

PredTox 2003 Theory of Operation

8 February 2004

Introduction

PredTox is a program for performing molecular aquatic toxicity predictions using a Linear Solvation Energy Relationship (LSER) model¹. Input compounds are written in SMILES², a linear notation for representing molecular structures; output consists of five computed LSER model variable values (V_i , Pi^* , Beta, Alpha, and Delta, called vector values) for each compound which are used to evaluate a set of multiple linear regression equations to compute the compound's estimated toxicity.

Each compound is analyzed and broken down into a set of constituent fragments and aromatic rings. PredTox contains a built-in set of fragments and aromatic rings with vector values for each; to compute the total vector value the constituent vector values are summed. One of several sets of values is chosen for each fragment depending on whether it is adjacent to an aliphatic or aromatic ring, or embedded in an aliphatic ring, or if it shares a ring with other fragments.

The purpose of this document is to describe the overall organization of PredTox and to describe some of the algorithms and data structures. The intended audience includes those who wish to gain a better understanding of the program logic as well as programmers who might need to modify PredTox in the future.

Feature Summary

PredTox supports both batch and interactive modes of operation. In interactive mode multiple documents each working with a single compound are supported, as well as multiple views of the same document, for more convenient viewing of a long analysis. An interactive document consists of a two connected panes, a left-hand selection pane, and a right-hand output pane. The selection pane is used to select the compound to be processed and to specify any processing options. The output pane shows the result of the molecular analysis for the selected compound. Interactive analysis output may be printed in the usual way. In batch mode, PredTox processes an input file of compounds and stores an output file for subsequent viewing or printing; the number of compounds that may be processed in batch mode is limited only by the amount of disk space available to hold the input and output files.

In both modes, compounds may be selected either by giving a CAS number or a name or partial name that is matched against a database supplied with PredTox, or via a SMILES string. In interactive mode, all compounds that contain the given partial name are presented in a dialog box; the user selects one compound from the list, which is then analyzed. In batch mode, all matches to a partial name are processed.

The level of detail for each analysis may be varied, to include or exclude detailed per-fragment or per-ring vector values, or to include or exclude the regression calculations.

Regression parameters are specified in a regression parameter file; this file may be changed to run the same regressions with new parameter values, or to specify new regressions.

¹ James P. Hickey, Andrew J. Aldridge, Dora R. May Passino, and Anthony M. Frank, "An Expert System for Prediction of Aquatic Toxicity of Contaminants," in Expert Systems for Environmental Applications, Judith M. Hushon, editor, American Chemical Society, 1990.

² Daylight Theory Manual, Chap. 3, in Daylight Documentation, Version 4.82 Release Date 06/16/03
<http://ftp.daylight.com/dayhtml/doc/theory/theory.toc.html>

Vector values for the built-in fragments may be overridden via a vector value override file. Overridden values are marked in the program output.

PredTox was developed using Microsoft Visual Studio 97 running on Windows 95 and tested on Windows XP.

Program Organization

PredTox is divided into two major components: a molecular parser which accepts a single SMILES string and performs the required analysis, and a GUI wrapper that interacts with the user. The two components are quite independent and interact along a narrow interface; this permits changes to be made to either component without great effect on the other.

The following discussion references the PredTox source code.

Molecular Parser

The molecular parser is composed of the following files:

- Patw.h
- Patw.c
- fragtablew.h

The heart of PredTox is the molecular parser, which has four phases of operation. In the first phase, the SMILES string is processed character by character and a tree is constructed describing the chemical. Bonds between atoms in the chemical are represented as links between nodes in the tree, and ring closures are noted. This tree is the central data structure in PredTox and permits the SMILES notation to be ignored once the tree is populated. This is of advantage for two reasons. First, the tree structure directly admits recursive solutions to most operations, such as finding rings; processing the SMILES strings in this fashion is more difficult since recursive patterns are much less obvious. Second, it is well-known that many distinct SMILES notations map to the same chemical. Processing SMILES strings therefore requires developing rules for generating unique strings that are themselves quite complicated. Using a tree structure obviates the problem; all equivalent SMILES strings generate topologically equivalent tree structures.

In the second phase, all rings in the compound are located and spurious aromatic and aliphatic rings are suppressed.

