UCSF
UC San Francisco Electronic Theses and Dissertations

Title
Ligand-based Perspectives on the Evolution of Enzyme Function

Permalink
https://escholarship.org/uc/item/4wr1x54m

Author
Chiang, Ranyee Agnes

Publication Date
2008-09-08

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Ligand-based Perspectives on the Evolution of Enzyme Function

by

Ranyee Agnes Chiang

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Biological and Medical Informatics

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, SAN FRANCISCO

Copyright 2008
by
Ranyee Agnes Chiang

Acknowledgements

The text of Section 2.2, Chapter 3 of this dissertation is a revised reprint of the material as it appears in the Journal of Molecular Biology and PLoS Computational Biology, respectively. The text of Chapter 5 has been submitted for review to the Journal of Structural and Functional Genomics. The coauthors Andrej Sali and Patricia Babbitt listed in these publications directed and supervised the research that forms the basis for the dissertation. For Section 2.2 and Chapter 5, the main contributors are Margaret Glasner and Ursula Pieper, respectively, and these sections contain the text that relates to my contributions.

Many people have brightened the path during my journey through graduate school, and I must thank all of them for their support.

First and foremost, I must thank my thesis advisors, Patricia Babbitt and Andrej Sali. They have been great models of how to be generous and considerate members of the academic community. I am grateful that they have given me the freedom to explore and challenge myself, and at the same time, that they have given me their optimistic support and encouragement.

My steady progression from orals to thesis defense was made possible by my orals committee and thesis committee. Ajay Jain, Matt Jacobson, Tanja Kortemme, and Jim McKerrow have given me valuable advice and steered me in fruitful and interesting directions.

It has been fun to be part of the BMI program, the program with the friendliest and most supportive faculty and students. The quality of the program comes from the

efforts and spirit of the program's directors, Tom Ferrin and Patsy Babbitt. And special thanks to our excellent program coordinators, Barbara Paschke, Denise Chan, Becca Brown, and Julia Molla, who have kept all of us on track and made sure we didn't fall through any cracks.

I would not be the volleyball player I am today (not very good, but much better than when I started!) without the Thunderforce volleyball team – Holly "The Recruiter" Atkinson, Liz "Humble Destroyer" Clarke, Kris "The Psychiatrist" Kuchenbecker, Ben "Bam Bam" Lauffer, Laura "Look Ma, No Arms" Lavery, Vito "Whoa" Mannella, Arjun "Air Jordan" Narayanan, Tuan "Cap'n Crunchtime" Pham, Hesper "Carpet" Rego. Whether we were winning championships or whether we didn't quite get everything together to earn additional shorts, it was all a ton of fun. Hoooo!

I worked with the Science and Health Education Partnership to teach science, and hopefully, some of my students learned something about science. But I probably learned way more than I taught, especially from Linda Akiyama, Jen Chu, Jean McCormack, Katherine Nielsen, Claudia Scharff , Rebecca Smith. It was a pleasure to work with you and thanks for helping me find my job.

Thanks to all of my labmates, past, present, and future. It was great to have you as colleagues. And although it probably made the attainment of this Ph.D. take longer, it was great to have you as friends with whom I could talk about any topic, play whiffle ball and Ultimate, and cook and eat food – all these activities were essential to my survival. A special little shoutout to my nearest cubicle neighbors, who made sure I didn't rush too much with my work - Courtney Harper, who always says the most unexpected things, Sunil Ojha, who, as someone who is super politically aware even without being able to

vote, should be a model for all the citizens of the country, Alex Schnoes, who I can always count on for advice, empathy, and hope, and Alan Barber - this thesis would not have all its signatures without his pen!

I was lucky to be part of the best BMI class ever! Simona Carini – Carini is only one letter different from "caring" and that's no accident. Thanks for sharing your home and your tasty desserts. Christina Chaivorapol, it was fun trading tasty treats and local foods back and forth with you! John Chuang, you are the nicest secret agent ever. Libusha Kelly, we can continue our good times on the East Coast! Mike Kim, I'm glad to have your analysis of everything! Juanita Li, your beautiful family makes me hope I can have one of my own someday. Tuan Pham, thanks for giving me a "second chance."

I would not be who I am and where I am without the love and support of my family. My parents valued education above everything else and I am eternally grateful for their example and for their support of my education. My little sisters started out as my toys and as my entertainment, and now they are also my friends.

