Some Basic Data Structures and Algorithms for Chemical Generic Programming

Wei Zhang, Tingjun Hou, Xuebin Qiao, Xiaojie Xu*
College of Chemistry and Molecular Engineering, Peking University, Beijing 100871

Abstract: Here we reported a template library used for molecule manipulation, we name it **M**olecular **H**andling **T**emplate **L**ibrary (MHTL). The entire library is composed of two parts: generic data structures and generic algorithms, and these two parts are associated to each other by two concepts: *Properties* and *Molecule*. Concept *Properties* describes the interface to access objects' property, which is a typical action that operates on all chemical objects. Data structures include template classes that are models of the two concepts, and algorithms are template functions that operate on the two concepts. In this paper, we discussed six methods to access objects' property and two basic methods to store molecular structure in computer. Each of these policies is realized as an individual template class. Algorithms in the form of template functions can be classified into three categories by their usage: one for molecular file input and output using many kinds of file formats, one for atom pattern recognition using SMARTS language, and the other for molecular 3D-rendering using OpenGL library. Based on MHTL, some chemical software or program concerned with determination of atomic chemical environment, operation or graphical representation of molecular structure may be developed easier and more rapidly.

INTRODUCTION

Generic programming (GP)[1] is a new programming paradigm supported by C++ programming language. Programs written in generic programming always include two parts: generic data structures (in the form of template classes) and generic algorithms (in the form of template functions). Algorithms always operate on data structures, but in generic programming the two parts are decoupled from each other, because in generic programming, data structures and algorithms are associated to each other by "concept". A concept describes a set of requirements on a data type, and when a specific data type satisfies all of these requirements, we say that it is a "model" of that concept. Actually, algorithms operate on models of its argument concept, and data structures are written to be models of algorithms' argument concepts. This "concept" abstraction is the essence of generic programming.

Generic programming has archived great progress in the past several years. The STL (The C++ Standard Template Library)[2], as the first commercial product of generic programming, was accepted in 1994 as a part of C++ standard[3]. Many basic components of C++ programming language, like string and stream, have been rewritten under generic programming.

However, in the field of chemical software development, the paradigm of generic programming has not been widely accepted. It is partly because chemists haven't realized the advantage of generic programming. Chemists are already familiar with the programming paradigm of object-oriented (OO) programming[4], which involves associating data types with a hierarchy of inheritance. Many molecular handling C++ libraries based on object-orient programming have been released in the past years. Each of them defined its set of data types to represent atoms, bonds and molecules. Users who want to use these libraries have to inherit their data types from these pre-defined data types. Things can be very hard when user wants to employ two or more libraries together, since multiple inheritance or private inheritance should be used here, and because both parent types have defined some common data member (for example, almost all atom types define a data member for element and position.), there will be either memory waste or synchronization problems, or even both.

* Correspond author, Phone: (86)-10-62757456 E-mail: xiaojxu@chem.pku.edu.cn

Now, generic programming provides us another alteration for chemical software development. As we have mentioned, instead of operating on a pre-defined data type, generic algorithms actually operate on a model of its argument concept, which give users much more freedom to define their desired data types instead of inheriting from library with pre-defined data types. Here, we reported a template library named MHTL (**M**olecular **H**andling **T**emplate **L**ibrary). The library includes some data structures and algorithms frequently used in computational chemistry. MHTL was developed when we were developing other chemical programs, including some computational programs and their graphical user interfaces. At first, we want to define a unified data type to represent chemical object, like atom and molecule. But latter, we realized that a single data type couldn't match different requirement of different programs, then we turn to generic programming and want to define a unified concept for chemical objects. Generic algorithms written under these concepts can operate on its models, which enable us to use different data type in different circumstance to get maximum efficiency. The schematic representation of MHTL is illustrated in Figure 1. The entire library includes two parts: generic data structures and generic algorithms, and in the following two sections, they were discussed systematically.

DATA STRUCTURE

In this section, we are going to describe the data structures and concepts in MHTL, on its method and usage. It includes some classes, these classes are mainly models of two concepts: *Properties* and *Molecule*. The concept *Properties* which describes a property's container is setup because the access of property is the commonest operation that run on all chemical objects. The concept *Molecule* is setup because it is the basic unit that chemical algorithm operate on. It should be noted that in this section we import many modules from BOOST[5], which is a template library distributed under GPL license, and is about to be accepted by C++ standard.

Concept Properties and Its Models. Technically, the commonest action of chemical algorithms operating on a chemical object is to set and get its properties. These properties include atom's atomic number, position, formal charge, partial charge, vdw radius, velocity, force, bond's order, molecule's boiling point, melting point, CAS identification number, and so on. As it can be seen, properties can be of any data type, even user defined types (UDTs). This uncertainty in property's type makes its access and storage a difficult job. Some old libraries were designed to use different member function to access property of different type, but a better solution to this problem in generic programming is to define a template member function inside the data type like the following:

Chart 1

```
template< typename Ptype > void SetProperty( const Ptype& value );
template< typename Ptype > Ptype GetProperty();
```

These two template member functions take property data type as their template argument, so property of any type can be accessed by these functions. However, properties of the same type can't be distinguished, and it requires that users declare each property a data type explicitly.