In the third phase, the fragments comprising the compound are determined by comparison with the fragments listed in the built-in fragment table, and the appropriate vector values to be used for the given fragment are determined. Because of the need to match every fragment in the fragment table with every atom of the compound, PredTox is optimized to suppress this match whenever possible. To this end, a characteristic vector is defined for every fragment in the table, noting the number of each kind of atom and the number of each type of bond for each fragment. A similar characteristic vector is accumulated for the input compound. A candidate fragment is only matched to the compound if the compound fragment characteristic vector dominates the candidate characteristic vector, that is, all elements of the former vector are at least as large as all elements of the latter. This prevents matching operations from being initiated that cannot possibly succeed. Measurements have shown that implementing this technique nearly doubles the speed of the parser.

In the fourth and final phase, the vector contributions of the rings and fragments of the compound are summed and the compound listing is generated.

Patw.h

This is the header file for the molecular parser. It contains all of the basic definitions needed by the parser. For example, the atoms accepted by the parser are enumerated here. Some constants and structures that should be in this header file are scattered through the parser; when looking for a definition, try the header

file first, then the beginning of the parser file, then between the top-level functions of the parser. The header file also contains function prototypes for all parser functions.

Patw.c

File `patw.c` contains the molecular parser; the parser is entered by calling the function `patw`. Each call to `patw` parses one compound.

Some important structures are defined in `patw.c`:

- `obj_t` Describes an atom, including links to other atoms of the compound, vector value for the atom, and pointers to the ring and fragment this atom is in, if any.
- `ringbook_t` The built-in "book" of aromatic rings known to PredTox.
- `summary_t` Describes mismatches between given and computed vector values for "given format" batch data.
- `pt_t` Describes a periodic table entry, for providing vector values to atoms not part of any ring or fragment.
- `ring_t` Describes a ring.
- `linked_t` Describes a pair of linked rings.
- `fused_t` Describes a pair of fused rings.
- `frag_t` The built-in list of fragments known to PredTox.

First, `patw` unpacks its arguments. After generating some initial output containing identifying information, it calls `pop_smiles` to create the tree representation from the SMILES string.

The next major routine to be called is `find_rings`, which computes all possible rings starting from all ring closures defined for the compound. Aromatic rings which are not of a valid length (currently 6, 7, or 8) or do not match any of the rings in the ringbook are discarded, as are two-element rings and duplicates of rings previously seen. All geometrically possible aliphatic rings are accepted. Acceptable rings are added to the ring list, and atoms that belong to more than one ring are remembered, since they indicate the presence of fused rings.

Next, linked aromatic rings are processed in `linked_ar_rings`. After building a list of candidate rings consisting of all aromatic rings in the compound, this routine counts the number of bonds connecting each pair of rings, ignoring any ring pair that shares a carbon atom (i.e. fused rings). A bond is not counted if either of the bond's atoms is fused to more than one ring, to suppress double counting in dense fused structures. Rings connected by bonds are added to a linked ring list for later tallying.

Next, fused aromatic rings are handled in `fused_ar_rings`. After building a list of candidate rings consisting of all aromatic rings that contain fused atoms, this routine generates a list of pairwise-fused rings, ignoring shared atoms that are not carbons and suppressing excessive sharing of carbon atoms in dense fused structures. Pairs of fused rings are added to a fused ring list. Finally, the pairwise-fused rings are divided into groups of connected rings by computing equivalence classes; two sets of fused rings are in the same equivalence class if they are transitively fused, i.e. if ring A is fused to ring B, and ring B is fused to ring C, then ring pairs AB and BC are in the same equivalence class. This grouping is necessary to apply different adjustments during tallying based on the number of fused rings in a group.

Following this, fragments in the fragment table are matched to the compound in `match_frags`. This routine attempts to match each fragment in the fragment list to a potential matching subtree rooted at each atom position of the compound. This matching is performed recursively; the match fails at the first mismatched atom, mismatched bond, or unavailable free bond between the fragment and the corresponding candidate compound subtree. In addition, fragment atoms matching ring atoms must have at least two bonds, to prevent fragments from matching ring positions without enough bonds to complete a ring. Atoms and bonds in the compound are marked as taken as the matching operation descends through the fragment tree; if everything matches the fragment is added to a list of located fragments for later tallying;

otherwise the markings are removed and the next fragment is tried. Candidate fragments whose characteristic vectors are not dominated by a compound subtree at a given position are ignored for that position. Fragments are matched in descending total bond count within descending total atom count order, to ensure that larger fragments match before smaller fragments do; this allows vector values to be given for a large fragment that are different from the sum of the vectors of the fragment's subfragments.