And lastly, special thanks to Mark Peterson, who not only belongs to so many of the above categories, but also belongs to a category of his own. We've gone through interviews, classes, and group meetings together and we've traveled much of the world together. At each step, I have been fortunate to have Mark as my greatest supporter and as my most vocal challenger. Because of his faith in me, I can jump higher, run faster, and I am stronger. And because he was with me every step of the way, I dedicate this dissertation to Mark. Thank you, Mark. If we can do this together, we can do anything together.

Abstract

Ligand-based Perspectives on the Evolution of Enzyme Function

Ranyee A. Chiang

Studying the evolution of enzymes and their functions improves our ability to determine the functions of unknown enzymes and engineer enzymes to perform new functions. An enzyme's function is determined by its sequence and structure and we can trace the evolution of enzymes and their function by analyzing their sequences and structures. This dissertation describes work to extend these analyses of sequences and structures to use comparisons of enzyme functions in order to study enzyme evolution.

The first studies described in this dissertation use "traditional" sequence and structure analyses follow the evolution of an especially complex superfamily of enzymes. We found that within the protein family that we were studying, despite having a single function and a single evolutionary origin, no sequence or structural motifs unique to this family could be identified. We also found that sequence and structural determinants of specificity may lie outside of the active site. These results show that the correlation between sequence, structure, and function is not always straightforward and demonstrate the need for direct analyses of functions to study enzyme evolution.

In analogy to sequence and structure-based studies of enzyme evolution, we have examined a large number of enzyme superfamilies using a new computational analysis of patterns of substrate conservation. The patterns that we observe among substrates during enzyme evolution suggest more complex patterns of functional divergence than what has

been proposed by previous theories of enzyme evolution. The method has been automated to facilitate large-scale annotation of enzymes discovered in sequencing and structural genomics projects. A data resource has been developed to share this data with researchers interested in improving predictions of enzyme function and in enzyme engineering.

The final study presented describes work to select templates for structural genomics efforts. The eventual goal is to increase the number of structures available to determine enzyme function and specificity using methods like comparative modeling, computational docking, and other experimental efforts.

Table of Contents

Chapter 1 Introduction **1**
1.1. Metabolism and Enzymes 1
1.2. Enzyme Function and Experimental Determination 2
1.3. Computational Prediction of Enzyme Function 3
1.4. Models of Enzyme Evolution 4
1.5. Studying Enzyme Evolution 7
1.6. Outline of Thesis 9

Chapter 2 Sequence Analysis to Find Determinants of Enzyme Specificity **12**
2.1. Introduction 12
2.1.1. Evolutionary trace 13
2.1.2. Evolution of the enolase superfamily 14
2.2. Evolution of Structure and Function in the *o*-Succinylbenzoate Synthase/N-Acylamino Acid Racemase Family 16
2.2.1. Introduction 16
2.2.2. Methods 18
2.2.3. Results 21
2.2.4. Discussion 36
2.3. Non-active Site Determinants of Enzyme Specificity 40
2.3.1. Introduction 40
2.3.2. Methods 41
2.3.3. Results 43
2.3.4. Discussion 46
2.4. Conclusion 48

Chapter 3 Evolutionarily Conserved Substrate Substructures for Automated Annotation of Enzyme Superfamilies **49**
3.1. Abstract 49
3.2. Introduction 50
3.3. Methods 52
3.3.1. Dataset – Enzyme superfamilies 52
3.3.2. Definitions 53
3.3.3. Finding the conserved substrate substructure 55
3.3.4. Finding the reacting substrate substructure 55
3.3.5. Overlap between reacting and conserved substructures 56
3.3.6. Variation in which substructure is reacting 57
3.4. Results 57
3.5. Discussion 65
3.5.1. Patterns of substrate conservation across many superfamilies 66
3.5.2. Functional annotation of superfamilies and enzymes 67
3.5.3. Guidance for protein engineering 71
3.5.4. Future directions for substructure analysis 73

3.5.5. Conclusions ... 74

Chapter 4 Substructures for Enzyme Evolution and Engineering Resource ... 76
4.1. Introduction ... 76
4.1.1. Enzyme evolution and superfamilies ... 77
4.1.2. Computational molecular docking to predict substrate specificity ... 77
4.1.3. Enzyme engineering ... 78
4.1.4. Data resource ... 78
4.2. Methods and Results ... 79
4.2.1. Data ... 79
4.2.2. MySQL database structure ... 80
4.2.3. Web interface ... 81
4.3. Conclusion ... 86