To help users define property types easily, we have defined a template class PropertyT as following:

Chart 2

```
template< int Pid, typename Ptype >
class PropertyT
{
public:
        PropertyT() {};
```

```
        PropertyT(const Ptype & rhs ):m_Value(rhs){ };
         operator Ptype() const { return m_Value; }
        static const int PropertyId = Pid;
private:
        Ptype m_Value;
};
```

PropertyT takes two template arguments: the first one is Pid, which is the identification number of property type, and can be used to distinguish different property; the second one is Ptype, the actual data type of property. Inside the class, a copy constructor and a data type conversion function is defined, which insures that a variable of the type PropertyT<Pid, Ptype> can be used just like the variable of Ptype. Moreover, a static and const data member named PropertyId has been defined inside the class, which is used to identify property type. We defined a trait for property to help users access PropertyId. The trait is like the following:

Chart 3

```
template< typename PropertyType >
struct PropertyTraits
{
    static const int PropertyId = PropertyType::PropertyId;
}
```

Sometimes users want to declare their property type without using PropertyT, and they may not be able to declare PropertyId inside the data type (for example, they may want a property with type “vector<int>”). Under such condition, they can declare PropertyId by specializing PropertyTraits like this:

Chart 4

```
template< >
struct PropertyTraits< vector<int> >
{
    static const int PropertyId = 10;
}
```

Type vector<int> is assigned a PropertyId 10 in this way.

Each of some usually used properties (element, formal charge, partial charge, atom name, position, and so on) has been declared as a data type using typedef, and the declaration is as following:

Chart 5

```
typedef PropertyT< 1, int> Element;
typedef PropertyT< 2, double > PartialCharge;
typedef PropertyT< 3, Vector3d> Position;
typedef PropertyT< 4, std::string> AtomName;
```

Vector3d is a user defined type to represent a three dimensional vector, which is very useful in molecular OpenGL rendering. All the declaration can be found in the supporting materials.

It is obvious that this property access action operates on data types represented chemical object like atoms, bonds and

molecules very frequently, so it is necessary to define a concept including all data types that have defined the above two functions. We defined this concept *Properties*, which means property's container.

With the concept of *Properties* defined, our problem now is how to realize these two member functions. A lot of methods can be used here.

Traditionally, property is accessed in the following stages: firstly, users declare a data member for the property inside the data type; secondly, they define setter and getter member function for this data member. This traditional way is also the most efficient way to access property, but in this way properties are not accessed by a unified interface, and the data type is hard to be extended (Part A of Appendix A illustrates a data type with properties of element, position, name and partial charge). It declares two member functions for each property.

Fortunately, we have the technique of template specialization which can help us to define a data type which is a model of *Properties* concept and use the traditional way to access property, and Part B of Appendix A demonstrates a data type named TsProperties which uses this technique to access element and partial charge. This method is called template specialization method.

When we are using template specialization method to access property, we still need to define two member functions for each property: one is setter function, and the other is getter function. This work can be very dull and hard when we have many properties to access, while it can be simplified by using the technique of template metal-programming[6].

Template metal-programming is a new kind of technique base on template technique. The main difference between template meta-programming and other traditional programming languages is that template meta-programming is not a run-time programming language, but a compile-time programming language. All the programs written in template meta-programming run and get their results in compile-time. The MPL module of BOOST library is a product of template meta-programming[7].

The core component of MPL is type list. People can firstly construct a type list, and then perform many kinds of operations on it, such as append, remove, find, and so on. Tuple[8] is a component of MPL which is appropriate to be used here, which is generated by linear inheritance of tuple field. The usage of tuple is like struct in C language. There are many example codes in BOOST library that illustrate the usage of tuple. We build a model of *Properties* which uses tuple to store each property. It is a template class named TuplePropertiesT, which takes property type list as its template argument. The declaration of TupleProperties is include as Part C of Appendix A.

These three property access techniques we discussed before (traditional method, template specialization method and tuple method) are all under such an assumption, that is we know which properties will be accessed in the run time. However, in some programs, we do not know which properties will be accessed. For example, when we are reading a molecular file, we do not exactly know before which property will be specified in the file, to save all the information in the file, so we need a property access mechanism that can convert any type property into a media type and can convert it back whenever we need.

Several methods can be used to establish such a mechanism: firstly, we can use malloc() to allocate memory, and use free() to release memory. The property is stored as void pointer, and the void pointer can be converted back to property by static cast. The code in Part D of Appendix A illustrates this method. However, this property access method is strongly not recommended because it is not type-safe.