Next, the correct vector value from the set of values for each fragment is determined in `frag_vecs`, based upon the relationship of the atoms in the fragment to the rest of the compound. Each fragment in the fragment table has six different vector values; one of these will be selected based on the relationship of the fragment to other fragments and rings in the compound. The six vectors are:

- Aromatic Fragment is on an aromatic ring, with no other fragments on the ring.
- Aromatic with other groups Fragment is on an aromatic ring, and there are other fragments on the same ring.
- Aliphatic in ring Fragment is in an aliphatic ring, regardless of the presence or absence of other fragments in the same ring.
- Aliphatic on ring Fragment is on an aliphatic ring, with no other fragments on the ring.
- Aliphatic with other groups Fragment is on an aliphatic ring, and there are other fragments on the same ring.
- Aliphatic If none of the previous conditions apply.

The vector chosen is identified in the output by appending a parenthesized abbreviation after the fragment name; in the order listed above, the abbreviations are: (ar), (ar wog), (al ir), (al or), (al wog), and (al). If the computed vector is not present, the relevant aromatic or aliphatic vector is used instead, and the missing vector portion is placed in parentheses: (ar (wog)), (al (ir)), (al (or)), and (al (wog)). A missing aromatic or aliphatic vector generates a diagnostic, and no total vector is computed.

In general, a fragment is on a ring if one of its atoms is attached to an atom of the ring by a bond. A fragment is in a ring if the ring passes through two free bonds of different atoms of the fragment, or if the fragment only possesses one atom that has two or more free bonds; otherwise the fragment is on the ring.

For fragments on multiple rings, the fragment is assigned a value in the following order of precedence: fragments on aromatic rings, then fragments in aliphatic rings, then fragments on aliphatic rings. If a fragment is on multiple aromatic rings, the aromatic ring with the largest number of attached fragments will be treated as if it were the only ring to which the fragment is attached. A similar strategy is used for a fragment on multiple aliphatic rings.

Finally, the vector values of each ring and fragment in the compound are tallied and displayed in `tally`. This routine first traverses the ring list, summing the vector values of all rings, and displaying a line containing the ring name and vector value if "Show fragment output" is checked. Following this ring adjustments are calculated to account for shared carbons, and Pi* and Beta adjustments are made for fused rings, applying adjustments for each ring in a fused ring group that depend on the number of rings in the group.

Further adjustments include recognizing an apparent five-element aromatic ring attached to three benzene rings as a two atoms of one benzene ring linked to one atom each in the other two rings. Azulene, another special case, is handled by recognizing and tallying the five- and seven-element component rings separately.

Fragments are then tallied, listing the fragment name (and parenthesized selected vector abbreviation as described above) and vector value. If the value has been overridden the line is marked with a trailing asterisk.

All remaining atoms not part of any fragment or ring are tallied next, using vector values from the built in periodic table subset, and olefins and alkynes not part of any fragment are tallied using built-in values.

The total vector value is computed and displayed, and for "given format" batch files the vector is compared to the given vector and a diagnostic is generated if they do not match.

Following the tally, regression processing is performed. If a regression parameter file named REGR.CSV exists in the directory containing the PredTox executable, the file is opened and records are read from it. Each record defines the name of a regression parameter set and values for ten parameters. For each valid record, a toxicity calculation is performed and output.

In a similar manner, vector override processing is performed. If a vector override file OVER.CSV exists in the directory containing the PredTox executable, the file is opened and records are read from it. Each record identifies a particular fragment in the fragment table, four selectors that determine which of the six vectors is to be overridden, and the new overriding vector value.

fragtablew.h

This file contains the built-in fragment table. Its construction is explained in more detail in the Modifying PredTox section below.

