The Visualization of Linear Algebra Algorithms in Apt Apprentice

Christopher Andrews, Rodney Cooper, and Ghislain Deslongchamps,
and Olivier Spet

University of New Brunswick, P.O. Box 4400 Fredericton NB E3B 5A3, Canada
c.andrews@unb.ca,
<http://www.cs.unb.ca/research-groups/mmsdt/>

Abstract. The development of tools that can increase the productivity of computational chemists is of paramount importance to the pharmaceutical industry. Reducing the cost of drug research benefits consumer and company alike. Apt Apprentice is a visual programming paradigm designed to reduce the overhead associated with creating software to implement algorithms in the data analysis phase of rational drug design. It draws on both standard programming language environments and programming by demonstration. The approach of Apt Apprentice and an example of its use in implementing a linear algebra routine are described.

1 Introduction

Chemical modelling *in silico* is essential to rational drug design. Quantum mechanics, molecular mechanics, and Newtonian dynamics are first used to analyze the chemical structure of compounds found in a training set of molecules all of which exhibit a specific biological activity to some degree. The structurally derived descriptions of these compounds are then statistically analyzed to find predictors of the specific biological activity exhibited by elements of the training set. The subject area of this type of research is called Quantitative Structure Activity Relationships or QSAR. If a predictive relationship between structure and activity can be discovered, other compounds with known structural properties can be analyzed to determine their likelihood of exhibiting specific biological activity. Testing of any promising compounds can then proceed to the laboratory. These compounds are called *drug leads*.

Testing an eventual drug can cost upwards of one billion dollars and take up to 15 years to complete. Since not all promising compounds will pass the tests required by regulatory agencies risk avoidance is paramount.

It is well understood in the pharmaceutical industry that chemical modelling must be done by chemists as the modelling algorithms are not foolproof and can easily be wrongly applied. Apt Apprentice is a visual programming tool designed to facilitate algorithm development in chemical modelling software. It is hoped that this will lead to faster implementation of new algorithms in existing software, aid in the training of chemists to perform this task and facilitate modifications to software in the light of chemical knowledge.

2 The Apt Apprentice Model

The next section describes some of the unique characteristics of Apt Apprentice along with an example of its use.

2.1 Fundamental Concept

At the University of New Brunswick we teach Biochemistry majors computer assisted drug design. Our chemical modelling tool of choice is the Molecular Operating Environment, MOE™, produced by the Chemical Computing Group in Montreal, Canada. MOE is based on the Scientific Vector Language which is a *Collection Oriented Language*. Collection Oriented Languages are programming languages that operate on aggregate structures rather than single elements. Such aggregates can contain many atomic elements and generally allow for parallel processing. APL is one early implementation of this type of language which allows operations on aggregates themselves such as summing, multiplying, and reversing their elements and implements functionality like *apply to each* allowing for operations on each element of a collection. Since MOE permits the insertion of modules designed by the user in SVL it is essential our students are provided with tools that can reduce the challenge learning numerical algorithms and SVL have for non-programmers. While text-based languages may be suitable for regular programmers, they do not necessarily provide an adequate solution for non-programmers. One of the largest hurdles for non-programmers is learning language syntax. Non-programmers do not want to concern themselves with semicolons, comma placement, or for that matter, any other issue related to syntax. In addition, a text-based language mentally places the programmer into "editor" mode, rather than "creator" mode. This means that programmers are often more concerned with how an algorithm is implemented, rather than simply focusing on what the computer needs to do. This situation is akin to an author who writes with a computer, rather than with pen and paper. Using the computer, the author switches into "editor" mode, hindering creativity with its logical structure and as a result, causes a decrease in productivity. That is, the writer spends time trying to do things like craft perfect sentences, instead of concentrating on getting the main ideas out. Of course, it is reasonable to assume that a professional programmer will want to consider all aspects of an implementation, but in the research community, it is often better to get something up-and-running as soon as possible. In lead development, it is the *result* of the computation that is generally of most interest.

