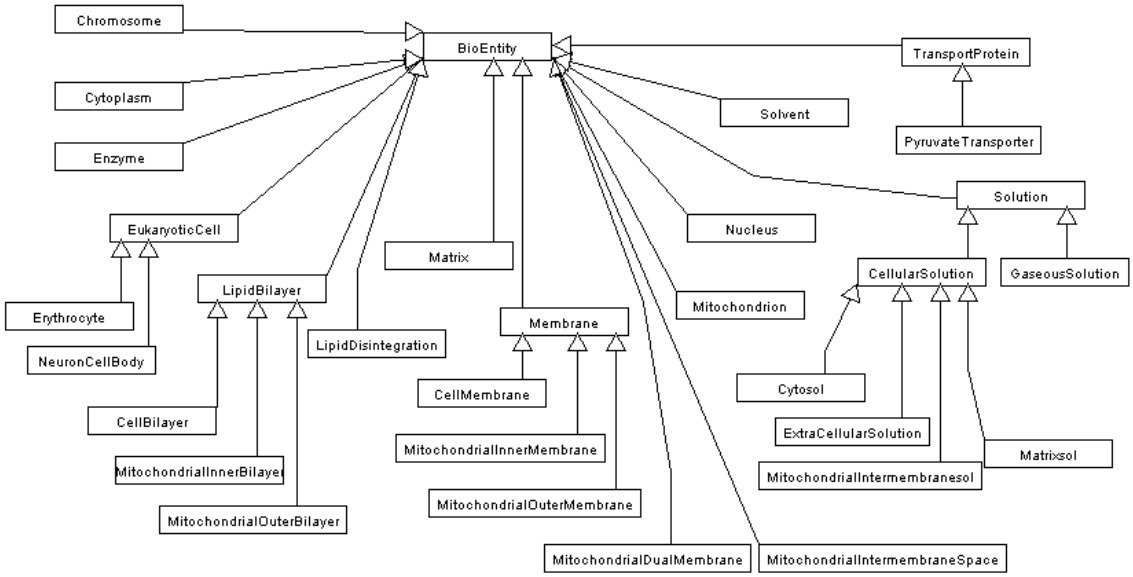

Figure 1 - Inheritance hierarchy of the system being modeled.

Both of these hierarchies are currently specified using the Unified Modeling Language (UML), and can readily be represented as trees. UML and RRT also allow associations between objects at arbitrary levels in the containment hierarchy tree, a feature heavily used in the biological cell application. The resulting structure is graph-like, although the containment hierarchy links and the lateral association links are of two quite different types. Where nodes have lateral connections, the links are directional. Active object nodes point to and are able to operate on one or more passive nodes. The relationship between active and passive nodes is many to many. The structure of lateral connections is to be determined at run-time. During initial configuration, and whenever the containment structure changes, each active node will traverse the containment tree starting at its own local node, and will search for passive node(s) with which it can meaningfully interact.