Graphical User Interface

The GUI is composed of the following C++ source files:

- ChildFrm.cpp
- MainFrm.cpp
- NameDialog.cpp
- PredTox.cpp
- PredToxDoc.cpp
- PredToxView.cpp
- ScrollerView.cpp
- SelectorView.cpp
- SplitterFrame.cpp
- StdAfx.cpp

In addition, Visual Studio provides a header file for each of these source files. These header files are generally not extensively modified and are not discussed further.

The relevant contents of each of these files is discussed below. The files contain the code that executes in response to user actions. Significant amounts of development went into creating the classes, member variables, message maps, menus, menu items, dialog boxes and controls that are used by this code. Most of these objects were created using Visual Studio's Class Wizard using standard techniques (see, for example, Horton's excellent text on the subject³) and are not documented further here.

ChildFrm.cpp, MainFrm.cpp, PredToxDoc.cpp, PredToxView.cpp, StdAfx.cpp

These files are almost entirely boilerplate code from Visual Studio.

NameDialog.cpp

This major function in this file is `OnInitDialog`, which is responsible for creating the list box holding the chemical names matching the user's given partial name. The ten database files are opened in turn and the CAS number of records matching the partial name are added to a global array `GlobalCASList` while the full compound names are added to the list box. After the user has selected a name and clicked OK, `OnOK` returns the full chemical name in the global `GlobalChemName` for further processing.

³ Ivor Horton, *Beginning Visual C++ 5*, Wrox Press Ltd, 1997.

PredTox.cpp

This file contains the `InitInstance` initialization code responsible for initializing the display. It also contains the `OnBatchRun` function which is called to process a batch run request. After putting up dialog boxes to select the batch input and output files, the batch input file is opened, and records are read sequentially from it. For each record, `proc_csv` is called to parse the CSV format. Three styles of batch files are supported: a "general format" whose lines each can specify one of CAS #, partial name, or SMILES string; a "given format" whose lines can each specify a compound name, given vector values, and a SMILES string; and a "given metal" format equivalent to the given format with the addition of the compound molecular weight. The formats are distinguished by the number of fields; general format records are either given to a routine that searches the database files for a single record matching the given CAS #, a set of records for the partial name, or directly to the parser. Given format records are given to the parser directly.

ScrollerView.cpp

This file implements the output pane of an interactive document. The `OnDraw` function is called whenever the output pane must be redrawn; after selecting a fixed-pitch Courier font, it passes the CAS #, compound name, and SMILES string to the molecular parser. On return, `OnDraw` breaks the output assembled into `GlobalOut` by the parser back into lines and displays them via `TextOut`, computing the document, line and page sizes along the way to adjust the output pane scrollbars appropriately.

SelectorView.cpp

This file implements the selection pane of an interactive document. The `OnUpdate` function is called whenever a change has been made to any of the fields of the selection pane. The editing change code towards the end of this file guarantees that only one of the three edit controls will contain text at the time this function is called, and handles the copying of text entered into the edit controls into the member variables of the active document, and vice versa. If the user has typed a CAS #, the `findcas` routine is called to search the database for the appropriate record, whose fields then update the SMILES and compound name document member variables. If the user typed a partial name, the name dialog described previously returns the selected chemical name and CAS#; the CAS # is used to look up the database record for the matching SMILES string. If the user typed a SMILES string, it is remembered in the document member variable but the CAS# and chemical name are not set.

The functions `GetFocusClass` and `DoClipboardOp` support the following Copy, Cut, and Paste operations, and the remainder of the functions handle GUI actions in response to user input.

SplitterFrame.cpp

This file implements a simple static splitter to construct the interactive document.

Component Interface

The interface between the molecular parser and the GUI is defined in `Proc.c`. Two kinds of interfaces are supported. For interactive use, a static buffer `GlobalOut` is used to accumulate results from the parser, which is accessed by the GUI afterwards and is used to build the scrolling output pane. A static buffer was used to limit the likelihood of memory leaks. The size of `GlobalOut` determines the maximum number of characters an interactive analysis can produce. In batch mode, the parser sends output to the I/O stream `os`, which is opened by the GUI on the batch output file.

The `proc` interface routine creates an argument vector similar to that expected by a standalone program and calls the parser. A normal return from the parser leaves the output either in the global buffer or on the I/O stream.