The paradigm that *Apt Apprentice* uses is designed to address these issues. It employs a direct manipulation interface, drawing on techniques from both text-based programming and programming by demonstration. The semantics of the environment attempt to mirror those found in a standard text editor, but also leverage the graphical interface to remove many syntax-related problems and facilitate program construction. It uses ideas from programming by demonstration to reduce the more technical aspects of algorithm implementation, allowing the programmer to concentrate on more abstract details. The environment makes

use of XML to produce a language independent description of the graphically inputted algorithm. XML serves as the input to an interpreter, which translates the XML into a standard text-based language in this case SVL (see figure. 1). XML is used because of its extensibility and its self-descriptive nature. Interpreters can be built that will not be broken by future additions to the XML schema. The programming environment can be improved while maintaining backwards compatibility.

Fig. 1. XML view of the Apt Apprentice Architecture

Apt Apprentice, unlike many other end-user programming environments, is intended to serve as a procedure level programming tool. As such, it assumes a more sophisticated end-user than many of its counterparts, and therefore does not attempt to remove all aspects of programming from the end-user. By allowing the programmer to partake in the programming process, Apt Apprentice is able to offer the power of a standard text-based language, while at the same time

using the graphical interface to keep this process at a higher level of abstraction. However, the design of such an environment requires the solution to its own set of problems.

2.2 Program Design

The first problem to be addressed in the design of Apt Apprentice is the issue of program flow, which is simply the order in which program statements are executed. In a text-based system, one expects a compiler (or person) to read a source code listing from top to bottom, moving from left to right. This is the natural way to read any document written using the spoken language of programmers¹. Unfortunately, as direct manipulation systems are based in graphical environments, this natural restriction on program flow is often lost. Many attempts have been made to indicate program flow using different symbols connected together by arrows in some form of constructed diagram, but the syntax and semantics of these are more learned than they are natural. The largest obstacles are: where to begin and end the path through the diagram, and in some cases, which arrow to take at any given time. Apt Apprentice overcomes this limitation by introducing the concept of *rows*. The idea is straightforward; based on the contents of a given row, the program either progresses to the right within the row, or drops down to the row directly below (see figure 2). This serves to promote the same natural flow that is found in standard source code. One simply starts at the first row, traverses that row according to the proper rules, then proceeds on to the next row.

Fig. 2. Elements of Apt Apprentice depicted to show three complete rows of the determinant algorithm, including an explicit loop

Of course, once the concept of flow is introduced, it must be controlled. In most text-based systems this is accomplished through the use of *if-else* statements. This is where the difficulty level rises for the novice programmer in a

¹ Assuming a European based language is used.

text-based system, since the program statements are no longer executed in a strictly top-to-bottom order. The implementation of conditional statements has also proved to be a difficult task in direct manipulation interfaces, with many of them omitting them entirely [9]. Within Apt Apprentice, it is *conditionals* that regulate the rightward flow within a particular row. Conditionals are represented as right pointing arrows. In other words, as long as the conditionals evaluate to a value of true, flow continues on to the right. A conditional that evaluates to false causes execution to drop down to the next row. To supply the notion of an *else* statement there needs to be a plausible state between not moving right and dropping down to the next row. This is accomplished through the use of a *sub-row* notation. A sub-row is a further subdivision of a standard row. These divisions are color coded as a means to differentiate them. They are also subject to the same rules as a standard row. That is, the algorithm continues to the right if the conditional evaluates to true. This criterion will ensure that at any given time only a single sub-row can be completely traversed to the right. This notation is similar to a *case* statement in a traditional text-based language. The flow of control in the program is thus readily apparent to anyone viewing the algorithm, and does not change even when the nesting of conditionals occurs. This idea can be seen in figure 2.

In the development of Apt Apprentice the focus to this point has been primarily in the area of Linear Algebra. The use of matrices is often a requirement in implementing statistical procedures for QSAR. This has constrained the exploration of various notations to a limited subject matter, while still maintaining the relevance to computational chemistry.