Secondly, we can convert property into string by using "<<" operator and can convert it back by using ">>" operator. In file string_properties.hpp in the supporting materials, we define a class StringProperties, which this operator way to access properties. The class StringProperties has a very interest feature that is it supports many kinds of conversion. If you set a property of type int with value 1 and get property of type string, you can get a string with "1"; or if you get property of double and get a double with value 1.00, this feature can be used to realize a persist property class. The definition of StringProperties can be found in Part E of Appendix A.

Thirdly, we can use pointer's dynamic cast, this is a new feature of C++ programming language. In order to use it, user

needs a compiler that supports RTTI (run-time type identification). To realize property in this method is fairly difficult, but we needn't do it by ourselves either. The "any" module of boost library has realized it for us[9]. The class DynamicProperties defined in Part F of Appendix A illustrates how to construct a model of *Properties* using dynamic cast of pointer.

Till now, we have introduced six methods to access property, the traditional method, the template specification method, the tuple method, the static cast method, the string method and the dynamic cast method. These methods can be classified into two categories. The first three methods can be classified into the direct access methods and the other methods can be classified into the indirect access methods.

In these methods, the static cast method is worst because it is not type safe, we want user not to use this method. The performance of the left five methods has been compared clearly. The comparison was done on four kinds of property: element (integer number), partial charge (floating number), atom name (std string class) and position (Vector3d). Compiler used here is GNU C++ compiler 3.2, and the code is compiled under two modes: none-optimization and optimized on Level 3. The result is illustrated in Table 1. From Table 1 we can found that: (1) the traditional way is the fastest in every case; (2) For properties of basic types such as *double* and *int*, the tuple method and the template specification method can be as efficient as traditional method, but for properties of complicate class, these two method is much slower than the traditional method; (3) the direct access methods are much faster than the indirect methods; (4) in the indirect methods, the string method is much slower than the dynamic cast method, which is mainly because string method uses IO subsystem.

Though the direct access method is much faster than the indirect method, in some program we still need to use the indirect access methods. We noted here that these two kinds of methods could be combined together to form an atom data type that uses the direct access method to some pre-defined properties and uses the indirect access method to access other properties. In Part G of Appendix A, we define such a template MixedPropertiesT, which takes three arguments. The first is typelist, which is a list of property types which should be accessed using the direct access method; the second is DirectProperties, the type name of direct access properties; the last one is IndirectProperties, the type name of the direct access properties. In the relization of MixedPropertiesT, we used a meta-function **find** to find out if a specified type is in a type list, which is imported from the boost's mpl module:

In the definition of MixedPropertiesT, we use a tuple composed of a DirectPropeties and an IndirectProperties. The class finds out whether the property is in the type list automatically. If it is, it saves the property in the DirectProperties part of the tuple, else, it saves the property in the IndirectProperties part.

Concept *Molecule* and Its Models. Molecule is the common object that chemical algorithms may operate on. Generally, a molecule contains several atoms, and atoms are associated to each other by bonds; meanwhile, molecule, atoms and bonds have their own properties. Moreover, molecules can be classified into monomers and polymers, and a polymer is normally made up of several residues.

It is very difficult to describe the requirement on data types to represent molecule, yet we still try to define a concept for it, because without it we can't write generic algorithms. In MHTL, the *Molecule* concept requires data types to realize the following member functions:

Chart 6

```
int NumberAtom();
template<typename Ptype> void SetAtomProperty(int index, const Ptype& value );
template<typename Ptype> Ptype GetAtomProperty( int index );
vector<int> GetBondsOfAtom( int index );
int GetBondBetweenAtom( int index_a, int index_b );
```

```
int NumberBond();
template<typename Ptype> void SetBondProperty(int index, const Ptype& value );
template<typename Ptype> Ptype GetBondProperty( int index );
int GetBondBeginAtom( int bond_index );
int GetBondEndAtom( int bond_index );

template<typename Ptype> void SetProperty(const Ptype& value );
template<typename Ptype> Ptype GetProperty( );
```

Because sometimes we need to add atoms and bonds or to delete atoms and bonds, we set up a concept named *MutableMolecule*, which re-implements the concept of *Molecule* and requires the realization of the following four functions:

Chart 7

```
int CreateAtom();
void DeleteAtom( int index );
int CreateBond( int atom_a, int atom_b );
void DeleteBond( int bond_index );
```

As for the realization of molecule data type, many methods can be used. In these methods, adjacent list and adjacent matrix are mostly used. The adjacent list method involves in saving bond associations in a bond list, while the adjacent matrix method involves in saving them in a 2-D matrix, if atom *i* and atom *j* are bonded to each other, then mat[*i*][*j*] and mat[*j*][*i*] should be 1, otherwise, they should be 0.

The two methods both have their advantages and disadvantages. The comparison between the two methods is list in Table 2.

We have realized two template class: AdjacentListMoleculeT and AdjacentMatrixMoleculeT. They are both models of *MutableMolecule*. Their definition can be found in the supporting materials, and the declaration is like:

Chart 8

```
template<AtomProperties, BondProperties, MoleculeProperties> AdjacentListMoleculeT;
template<AtomProperties, BondProperties, MoleculeProperties> AdjacentMatrixMoleculeT;
```

The three arguments correspond to the property access method of atom, bond and molecule, respectively.