To support abnormal returns, in case the parser cannot continue execution due to a severe error, a non-local goto is taken via the `set jmp/long jmp` mechanism. An example of a severe error would be if the global

buffer fills; such conditions could be detected at a fairly deep nesting level in the parser, and it would be impractical to unwind the call stack before returning.

Modifying PredTox

This section briefly describes how to modify the PredTox source code to extend its capabilities. Such modifications should be undertaken only with extreme care. A backup copy of the source code should be made before any changes are undertaken. It is assumed that Visual Studio will be used to make changes to the code, that a new project has been defined in Visual Studio housing a copy of the PredTox source code, and that the user is familiar with Visual Studio procedures. In particular, Visual Studio should be set to link as a static library, set to produce a release configuration, and pre-compiled header files should be turned off.

After making the desired changes, build `PredTox.exe`. If no errors were introduced a new version of the project will be created in `Release/PredTox.exe`. Copy this executable to the directory in which you installed PredTox.

Note: if you are using Visual Studio 5 or earlier, you may get complaints about being unable to compile the help file when you recompile the source code after making these changes. These diagnostics may be ignored; it is not necessary to recompile the help file to make any of the following changes.

Modifying the ringbook

To change vector values of an existing ring in the ringbook, locate the line headed by the name of the ring whose values you wish to change. The four values in braces immediately following the ring name comprise the vector value for that ring, in the canonical order: V_i , Pi^* , Beta, and Alpha. Change these values to the new desired values. All values are integer representations of fractional values. V_i must be specified to four significant figures; the fourth value is usually 0. The other values are specified to two significant figures. For example, a V_i value of .491 would be specified as 4910, a Pi^* value of .59 would be specified as 59.

Adding to the ringbook

To add a ring to the ringbook, locate the last line headed by the name of a ring in the ringbook, and add a comma to the end of that line; then add a new line immediately after that line. Specify the name of the new ring in quotes, followed by a brace-enclosed vector value for the ring with numeric values specified as above, followed by an integer representing the number of atoms in the new ring, followed by a brace-enclosed list of atom names. Use the previous line as model. Several lines may be added; all but the last must end with a comma. Currently, rings must have 6, 7, or 8 members.

Care should be taken not to add rings which will be hidden by previous ones. PredTox sequentially matches ringbook rings to a compound by rotating each ring an atom at a time, and then inverting and rotating again. Thus a ring CNNCCC early in the ringbook will hide a later ring CCCNNC. Such a situation is not detected by PredTox, and ring CCCNNC's vector values will never be used.

Adding fragments to the fragment list

Adding fragments to the fragment list is a somewhat tedious process requiring painstaking attention to detail.

To add a fragment to the fragment list, examine `fragtablew.h` and make a copy of an existing fragment list entry with which to work. A fragment list entry has an initial static part beginning with the string `"/* frag n */"`, and a final atom list with one line for each atom of the fragment, ending with two successive closing braces, a comma, and a blank line. No more than 10 atoms may be defined for a single fragment.

On the first line of the copy, change the fragment name enclosed in double quotes to the new fragment name; for existing fragments this name is actually a SMILES-like string, It is not further interpreted but is printed in the fragment list. The next number on the line is the fragment number. For the first added fragment, set this number to one more than the fragment number of the last fragment in the table, and increment this number for every fragment added. You will need to keep track of this number for subsequent fragment additions, so that no two fragments in the table have the same number. The third and final number on the first line should remain zero.

The next three lines of the copy define six sets of vector values in the following order: aliphatic, aromatic, aliphatic with other groups, aromatic with other groups, aliphatic on ring, and aliphatic in ring. Provide correct vector values for the desired sets, with numeric values specified as above, and use the string `UNPOP_VEC` for those cases where no vectors are to be defined.

The next line specifies the characteristic vector for the fragment. The first group of 12 values counts the number of each type of atom in the fragment, in the order C, N, O, F, Si, P, S, Cl, Br, Sn, and I. The second group of three values counts the number of each type of non-free bond in the fragment (e.g. bonds that are connected to other atoms of the fragment), in the order single, double, triple. The final group of three values counts the number of each type of free bond in the fragment (e.g. bonds that are available for bonding to other atoms in a compound), in the order single, double, triple. Adjust the characteristic vector appropriately for the new fragment.