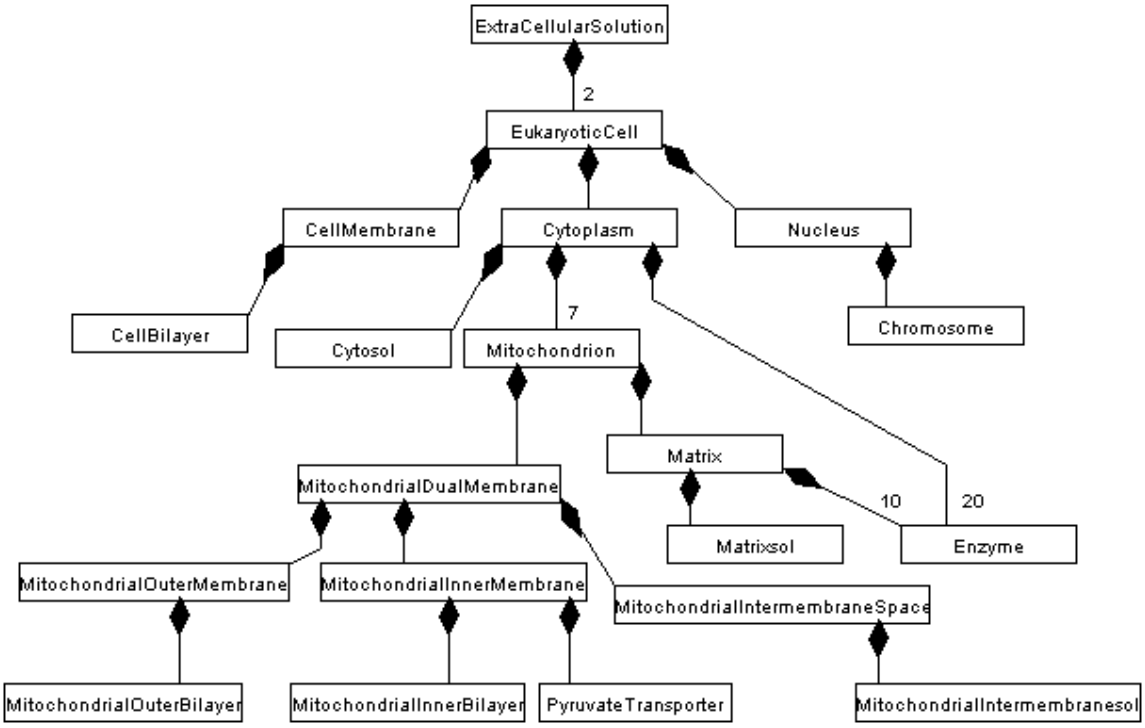

Figure 2 - Containment hierarchy of the system being modeled.

The set of programs reported on in the paper are based on a tree structure. The abstract data structure should support two types of concrete tree node objects: BioEntity objects, and BioEntityClass objects.

The data structure must be manipulable using the Genetic Programming (GP) technique (Koza, 1992). GP makes use of binary trees, in which two arbitrary subtree nodes can be swapped (genetic cross-over), or one arbitrary subtree node can be completely replaced by something new (mutation).

Nodes within a tree must be able to navigate the tree of which they are a part. They will for example need access to their parent, their siblings, their children, and to the application data stored within each node as they traverse the tree. Each node within a tree is an agent with effectively its own thread of control. This contrasts with the traditional approach in which some outside agency navigates the tree.

This system is an initial proof-of-concept prototype to demonstrate that the basic architecture described in Webb & White (2003) can be implemented in a straight-forward manner using C and C++. Many more advanced features have been left out of this prototype, including:

- There is no GUI, just a console and file interface. A GUI for such a simple system would result in reduced portability for little benefit at this point.

Design

The system is implemented using C++ classes, with a main program in C. It was developed using Microsoft Visual C++ 6.0, and the same code has also been successfully compiled and run using the Mingw version of the GCC compiler under the control of the Dev-C++ IDE (version 4.9.8.0). Both resulting executables produce the same output when run as Windows console applications. Advanced features of C and C++ that might interfere with portability, such as C++ templates and the Standard Template Library (STL), C++ streams, or any library function not found in Kernighan and Ritchie (1988), have not been used. Basic C++ class features such as inheritance, object instantiation, constructors, destructors, new (rather than the various versions of alloc), and delete, have however been used.

The resulting system is intentionally very C-like. ANSI C would be a good choice as the implementation language for performance reasons, because of its high portability, and because C functions can be called by applications written in C, C++, Java and a host of other languages. C++ was chosen instead because of the benefits to the programmer of using object-oriented concepts, a powerful paradigm for designing system architecture. But I also believe in retaining the sparseness and simplicity of C. Because the implementation is so C-like, it would be relatively straight forward to convert it to C.