Moreover, we defined a template: PolymerT. It specialized the operation for set atom property of residue name to contain residue information, which will be useful for proteins and nuclear acids.

As we have mentioned, molecule is a very complicate concept. Our definition and realization for molecule is far from perfect, but we believe that all necessary operations on molecule have been included in the concept. When a new and better concept appears, our algorithms can be easily modified to adopt the revision.

ALGORITHM

In the last section, we have introduced the data structures used in MHTL. In this section, we introduce the algorithms used in MHTL. These algorithms are all in the form of template functions, using molecule data type as their template arguments. Algorithms in MHTL can be divided into three parts:

(1) molecule input and output subroutines using different file formats;

(2) SMARTS language interpreter and atom pattern recognition algorithms;
(3) molecule 3d-rendering subroutines using OpenGL library.

Molecular file input and output routines. There are already some released libraries that can be used to do molecular file format conversion, but these libraries all employ a huge, full-described media type, and then subroutines read molecular file into a structure of media type and write the structure into another type. During this conversion process, some parts of the type can be of no use, and since the media type is huge and complicated, so there is some efficiency lost more or less. Sometimes this waste of memory and efficiency is unbearable. The advantage of generic algorithms is that people can customize their molecular type, and make sure there is no memory waste.

The prototypes of our molecular input and output algorithms are like:

Chart 9

```
template< typename Molecule >
void ReadMdl( std::istream& is, Molecule& molecule );

template< typename Molecule >
void WriteMdl( std::istream& is, const Molecule& molecule );

template< typename InputMolecule >
void ReadPdb( std::istream& is, Molecule& molecule );

template< typename InputMolecule >
void WritePdb( std::istream& is, const Molecule& molecule );

template< typename InputMolecule >
void ReadSmiles( std::istream& is, Molecule& molecule );

template< typename InputMolecule >
void WriteSmiles( std::istream& is, const Molecule& molecule );

template< typename InputMolecule >
void ReadSybyl( std::istream& is, Molecule& molecule );

template< typename InputMolecule >
void WriteSybyl( std::istream& is, const Molecule& molecule );
```

As the prototype indicates, the argument type of these template functions should be a model of concept *MutableMolecule*. These algorithms are written according to the newest documents[10-13].

SMARTS language interpreter and atom pattern recognition algorithms. The ability to recognize atom's chemical environment is frequently needed by chemical software. The most famous solution to this problem is SMARTS language[14], a well-defined language used to describe an atom's chemical environment. Software should firstly interpret the string written in SMARTS language into a machine-readable pattern, and then test every atom to see if any one matches the pattern.

The SMARTS language uses a key character to indicate the content of each simple pattern, and some usually used key

characters include:

'D': the number of heavy atoms that bond to queried atom.
'X': the number of atoms that bond to queried atom.
'H': the number of hydrogen atoms that bond to queried atom.
'R': the ring size of queried atom.
'r': the aromatic ring size of queried atom.
'$': the substructure of queried atom.
'^': the hybrid number of queried atom.

For the full description of SMARTS language, please refer to the references[14].

In MHTL, we defined a template class SmartsPatternT to present the machine-readable pattern. The template class takes only one argument, the molecule data type. The following codes demonstrate how to use SmartsPatternT:

Chart 9

```
SmartsPatternT<Molecule>* pattern = new SmartsPatternT<Molecule>("[$(#7)]");

for( int i=0; i < molecule.NumberAtom(); i++ )
{
    if( pattern->Match( molecule, i ) )
        cout << "Atom " << i << " matches the pattern [$(#7)]\n";
}
```

Among the SMARTS key characters, the realization of 'r', 'R' and '$' is so hard that it worth more discussion. Character 'r' and 'R' involves the search of ring in a molecule, while character '$' involves the search of substructure. In MHTL, the two key characters are both realized using depth first search (DFS) method. It is the easiest algorithm for these two questions, but is not the best one, while in consideration of the usually used SMARTS patterns, we do not think DFS will be too inefficient. Better algorithms are under development. In MHTL, the aromatic ring and non-aromatic ring are justified by Huckel's $4n+2$ rules.

Practically, SMARTS patterns are always organized together to be a typing rule. Each pattern in the typing rule has been assigned an identical name or index. Typing rule is often used in force field development, ADME prediction and other computational related areas, so we defined a template class TypingRuleSetT to represent it. Its declaration is like the following:

Chart10

```
Template< typename Molecule >
Class TypingRuleSetT
{
public:
TypingRuleSetT( const char* file_name );
Virtual ~TypingRuleSetT();
Std::string TypingAtom( const Molecule& molecule, int i );
};
```

As it can be seen from the declaration, TypingRuleSetT takes Molecule data type as its template argument. Its constructor read the rule from a text file (the format of typing rule file is discussed later). It has one member function TyingAtom, which performs the atom tying job one the *i*th atom of a molecule, and returns its typing name. For atoms whose type can't be determined, it throws an exception.