Chapter 5 Target Selection and Annotation for the Structural Genomics of the Amidohydrolase and Enolase Superfamilies ... 87
5.1. Abstract ... 87
5.2. Introduction ... 88
5.3. Methods ... 90
5.3.1. Target selection ... 90
5.3.2. Analysis of the target structures ... 92
5.4. Results and Discussion ... 93
5.4.1. Target selection ... 93
5.4.2. Analysis of the resulting crystallographic structures ... 97
5.5. Conclusion ... 101

Chapter 6 Conclusion and Future Directions ... 102

References ... 108

Appendix A. Evolutionary Trace ... 116
A.1.1. Usage ... 116
A.1.2. Script Code ... 119

Appendix B. Reacting and Common Substructures ... 127
B.1.1. Usage ... 127
B.1.2. Program Code ... 130

List of Tables

Table 2.1. Relative divergence of the OSBS family 26

Table 2.2. Location of class-specific residues 46

Table 3.1. Overlap between reacting and conserved substructures (f_c and f_r) 59

Table 4.1. Number of Entries in SEEER Tables.. 80

Table 5.1. Success rates for the steps in the structural genomics pipeline as of June 2008. 97

Table 5.2. Comparison of template-based modeling statistics for the 63 ENSPEC/NYSGXRC structures and all 327 NYSGXRC structures (May 2007). .. 98

List of Figures

Figure 1.1. Pathway evolution by the recruitment of enzymes from different pathways (Schmidt, Sunyaev et al. 2003) 5
Figure 2.1. Evolutionary trace example. 13
Figure 2.2. Capping and barrel domains in the enolase superfamily. 15
Figure 2.3. Genomic context of menaquinone biosynthesis genes. 22
Figure 2.4. Bayesian phylogenetic tree of proteins in the MLE subgroup. 24
Figure 2.5. Bayesian phylogenetic tree of the proteins in the OSBS/NAAAR 28
Figure 2.6. Analysis of sequence conservation in the OSBS/NAAAR family 30
Figure 2.7. Comparison of OSB binding orientation 33
Figure 2.8. Comparison of the 20s and 50s loop positions in *E. coli* OSBS and *Amycolatopsis* OSBS/NAAAR 34
Figure 2.9. Evolutionary trace for four families in enolase superfamily. 44
Figure 2.10. Class-conserved residues in domain interface. 45
Figure 3.1. The conserved substructure (c) (blue square) 54
Figure 3.2. Reacting substructure (r) (red triangle) 54
Figure 3.3. Measures of overlap between reacting and conserved substructures 56
Figure 3.4. Summary of superfamilies and their conserved substrate substructures 58
Figure 3.5. Distribution of average fraction of conserved substructure that is reacting... 61
Figure 3.6. Patterns of overlap between reacting and conserved substructures 62
Figure 3.7. A) Variation in the fraction of the conserved substructure that is reacting. B) Variation in which part of conserved substructure is reacting. 64
Figure 3.8. Protein structures with unknown function can be annotated with superfamily-conserved substructures. 70
Figure 3.9. Enzyme engineering strategy. 72
Figure 4.1. Summary of database schema. 80
Figure 4.2. Database interface – Browse superfamilies. 82
Figure 4.3. Database interface - Superfamily display page. 84
Figure 4.4. Database interface - Enzyme display page. 85

Figure 5.1. Flowchart of the target expansion strategy 95
Figure 5.2. Phylogenetic tree of the organisms for the selected amidohydrolase targets . 96
Figure 5.3. Cytoscape clustering for the amidohydrolase superfamily (a) and enolase superfamily (b). 100
Figure 6.1. Flowchart of the substrate prediction strategy 105

Chapter 1

Introduction

1.1. Metabolism and Enzymes

All living systems take in nutrients from their environment. Animals eat, plants take in gases from the atmosphere and nutrients from the soil, and simpler organisms take in individual nutrient molecules. Living systems actually use a more varied and often more complicated set of molecules than the basic nutrients that they ingest. Thus, to survive, living systems must be able to chemically convert the simple set of nutrients into all the necessary forms. Living systems also increase their chance of survival if they have a mechanism to chemically break down various toxins into less harmful forms. Living systems use enzymes to satisfy these basic survival needs.

Enzymes, a subset of biological molecules called proteins, are essential for driving these chemical reactions in living systems. They convert molecules (substrates) into a chemically different form by first attaching themselves (binding) to the substrate and then facilitating the substrate's chemical change (catalyzing a chemical reaction). An enzyme lowers the thermodynamic energy barrier for its reaction to proceed, often through the stabilization of transition states that are very unstable when the enzyme is not

present. Each enzyme has the capability to catalyze a specific reaction on a specific substrate (or sometimes, on a set of substrates). (Fersht 1985) An organism's entire assortment of enzymes make up its metabolic network and allow the organism to perform all the different chemical reactions needed to convert nutrients into usable forms, break complex molecules down into usable pieces, and break down harmful or unneeded molecules to allow the pieces to be reused or excreted.