Performance is very important for this application. Eventually it will have up to a million object instances, many of which will be actively running some bit of code each time step. The C++ STL does not provide a tree container, but I did try an STL-based tree implementation (Peeters, 2003). I was not happy (subjectively) with the performance it gave me, and I felt it would not give me the flexibility I need for something that is essentially basic research. Java would be an inappropriate language to use for this application for performance reasons, although a Java GUI would be possible using the Java native interface.

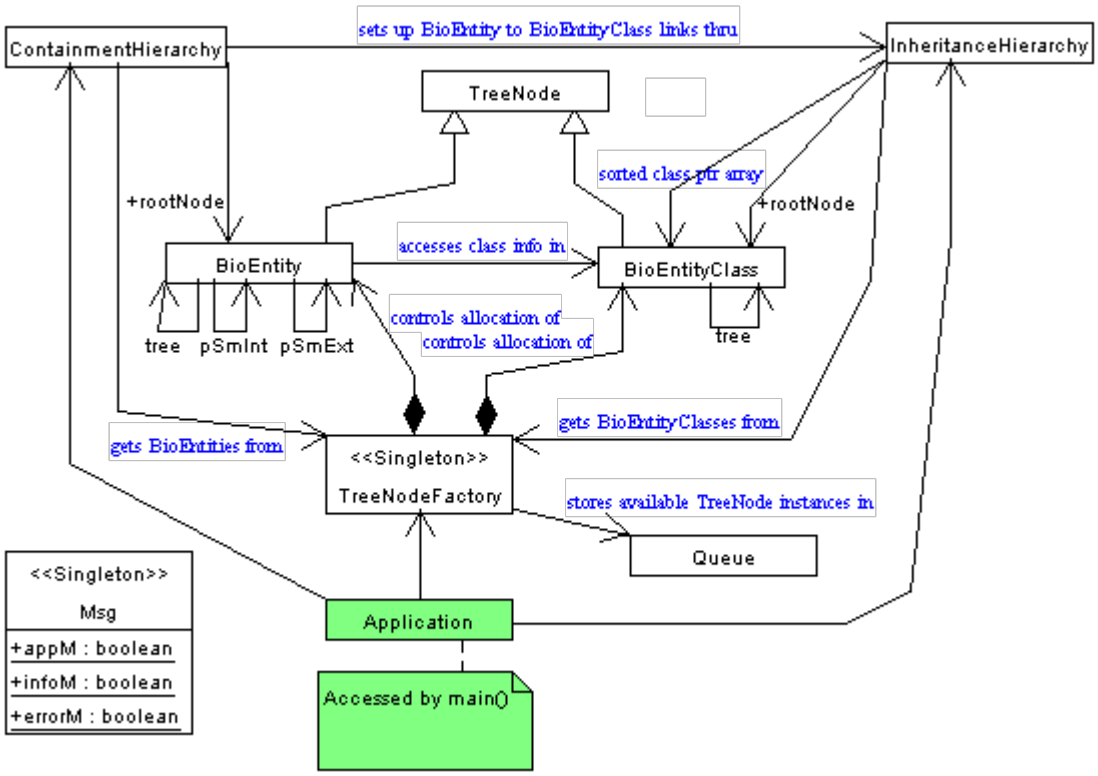

Figure 3 - UML class diagram showing the architecture of the system described in this paper.

The overall structure of the system is as shown in Figure 3, a UML class diagram. An application creates single instances of TreeNodeFactory, InheritanceHierarchy, and ContainmentHierarchy. InheritanceHierarchy reads the names and IDs of all possible BioEntityClass nodes from a file, and then reads a second file to create a tree of BioEntityClass nodes with a root node. ContainmentHierarchy reads another file to create a tree of BioEntity nodes with its own root node. Each BioEntity node points to a BioEntityClass node from which it accesses its class information. A BioEntity node may also have cross links to other BioEntity nodes (pSmInt and pSmExt) representing interactions with adjacent internal and external entities. BioEntity and BioEntityClass are both concrete subclasses of the abstract TreeNode class. TreeNodeFactory statically (for faster performance in applications that dynamically add and remove tree nodes) controls allocation of space for instances of both TreeNode subclasses using a set of two arrays, and stores available TreeNode instances in Queues. InheritanceHierarchy gets BioEntityClass nodes, and ContainmentHierarchy gets BioEntity nodes, from the TreeNodeFactory.