The next line contains three values: the total number of atoms in the fragment, the total number of bonds of all types in the fragment, and the third value remains zero. Adjust the counts to reflect the new fragment.

The next lines form the atom list of the fragment with one line per atom of the fragment. These lines construct a tree for the fragment. Each line begins with the atom name. The next group of three values remains zero. The next group of four values represents bonds from that atom to other atoms in the fragment, free bonds available for bonding to other atoms of a compound, and unpopulated (nonexistent) bonds. A bond to another atom in the same fragment is written $L(n)$, where n is one less than the number of the line in the atom list describing the atom to which this atom is bonded. For example, if this atom were bonded to the atom described by the third line of the atom list, the value $L(2)$ would be used. A free bond is denoted by the string `FRE`; a nonexistent bond is denoted by the string `UNP`. The last group of four values represents the type of bond at each position of the previous group, where 3 means a triple bond, 2 a double bond, 1 a single bond, and 0 a nonexistent bond at this position. For example, a 2 in the second position of the last group means that the bond in the second position of the previous group is a double bond. Using the pseudo-SMILES string of the new fragment name, create an atom list with one line for each atom of the new fragment, and adjust the bond linkage and bond type groups appropriately for each atom to properly construct a tree representing the fragment.

This completes the construction of the new fragment, which must be moved to the correct place in the fragment table. The fragment table is ordered by descending number of bonds within descending number of atoms. In other words, all fragments with the same total number of atoms are grouped together; the groups are ranked in order of descending total atom number. Within a group of fragments with the same total number of atoms, fragments are ranked within in order of descending total bond number. Move the new fragment to the correct place in the fragment list based on these ordering rules.

After recompilation, check the correctness of the fragment table changes by running a compound containing the new fragment(s) through PredTox in batch mode with "Show fragment list" checked. If the batch output is free of diagnostics complaining about fragment table inconsistencies and shows the new fragment(s) with their proper vector values in the fragment list, the fragment table was successfully updated. Due to its complexity, PredTox performs extensive checking of the fragment table at runtime. Note: in this case, a "notify programmer" error message complaining about fragment table errors after user modifications almost certainly refers to errors in the modifications and not to errors in PredTox as shipped.

Adding groups to the reactive group list

Adding groups to the reactive group list is also a tedious if somewhat simpler process, again requiring painstaking attention to detail.

To add a group to the reactive group list, examine `rgrouptablew.h` and make a copy of an existing group list entry with which to work. A group list entry consists of four lines; the first line consists of the string `/* group n */`, and the last ends with two successive closing braces and a comma. No more than 10 fragments may be defined for a single reactive group entry.

On the first line of the copy, change the group number in the comment. For the first added group, set this number to one more than the group number of the last group in the table, and increment this number for every group added.

On the second line of the copy, replace the string with the correct string to be output should the new group be matched to the compound.

The next line contains one group of values enclosed by braces, an integer value, and two more groups of values enclosed by braces. The first group consists of four values that should be set to zero. The following integer value should be set to the number of fragments in this group. The next group of values should be set to the fragment numbers of each fragment making up this group; the fragment numbers must match the fragment numbers assigned in the `fragtablew.h` file. Only fragments listed in the `fragtablew.h` file can be recognized. Unused entries in this group should be set to -1. The final group of values indicate the numbers of each of the fragments required to make up this reactive group. Unused entries in this group should be set to zero. The number of entries used in the second and third groups of values should match the preceding integer value of the number of fragments in the group.

The last line of the entry consists of a final group of values that specifies the name of each fragment. This name should match the name of the fragment as given in the `fragtablew.h` file; this is encouraged but not required. The name is not further interpreted but is printed as part of the reactive group should this group be matched to the compound.

This completes the construction of the new reactive group, which must be moved to the correct place in the reactive group table. This table is ordered by descending number of fragments. In other words, all groups with the same total number of fragments are grouped together, and the groups are ranked in order of descending number of total fragments. Move the new reactive group to the correct place in the group list based on these ordering rules.

After recompilation, check the correctness of the reactive group table changes by running a compound for each new group through PredTox in batch mode with "Show toxicity estimates" checked. If the batch output shows each new reactive group identified in its compound, the reactive group table was successfully updated. Note that in distinction to the fragment table, it is not necessary for PredTox to check the reactive group table for errors at runtime.