The typing rule is a free format text file. There can be three kinds of lines in the typing rule file: comment lines, blank lines and the data lines. Comment lines always begins with a "#", and blank line is made up of blanks. These two kinds of lines are both ignored in typing rule file parsing procedure. A data line in tying rule file has two columns: the first is the SMARTS description the pattern, and the second is the name of the pattern. The pattern should be written in a top down sequence. It means that a former pattern can never be the sub-pattern of a latter pattern. An example typing rule file is included in the supporting materials.

Molecular 3D-rendering algorithms. When chemists are developing chemical software, they always want to display molecular 3D-structure on the screen. OpenGL[15] is a set of multi-platform subroutines to display 3D-objects and is widely accepted in displaying chemical object in all kinds of software. However, it is hard to be mastered. MSRDPGL is developed to aid chemists to develop their OpenGL chemical software rapidly and easily.

MSRDPGL supports four basic display styles: line style, stick style, ball-stick style and CPK style, and allows users to define new style. In MSRDPGL, each display style is defined explicitly as a template class, which takes Molecule as argument concept. Inside the class, two member functions are defined:

Chart 10

```
void DisplayAtom( const Molecule& molecule, int atom_index );
void DisplayBond( const Molecule& molecule, int bond_index);
```

To use style class to display molecule is fairly easy, like the following:

Chart 11

```
LineStyle style(0.1);
for( int i=0; i < molecule.NumberAtom(); i++ )
{
   style.DisplayAtom( molecule, i );
}

for( int i=0; i < molecule.NumberBond(); i++ )
{
   style.DisplayBond( molecule, i );
}
```

To import new style into MSRDPGL, users need to define new style class.

APPLICATION AND FURTHER DEVELOPMENT

We have set up some abstract concept and generic data structure to represent chemical objects. Upon them, we develop some generic algorithms. These algorithms can be used in molecular file input and output, atom pattern recognition and molecular 3D-rendering. Now, the new version of the SASWA program[16] was developed on MHTL. The input of the

molecular database file and the recognition of atom types are based on the functions in MHTL. Furthermore, the SFDOCK program[17] developed in our group is rewritten based on MHTL, and the new version of SFDOCK should be more concise and efficient. Certainly, MHTL need more elaborated improvements, and the future wok will be concentrated on the following aspects:

Firstly, we will try to simplify our concepts and develop more efficient generic data types. We think the *Molecule* concept we are using now is too generous, and we will try to setup several different molecular concepts in the future. On the other hand, there are still many methods to represent a molecular structure except for adjacency list and adjacency matrix. Each of the methods has its advantages and deficiencies, and is proper to be used in some special circumstance; we will try to realize them all.

Secondly, we are going to improve the graph algorithms, which will speed up the procedure of atom pattern recognition. For example, we can use Ulmann's algorithm in substructure matching procedure, which is more effective than the simple DFS algorithm used here.

Thirdly, we are going to develop some numerical generic algorithms. Algorithms in development include: all kinds of force fields, molecular dynamics, surface area calculation, Poission-Boltamann equation's solver and generalized Born model calculation. These algorithms are written in Fortran and released as discrete software in the past, which restrict their usage in software development. We are now rewriting them as generic algorithms using template meta programming, which helps chemists import these algorithms into their software easier.

Fourthly, we are planning to develop some COM components that control molecules using OpenGL technique. We are now designing for two components, one is MoleculeViewer, which supports only the display and transformation of molecule, and the other is MoleculeEditor, which supports the modification of molecule and an undo buffer.

CONCLUSION

Generic programming is a kind of useful programming paradigm supported by many programming language, while its usage in computational chemistry is limited. We have tried to import it into chemical software development. We defined two commonly used concepts and defined many models of these concepts based on different policies, and wrote many algorithms operating on the concept. These algorithms have three basic functionalities: molecular file input and output in different file formats, atom pattern recognition using SMARTS language and molecular 3D-rendering using OpenGL. We hope these algorithms can help chemists build their software easier and more quickly.

ACKNOWLEDGMENTS

This project is supported by National Natural Science Foundation of China (NSFC 29992590-2 and 29873003).

Supporting Information Available. The complete version of MHTL consists of approximately 6000 lines code in C++ programming language. All calculations experiments were carried out on PC. The program has been tested on IRIX, Linux and Windows operation systems, and the source codes can be obtained freely from authors upon request.

REFERENCE AND NOTES

(1). Austern, M. H. Generic Programming and the STL; Addison-Wesley Pub Co., 1st Ed., New York, Oct. **1998**.

(2). ISO/IEC 14882, In Programming languages-C++, ISO/IEC, 1st Ed., Sept. **1998**.

(3). Stepanov, A.; Lee M. The Standard Template Library, Hewlett-Packard laboratories, **1994**.