A number of data structures are used. TreeNode subclass instances are allocated in static arrays, stored in availability queues, and used as nodes in a set of two trees. Both trees are recursive structures. The tree data structure was chosen as appropriate for this system because of its similarity to the real world being modeled. Biological cells have a natural structure of

nested components contained within other components, and there are also many different types of closely-related entities that can be naturally modeled in a class inheritance hierarchy. This system is also an implementation of the more generic object and class paradigm, and uses trees to represent both of these.

The entire system is based on binary trees used to represent n-ary trees. Each tree (the InheritanceHierarchy tree and the ContainmentHierarchy tree) has a single root node. Each node in the tree has links to a parent node (except for the root node), an optional leftChild node, and an optional rightSibling node. A leftChild node represents the start of a new level in the hierarchy, while a rightSibling node is a node at the same level and with the same parent. Each node in a rightSibling chain has a pointer to the same parent. This tree structure is thus different from the more familiar binary search tree (Kernighan & Ritchie, 1988, p.139) in which left and right nodes are both at a level below that of the parent. Binary search trees have symmetrical semantics, while the binary trees described in this paper are asymmetrical.

BioEntity nodes also have cross links to other BioEntity nodes. In one sense this turns the tree into a graph. But it is still treated as a tree, because the standard tree links (parent, leftChild, and rightSibling) can be easily distinguished from the cross links (pSmInt and pSmExt). The application benefits from using the simpler tree structure, with enhancements that give it the added power of a graph representation. The cross links are established dynamically at run-time. Each BioEntity node follows a set of navigation and other instructions given to it by its BioEntityClass node. The instructions differ based on the class of BioEntity. BioEntities that are active objects are instructed to navigate the tree moving relative to their own local position in the tree. They can navigate up, down and right in the tree to find a passive BioEntity with which they can interact. The active BioEnities also have genes that tell them what to do during each time step. The passive BioEntities have phenomic data instead. Both genotype and phenotype information are given as part of the BioEntityClass-provided instructions.

An initial important question was whether BioEntity and BioEntityClass should be components of or subclasses of TreeNode. A general principle of good software design is to maximize cohesion and minimize coupling by having an entity only perform one function, and the usual practice in this case is to separate the treeness of a node from its contents by having the tree node contain the contents (Kernighan & Ritchie, 1988, p.139; Goodrich & Tamassia, 2001, p.233). I have chosen instead to use subclassing, with the result that each instance of the BioEntity subclass has both treeness and problem domain properties. I believe this more naturally captures the natural world that I am modeling, in which objects are physically contained within other objects. It is generally considered good programming

practice to have the computer system match as closely as possible the problem domain, in this case biology. Thus I have chosen a domain-specific approach rather than complete generality.

Another important good programming practice is to hide data and only make it accessible by calling publicly advertised interface functions. All of the variables in my C++ classes are private (or protected in the case of classes that may be subclassed), and can only be accessed using public set and get functions. This allows the class to control how and if its data is accessed, and makes it possible to change how its data structures and algorithms are implemented without effecting other classes. In at least one case (BioEntity::getName()) the get function constructs the data when called rather than keeping it as a redundant combination of two other pieces of data.

A corollary of the data hiding principle is not to use global variables. I have chosen to use global variables in one well-defined case. The Msg class consists entirely of three global variables that control the printing of messages during program execution. The intent here is to ease the job of the programmer, by not adding a lot of unnecessary overhead. In several cases I have used class variables, where some constant but initially-configurable value needs to be accessed by all instances of the class. This is common practice, and avoids the memory overhead of having each instance instantiate a copy of the same data. BioEntityClass uses this approach, by defining four variables as private and static, and prohibiting external access to these by not providing any get functions. Global constants are used in a number of cases to implement a common protocol between instances of different classes. For example, Queue uses `const int` to define a set of three global constants, so that instances of TreeNodeFactory can call instances of Queue and correctly interpret the returned code.