To illustrate various features the example of calculating the determinant of a matrix using Gaussian elimination will be used. This algorithm can be found in any standard textbook on Linear Algebra. The user begins describing the algorithm by entering an example matrix, say: $[[1,2,3],[4,5,6],[7,8,8]]$. The first step of the algorithm involves examining the element in position (1,1) to determine that it is non-zero. If it is non-zero the element becomes the pivot element for this iteration. If it is zero an element below must satisfy the condition; or execution stops. The process is repeated for each pivot element found on the main diagonal of the matrix. Row 1 of figure 2 is Apt Apprentice's graphical representation of this process, with the exception of the stopping condition when the final pivot element is reached. To tell Apt Apprentice the sequence of pivots the programmer simply clicks on each element on the main diagonal. This guides Apt Apprentice in choosing the next pivot in each iteration. The idea of a guide is not new, and has been used in several implementations of programming by demonstration [9]. To our knowledge however, it has not been used before to depict a sequence of elements operated on by an algorithm. The *pivot* element is defined by the user and given a name. The system is apprised of the conditions regarding this element (i.e. that it not be equal to zero). After defining the guide, the system generates an arrow pointing to the right, which represents the situation when the pivot is equal to zero. The algorithm traverses to the right if and only if the pivot element is zero, otherwise either the next sub-row is

chosen, or if there are no more sub-rows, the next main row. If the pivot is zero, a non-zero element must be found below in the same column. This is achieved in Apt Apprentice through the *scan* operation. Scanning may be performed in many different directions, but in this example we confine ourselves to downwards only. According to the algorithm, the scan result must also be non-zero. The system automatically adds a new right arrow to direct the algorithm if the scan is successful. The final step, once a non-zero element has been found, is to swap the original pivot row with the one uncovered by the scan. The *swap* operation is used to perform this task. The operation is automatically associated with the proceeding *scan*, and the user only need specify the other row involved in the swap. Dialog boxes are used to allow the user to select the appropriate item. The use of dialog boxes to aid in programming by demonstration systems has been well documented [9],[6],[2]. Finally, the system realizes that there is still one condition unspecified, that is, what should the system do if the *scan* is unsuccessful. The second sub-row of row 1 gives the proper course of action. For this condition to be evaluated, the pivot must be equal to zero; thus the system automatically replicates this portion of the upper sub-row. The user is then only required to show the system how to manage the case when the entire column is zeroes (i.e. there is no row available for a swap). In this case, the user indicates that the algorithm has finished through the use of the *exit* icon. The exit icon behaves like a standard *return* statement, allowing the user to indicate the value to produce at this point. The rightward *value* arrow is notational in nature and only serves to remind the user to progress to the right within the row. The user then provides the exit value, which in this example is simply the value contained in the pivot variable. Upon completion of the first row the pivot elements of the matrix have been defined and the algorithm has been told how to handle exceptional values. In essence, a nested conditional statement has been created.

To continue on to row 2 the pivot element must be non-zero. In other words, at least one conditional arrow in each sub-row of the proceeding row has evaluated to false. Row 2 describes the process of eliminating the remaining elements in the pivot column. According to the standard algorithm this is carried out through elementary row operations. In Apt Apprentice a *scan* operation is first used to find an element in the pivot column that is non-zero, and also not located at the position of pivot. Every variable has a position associated with it by default. Positions may be compared with operators such as *@* and *not @*. If an element is found that matches this criteria, the algorithm continues to the right.

At this point Apt Apprentice must be shown how to carry out the required calculations. The notation used for this operation is a different style arrow which again points to the right. The user demonstrates the calculation to Apt Apprentice using appropriate drag and drop operations. The user shows how the element found by the *scan* is reduced to zero using an elementary row operation. A series of dialog boxes is used to connect the proper rows to the position of the pivot element and the scan result.