1.2. Enzyme Function and Experimental Determination

There are a several types of assays to confirm the substrate and product of an enzyme. (Bergmeyer 1974) Using spectrophotometric methods, the appearance of a new product can be detected by a change in light absorbance that is associated with the substrate changing into a product. The enzyme's reaction can also be coupled with another enzyme and a downstream product can also be detected. The heat absorbed or released during a chemical reaction can also be measured in calorimetric assays. These assays can be used to monitor the substrate and/or product concentration over time which can then be used to calculate the maximum velocity (V_{max}), reaction rates (k_{cat}), substrate concentrations required for the enzyme (for K_M, concentration required for enzyme to reach half of V_{max}), and the efficiency of an enzyme (k_{cat}/K_M). These values are most easily calculated for enzymes that follow Michaelis-Menten kinetics (Briggs et al. 1925). Using any the above methods requires that the substrate and/or product are known, or at least that the substrate and/or product have been narrowed to a smaller list of molecules that can be reasonably tested. Therefore, methods for predicting an enzyme's function *in silico* from its sequence or structure can greatly facilitate the experimental determination of an enzyme's function. Because sequence and structure information are expanding

quickly due to genomics, metagenomics (Riesenfeld et al. 2004), and structural genomics projects (Chandonia et al. 2006) and because experimental methods are time and resource intensive, it is often only possible to determine enzyme function computationally.

1.3. Computational Prediction of Enzyme Function

The general strategy for computationally predicting the function of uncharacterized enzymes usually involves finding homologous enzymes that are likely to perform the same function, and then transferring the function from the characterized to the uncharacterized enzyme. This success of strategy depends on 1) the strategy used to find homologous enzymes and 2) how well the functions have been conserved between homologous enzymes. These issues are discussed separately in the following paragraphs.

The basic algorithm for finding homologous enzymes is the Basic Local Alignment Search Tool (Altschul et al. 1990). Starting from the nucleotide or amino acid sequence of a given query enzyme, the BLAST algorithm is used to search sequence databases for other proteins with statistically significant sequence similarity. While BLAST is suitable for finding closely to moderately related sequences, an iterative version of BLAST called PSI-BLAST (Altschul et al. 1997) is more appropriate for finding more distantly related sequences. There are additional methods that can also be used to find homologs based on sequence similarity (Pegg et al. 1999; Krishnamurthy et al. 2005). Each of these methods have their own strengths and weaknesses (Brenner et al. 1998; Sauder et al. 2000), but in general, more distantly related proteins are harder to detect accurately than less distantly related proteins. Because structures are more conserved than sequences, structural similarity can be used as additional evidence of common ancestry when sequence similarity is difficult to detect. There are a number of

methods that can be used to detect homology by structure similarity (Holm et al. 1996; Shindyalov et al. 1998; Lupyan et al. 2005).

Using these sequence and structure similarity methods to predict functions becomes more difficult with increasingly distant relationships. Not only are these relationships harder to detect, functions are also less likely to be conserved at these distant levels of relatedness (Hegyi et al. 1999; Wilson et al. 2000). In addition, the conservation of function is not uniform across different families (Glasner, Fayazmanesh et al. 2006) and superfamilies (Gerlt et al. 1998) – some sequences can diverge considerably while retaining the same function while other sequences can diverge very little and have different functions (Seffernick et al. 2001). Several large-scale studies have examined the extent of the non-uniformity in the evolution of enzyme function (Rost 2002; Tian et al. 2003). This variation in the evolution of enzymes leads to difficulties with predicting enzyme function accurately, and thus, many sequence databases are filled with erroneous annotations (Brenner 1999; Devos et al. 2001; Gilks et al. 2002). To understand how to improve the prediction of enzyme function, we need to examine in more detail the process of enzyme evolution and how functions are conserved or vary.