There is much flexibility in the system that will allow it to be extended in the future. This has been one of the goals of the current implementation. Additional tree functionality can be added to either BioEntity or BioEntityClass, or to the TreeNode superclass if it's something that both subclasses will need. There is a requirement for two other types of trees and tree nodes (one to construct complex behaviors from a tree of primitive actions, and a second to store a large variable number of different phene values within the same container) in an expanded version of this application. These can be subclassed from TreeNode. The concept of gene and phene makes it possible to use Genetic Algorithms to evolve the gene and phene integer values used in the system. The use of trees makes it possible to use something like Genetic Programming to evolve and improve the actual structure of the system, rather than just reading in a static structure from a file. A major future goal is to allow the architecture to evolve bottom-up rather than specifying it top-down. The use of tree-structures enables this.

User Manual

As a user, it is best to think of this system from a Model-View-Controller (MVC) perspective. The system itself is a simple proof-of-concept *model* of a biological cell and of an abstraction of interaction between entities within that cell. You can obtain a *view* of what's going on in the model by observing the output in the console window. You can *control* what goes on by manipulating a set of configuration files, and through command line parameters.

Model

The model has already been described at some length in the rest of this document.

Controller

The main configuration file is called ConfigFile.txt. Each line in the file consists of one or more values followed by the comment character # and a very brief identifying description. The first four variables, on the first and second lines, control what type of information the model will output to the view. Only the application data will be displayed if only the appM variable (the first one on the second line) is set to 1, and all others are set to 0. To confirm the values that the model is reading from this file, set the first variable (first line) to 1. To display error messages such as "file not found", set the fourth variable (errorM) to 1. To display a large amount of informative data, set the third variable (infoM) to 1.

MaxBioEntities must have a value greater than the number of BioEntity nodes that will be created. In the current implementation, this is the sum of the third column in the ContainmentHierarchy data file, which in the default file (CellContainData_2.txt) would have a value of 49 (1 + 1 + … + 1 + 10 + 20 + 1 + 1). MaxBioEntityClasses must have a value greater than the number of entries in the BioEntityClass names file, which in the default file (TypeData.h) is 427.

BevalGeneMin, BevalGeneMax, BevalPheneMin, and BevalPheneMax directly impact what you will observe in the view as the model progresses, if you have set the appM variable to 1. These are two sets of ranges, one for genes and the other for phenes. The two BevalGene variables can be any valid integer, but typical values are close to 0, such as the default values of -10 and +20. During each processing loop, each gene node (an enzyme or bilayer) increments the phene value of one or two other nodes by a constant random amount between BevalGeneMin and BevalGeneMax. The two BevalPhene values can be any valid integer greater than or equal to 0, with typical values being at least 10000, such as the default values of 10000 and 100000. Max values must be greater than or equal to Min values. If a phene value becomes greater than LONG_MAX (+2147483647L for many C++ compilers) during the main processing loop, it will wrap around to LONG_MIN (a very large negative number).

maxProcessLoops controls the number of processing loops or time steps that the application will put each BioEntity node through. If you have set the appM variable to 1, you will observe maxProcessLoops lines of data in the view. This can be any valid integer from 0 to LONG_MAX (+2147483647L for many C++ compilers). Typical values would be between 10 and 1000, enough iterations to indicate that the main processing loop is working, and that each phene value is continuously increasing or decreasing.

The BioEntityClass names file (default: TypeData.h) is a C header file with an enumerated list of several hundred potential BioEntityClass node IDs. Great care should be exercised if you choose to modify or replace this file. The compiler uses it as a header file, and the executing application reads it as a data file. The two other configuration files use the same enumerated values.

The InheritanceHierarchy data file (default: CellData_2.txt) specifies the inheritance relationships between BioEntityClass nodes. Each line in the file contains a level in the inheritance hierarchy, and the ID of a BioEntityClass.