In row 3, a loop is introduced using a *return* operation. A *return to* works in exactly the way one would expect. The icon is placed on a row; then the

row that the algorithm is to *return to* is selected using the mouse. This is the familiar **GOTO**. Forward **GOTOs** are not allowed. The responsibility falls to Apt Apprentice to generate code that avoids the usual pitfalls of using a **GOTO** statement.

To begin, a scan is performed to determine if there are any remaining non-zero elements in the column (excluding the pivot value). If there are non-zero elements, the algorithm returns to row 2. At the point when the scan is unsuccessful, the condition associated with the arrow evaluates to false and algorithm execution moves on to row 4. The remainder of the algorithm can be seen in figure 3. Row 4 defines the *finalpivot* position and a condition that *pivot* is not @ *finalpivot*. If this is the case, then the algorithm has not visited every pivot element, and so, should return to row 1. Once the *finalpivot* has been reached execution drops down to row 5. Here Apt Apprentice is once again directed to exit, however this time the exit value demonstrated is the product of the diagonal elements. Apt Apprentice maintains properties associated with specific operations such as the number of times the swap is called. Swapping rows of a determinant changes its sign so the pivot value is multiplied by (-1) raised to the power of the swapcount value. The result of this expression is the determinant of this particular matrix.

3 Conclusions

Apt Apprentice presents a hybrid method for end-user programming. It attempts to merge the feel of a standard text-based programming language with the ab-

Fig. 3. Depiction of Apt Apprentice showing the complete determinant algorithm

straction offered by direct manipulation environments. The ultimate goal of Apt Apprentice is to be a system that enables computational chemists to extend applications, without requiring the activation of a computer science team. Until now, the focus for the programming capabilities of Apt Apprentice has been on standard computer science algorithms directed towards classroom use. Apt Apprentice assumes a more sophisticated end-user than many of its peers and so does not attempt to hide all aspects of the programming task. In the future, it will be necessary to develop a larger repertoire of graphical constructs as well as submit the system to extensive user testing.

Acknowledgments

Student PGSB support provided by the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

References

1. Amant, R. St., Lieberman, H., Potter, R., and Zettlemoyer, L.: Programming by example: visual generalization in programming by example. *Comm. ACM.* **43** (2000) 107-114.
2. Bocionek, S. and Sassin, M.: Dialog-Based Learning (DBL) for Adaptive Interface Agents and Programming-by-Demonstration Systems. CMU-CS-93-175, Carnegie Mellon University (1993) 1-51.
3. Chang, S., Korfhage, R. R., Levialdi, S., and Ichikawa, T.: Ten years of visual languages research. *Proc. IEEE Sym. Vis. Lang.* (1994) 196-205.
4. McDaniel, R. G. and Myers, B. A.: Gamut: demonstrating whole applications. *Proc. 10th Ann. ACM Sym. User Int. Soft. Tech.* (1997) 81-82.
5. Myers, B. A.: Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. *Proc. SIGCHI Conf. Hum. Fac. Comp. Sys.* (1986) 59-66.
6. Patry, G. and Girard, P.: End-user programming in a structured dialogue environment: The GIPSE project. *IEEE 2001 Sym. Hum. Cen. Comp. Lang. Env.* (2001) 212-219.
7. Shneiderman, B.: Direct Manipulation: A step beyond programming languages. *IEEE Comp.* **16** (1983) 57-69.
8. Shu, N. C.: Visual Programming. Van Nostrand Reinhold Company Inc. (1988).
9. Smith, D. C., Lieberman, H., Witten, I. H., Finzer, W. F., Gould, L., Kay, A., Halbert, D. C., Myers, B. A., Maulsby, D., Mo, D., Cypher, A., Kurlander, D., Jackiw, R. N., Feiner, S., Potter, R., Piernot, P. P., and Yvon, M. P.: Watch What I Do: Programming by Demonstration. MIT Press. (1993).
10. Smith, D. C.: KidSim: programming agents without a programming language. *Comm. ACM.* **37** (1994) 54-67.