(4). Booch, G. Object-Oriented Analysis and Design with Applications, Addison-Wesley Pub Co. 2nd Ed., New York, Oct. **1993**.

(5). BOOST, http://www.boost.org.

(6). Alexandrescu, A. Modern C++ Design, Addison-Wesley Pub Co., 1st Ed., New York, Feb. **2001**.

(7). Gurtovoy, A.; Abrahams, D. In The BOOST C++ Meta Programming Library, http://www.mywikinet.com/mpl/paper/mpl_paper.pdf.

(8). Jarvi J. Tuple Types and Multiple Return Values, C/C++ Users Journal, 12(8), the 2nd paper, **2001**.

(9). Henney K., C++ Report 12(7), July/Aug. **2000**.

(10). CTFile Format, MDL Co., Aug. **2002**, http://www.mdli.com.

(11). PDB Format Description Version 2.2, the Research Collaboratory for Structural Bioinformatics(RCSB), Dec. **1996**, http://www.rcsb.org/pdb/docs/format/pdbguide2.2/guide2.2 frame.html.

(12). James, C.A.; Weininger D.; Delany J. Daylight Theory Manual, Daylight Chemical Information Systems Inc., Chapter 3, Jun. **2003**, http://www.daylight.com/release/manuals.html.

(13). Tripos Mol2 File Format, Tripos Inc., http://www.tripos.com/custResources/mol2Files/index.html.

(14). James, C.A.; Weininger D.; Delany J. Daylight Theory Manual, Daylight Chemical Information Systems Inc., Jun. **2003**, http://www.daylight.com/release/f manuals.html;

(15). Segal, M.; Akeley, K. The OpenGL Graphic System: A Specification (Version 1.5), Silicon Graphic, Inc., **2003**, http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf.

(16). Hou, T. J.; Xu, X. J. Empirical aqueous solvation models based on accessible surface areas with implicit electrostatics. *J. Phys. Chem. B* **2002**, 106, 11295.

(17). Hou, T. J.; Wang J. M.; Xu, X. J. Automatic docking of peptides and proteins using a hybrid method combined with genetic algorithm and tabu search. *Protein Eng.* **1999**, 12, 639.

Appendix A: Realization of Six Property Access Method

Part A:

```
class TraditionalProperties
{
public:
    void SetElement( int element )     { m_Element = element; }
    int GetElement( void )             {   return m_Element; }
    void SetPartialCharge( double charge ) { m_PartialCharge = charge;}
    double GetPartialCharge( )      {     return m_PartialCharge;     }
    void SetName( const std::string& name )      { m_Name = name; }
    std::string GetName( )   {    return m_Name;    }
    void SetPosition( Vector3d& position )   { m_Position = position; }
    Vector3d GetPosition( )      {    return m_Position; }
private:
    int m_Element;
    double m_PartialCharge;
    std::string m_Name;
    Vector3d m_Position;
};
```

Part B:

```
class TsProperties
{
public:
    template< typename Ptype > void SetProperty( const Ptype& value )      { }
```

```
    template< typename Ptype > Ptype GetProperty( void ) { return Ptype(); }
private:
    Element m_Element;
    double    m_PartialCharge;
    string    m_Name;
    Vector3d m_Position;
};

template<>inline void TsProperties::SetProperty(const Element& value ) { m_Element = value;}
template<>inline void TsProperties::SetProperty(const PartialCharge& value ){ m_PartialCharge = value;}
template<>inline void TsProperties::SetProperty(const AtomName& value ){    m_Name = value; }
template<>inline void TsProperties::SetProperty(const Position& value ){       m_Position = value;}
template<>inline Element TsProperties::GetProperty( void ){       return m_Element;}
template<>inline PartialCharge TsProperties::GetProperty( ) {     return m_PartialCharge; }
template<>inline AtomName TsProperties::GetProperty( ) {     return m_Name; }
template<>inline Position TsProperties::GetProperty( ){       return m_Position;}
```

Part C:

```
template< typename Typelist >
class TuplePropertiesT
{
public:
    template<typename Ptype > void SetProperty( const Ptype& value )
    {
        field< Ptype > ( m_Properties ) = value;
    }
     template<typename Ptype>       Ptype GetProperty()
     {
        return field< Ptype >( m_Properties );
    }
private:
    typename mpl::inherit_linearly< Typelist,
                                        mpl::inherit< _1, tuple_field<_2> >
                                    >::type m_Properties;

};
```

Part D:

```
class StaticProperties{
public:
    StaticProperties() {}
     virtual ~StaticProperties()
     {
        std::map< int, void*>::iterator i = m_Pmap.begin();
        for( ; i != m_Pmap.end(); i++ )
```

```
            free( i->second );
        }

        template <typename Ptype> void SetProperty(const Ptype& value)
        {
            void* ptr = malloc( sizeof(Ptype) );
            new(ptr) Ptype(value);
            m_Pmap[ Ptype::PropertyId ] = ptr;
            m_Pmap[ Ptype::PropertyId ] = new Ptype(value);
        }

        template <typename Ptype> Ptype GetProperty( )
        {
            return *(Ptype*)( m_Pmap[ Ptype::PropertyId ] );
        }
    private:
        std::map< int, void* > m_Pmap;
    };
```