The ContainmentHierarchy data file (default: CellContainData_2.txt) specifies the containment relationships between BioEntity nodes. Each line in the file contains a level in the containment hierarchy, the ID of the BioEntityClass that it is an instance of, and the number of such nodes that should be created in the BioEntity.

View

The view, displayed in the console window, will differ depending on the parameter values specified in the configuration files. If you have chosen to view application data (appM = 1), you will see the names of four solution spaces found within and surrounding a biological cell on the first line, and the changing values associated with each of these spaces as processing advances in time on subsequent lines. If you are using a Windows computer, you can double-click go.bat (or run it from a console window) to run the application and view the resulting columns of data in an Excel spreadsheet. You might then want to select the Excel chart wizard to generate a line graph of the results. The data in each of the four columns will be observed to be linear.

If you have chosen to view additional informative data (infoM = 1), you will see the names of all BioEntityClass nodes, the InheritanceHierarchy tree with the level number shown as a sequence of dots, the ContainmentHierarchy tree shown in the same way, and the configuration data given to each BioEntity node by its BioEntityClass node.

Testing

Extensive unit testing of program correctness has been done for all classes. All test code is included in the TreeNodePlus_Main_Test1.cpp source file. I developed the test code in parallel with the development of each class. Before integrating classes together, as much as possible I confirmed that each class worked on its own. The test cases call each function in the class that they are testing, using typical values. At least 50% of the time spent on the system was spent developing and running these test cases.

Testing has so far not been completely comprehensive. It is common practice in the industry to have a separate test team whose job it is to prepare and perform comprehensive testing, because it is generally recognized that developers are not sufficiently motivated to find bugs in their own code. If this were professional software, I would do some additional unit testing, especially of boundary conditions and of errors that could occur while reading in the configuration files. I would then pass it on to the test team.

Another form of testing was running the application under the various parameters included in the configuration file (ConfigFile.txt) to confirm that the resulting values correspond with expectations.

Some quick performance testing shows that 20,000 active objects (enzymes) all running their own bit of code, can run through about 667 time steps per second on a 2.4GHz Celeron Dell laptop, for a total of about 13,300,000 enzyme operations per second (i.e. calls to the BioEntity::act() function).

Although no use of a profiling tool has been made, it has been generally confirmed that there are no serious memory leaks. This was done by observing the amount of memory currently in use as reported by the Windows Task Manager.

Conclusion

Using generic C++, good programming practices, and ideas inspired by biology, I have reproduced the major conceptual elements described in Webb & White (2003) which was based on the proprietary Rational Rose RealTime. These elements include a concept of object (BioEntity), a concept of class (BioEntityClass), inheritance and containment hierarchies, a concept of adjacency implemented by allowing each node to navigate the tree to find other entities with which it can interact, and the ability for active objects to act on passive objects over a series of discrete time steps. The output data produced by the program is quite uninteresting, and is there just to demonstrate that the architecture works. What is interesting is the flexible tree-based architecture that was used to produce the data. I believe this provides me with a good basis for moving forward to a more complete non-proprietary implementation.

References

[1] Goodrich, M. and Tamassia, R. *Data Structures and Algorithms in Java, 2nd ed*. New York: John Wiley, 2001.

[2] Kernighan, B. and Ritchie, D. The *C Programming Language, 2nd ed*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[3] Koza, J. *Genetic Programming – on the programming of computers by means of natural selction*. Cambridge, MA: The MIT Press, 1992.

[4] Peeters, K. *tree.hh library for C++*. 2003. http://www.damtp.cam.ac.uk/user/kp229/tree/

[5] Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. *Numerical Recipes in C – the art of scientific computing, 2nd ed*. Cambridge: Cambridge Univ. Press, 1992.

[6] [1] Webb, K., and White, T. UML as a Cell and Biochemistry Modeling Language. Carleton University Cognitive Science Technical Report 2003-05. http://www.carleton.ca/iis/TechReports