Part E:

```
    class StringProperties{
    public:
        template <typename Ptype>
        void SetProperty(const Ptype& value)
        {
            int pid = PropertyTraits<Ptype>::PropertyId;
            std::ostringstream oss;
            oss << value;
            m_Pmap[ pid ] = oss.str();
        }
        template <typename Ptype>
        Ptype GetProperty( )
        {
            int pid = PropertyTraits<Ptype>::PropertyId;
            if( m_Pmap.count( pid ) )
            {
                std::istringstream iss( m_Pmap[ pid ].c_str() );
                Ptype value;
                iss >> value;
                return value;
            }
        }
    private:
        std::map< int, std::string > m_Pmap;
```

```
};
```

Part F:

```
class DynamicProperties{
public:
    template <typename Ptype> void SetProperty(const Ptype& value)
    {
        int pid = PropertyTraits<Ptype>::PropertyId;
        m_Pmap[ pid ] = value;
    }
    template <typename Ptype> Ptype GetProperty( ) const
    {
        int pid = PropertyTraits<Ptype>::PropertyId;
        if( m_Pmap.count( pid ) )
        {
            return boost::any_cast<Ptype>( m_Pmap.find( pid )->second );
        }
    }
private:
    std::map< int, boost::any > m_Pmap;
};
```

Part G:

```
template< typename typelist, typename DirectProperties, typename IndirectProperties >
class MixedPropertiesT
{
public:
    template< typename Ptype >
    struct ChooseProperties
    {
        typedef typename mpl::find< typelist, Ptype >::type iter;
        typedef typename mpl::end< typelist >::type end_iter;
        typedef typename mpl::if_< boost::is_same< iter, end_iter >,
                                   IndirectProperties,
                                   DirectProperties >::type    type;
    };
    template<typename Ptype >
    void SetProperty( const Ptype& value )
    {
        typedef ChooseProperties< Ptype >::type Properties;
        field< Properties >( m_Properties ).SetProperty( value );
    }
    template<typename Ptype>
    Ptype GetProperty()
```

```
    {
        typedef ChooseProperties< Property >::type Properties;
        return field< Properties >( m_Properties ).Get<Ptype>();
    }
private:
    typename mpl::inherit_linearly< mpl::list< SpecializedProperties, OtherProperties >,
                                    mpl::inherit< _1, tuple_field<_2> >
                                  >::type m_Properties;
};
```

Tables:

Table 1: The comparison between property access methods

Compiler Setting	Property Name	Property Access Method				
		Traditional	Template Specialization	Tuple	String	Dynamic
No-optimized	Element	20	47	56	13298	951
	PartialCharge	42	89	58	17450	971

	AtomName	849	2040	1560	12020	2714
	Position	147	413	324	38440	1210
Level-3 optimized	Element	2	2	2	11925	262
	PartialCharge	3	4	4	16762	272
	AtomName	745	1737	1310	14617	1765
	Position	73	272	210	36816	514

Table 2: The comparison between adjacent list method and adjacent matrix method

Function	Adjacent List	Adjacent Matrix
GetBondOfAtom	Slow	Fast
GetBondBetweenAtom	Slow	Fast
NumberBond	Fast	Slow
GetBondBeginAtom	Fast	Slow
GetBondEndAtom	Fast	Slow

Figures:

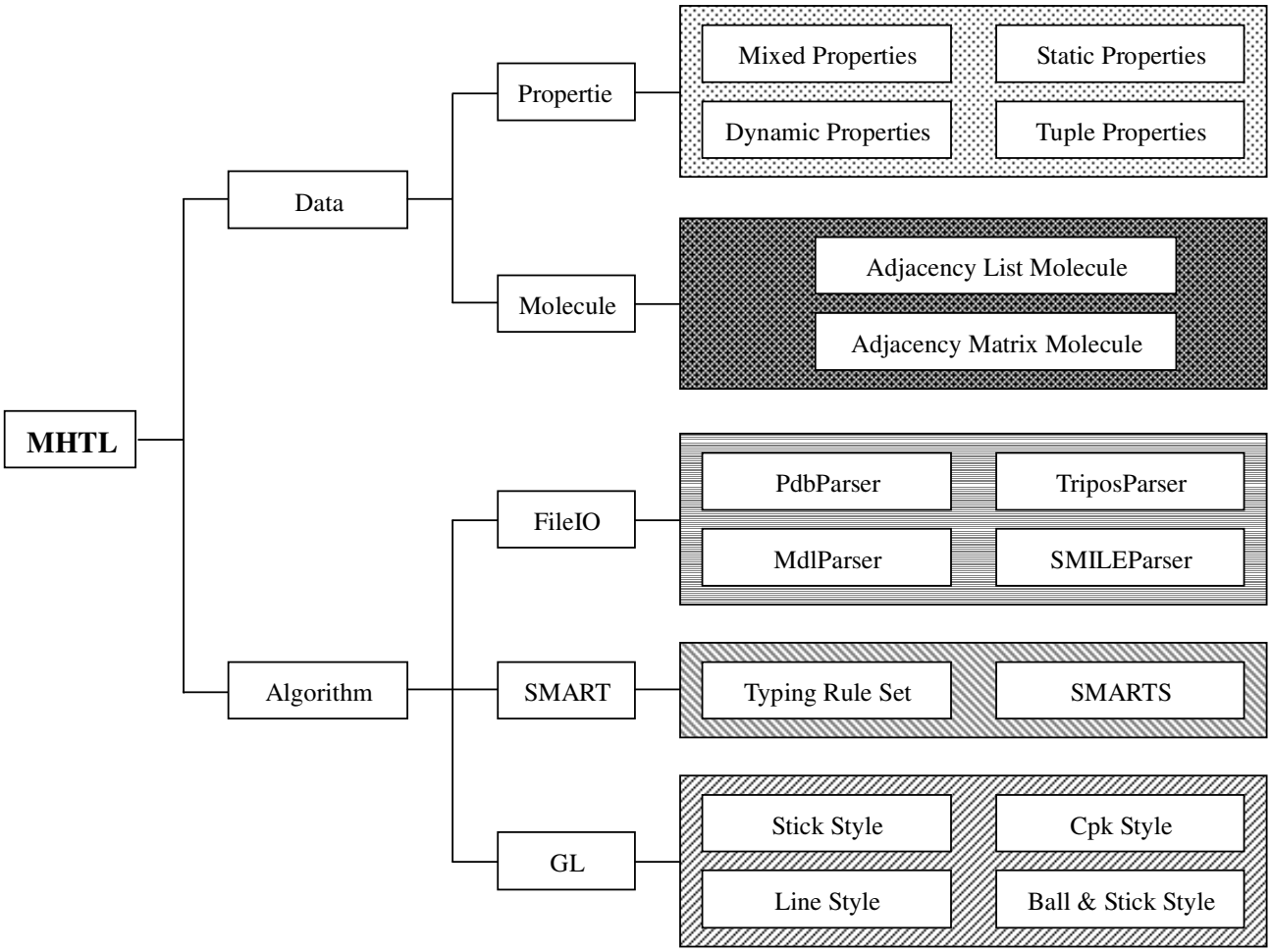

Figure 1. The schematic representation of MHTL

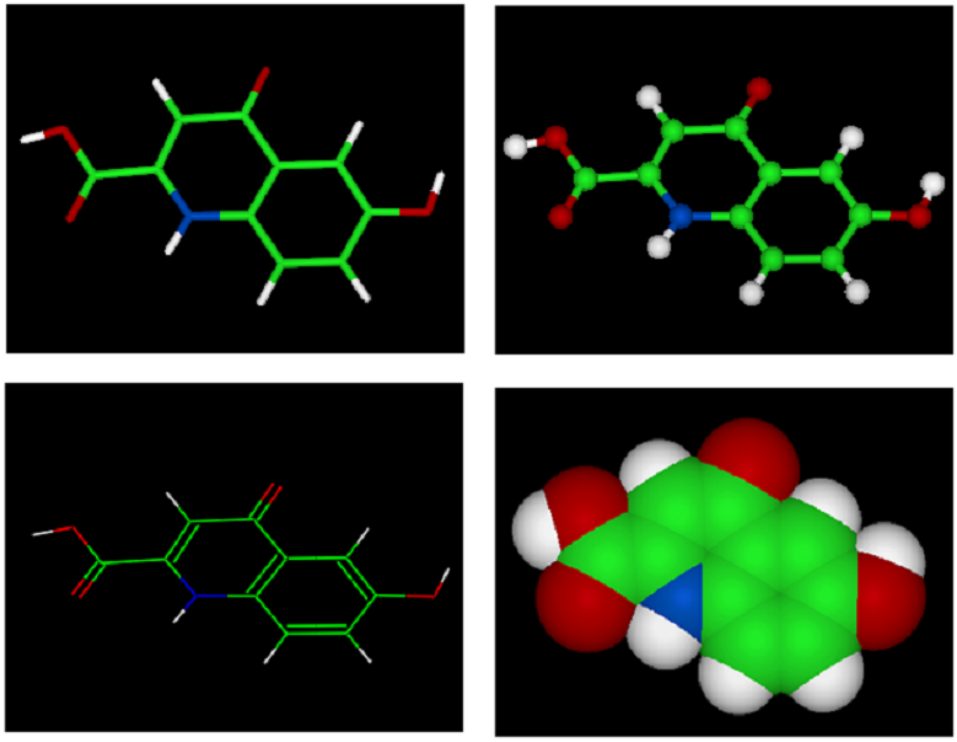

Figure 2. Molecules displayed in different styles