1.4. Models of Enzyme Evolution

Over time, changes in the genes that code for enzymes lead to variations in the enzymes themselves. Some enzymes are more successful in catalyzing needed reactions and meeting new environmental demands. These "fit" enzymes contribute to the survival of the individual organism with those enzymes, and, in turn, the survival of the enzymes themselves. Inversely, enzymes that are detrimental to the organism will lead to the

organism being less likely to survive. The primary mechanism that leads to diversity in the enzyme repertoire involves the duplication and then divergence of enzymes (Todd et al. 2001). After an ancestral enzyme undergoes a gene duplication, there will be a redundant copy of the enzyme. One copy of the enzyme, now free from functional constraints, can accumulate mutations. This divergence leads to new enzymatic functions, both detrimental and beneficial, upon which natural selection can act. There are several hypotheses that describe in more detail how enzymes and their functions diverge (Schmidt, Sunyaev et al. 2003) and these hypotheses are described in the following paragraphs.

To understand how individual enzymes evolve, it is useful to examine how pathways of enzymes evolve. Jensen first proposed in 1976 that new enzyme pathways are assembled by duplicating enzymes from different existing pathways in a patchwork fashion (Jensen 1976) (Figure 1.1). This hypothesis, called the "patchwork hypothesis," has subsequently been established by a number of examples and studies (Babbitt et al. 1997; Copley 2000; Aharoni et al. 2005).

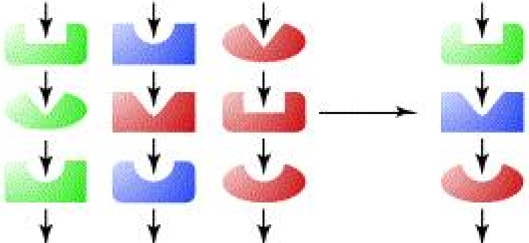


Figure 1.1. Pathway evolution by the recruitment of enzymes from different pathways (Schmidt, Sunyaev et al. 2003)

During the evolution of new pathways according to the patchwork hypothesis, it is easier to "reuse" an enzyme that already promiscuously or partially performs the function needed for the new pathway (O'Brien et al. 1999). Thus, existing enzymes are

recruited during evolution to perform modified functions while often maintaining some aspects of the ancestral function. Consequently, among contemporary enzymes we observe groups of evolutionarily related enzymes that share some aspects of molecular function and differ in others. The most divergent groups of evolutionarily related enzymes that still share aspects of function are called superfamilies. Within a superfamily, we define a family as a set of proteins that perform the same overall catalytic reaction in the same way.

Previously, both large-scale and focused studies of enzyme evolution have recognized two primary models of how function is conserved. In the retro- or substrate-conserved model of enzyme evolution, Horowitz's original hypothesis describes how an existing enzyme in a pathway is duplicated and then evolves to convert new molecules into the substrate for the original enzyme in a metabolic pathway (Horowitz 1945; Horowitz 1965). In the resulting pathway, the newly evolved enzyme will function to provide a reaction required upstream of the original enzyme (i.e., the product of the newly evolved enzyme would be the substrate for the parent). In the second model, chemistry-constrained evolution, the ancestral enzyme, which can be from any pathway, is already promiscuous for or performs a fundamental type of chemistry (often a partial reaction) in common with the function of the daughter enzyme. The aspect of catalysis shared by the ancestral and daughter enzymes is maintained through conservation of structural features such as active site residues (Babbitt et al. 1997; Gerlt et al. 2001; Porter et al. 2004). The key difference between these two models is in the pattern of function conservation within each. Related proteins that have diverged via the retro- or substrate-conserved model will bind substrates in common while the chemical reactions

with those substrates differ. In the chemistry-constrained model, divergence can give rise to large superfamilies performing many different reactions. Members of such superfamilies will have conserved some aspect of the chemical reaction, which is often a partial reaction, while the substrates they use and their overall chemical reactions differ.

1.5. Studying Enzyme Evolution

The value of any analysis of the evolution of enzyme function depends on how we describe enzyme function, with respect to both the detailed molecular functions of individual enzymes and the properties of function shared across diverse members of enzyme superfamilies. Previous approaches to study enzyme evolution range from detailed manual analyses of small numbers of related enzyme families and superfamilies to automated analyses of many superfamilies. The former have often included not only analyses of sequences and structures but also comparisons of the substrates and reaction mechanisms of the constituent enzymes. These studies have been useful for annotating new sequences and structures and for generating and testing hypotheses about patterns of enzyme evolution (see (Babbitt et al. 1996; Bessman et al. 1996; Holden et al. 2001; Allen et al. 2004; Mildvan et al. 2005) for examples). However, because of the expert knowledge required and their time-intensive nature, these types of analyses are not feasible for large numbers of superfamilies. Other semi-automated efforts have contributed to our understanding of enzyme evolution and data from these analyses have been made available in a number of online resources that include the Structure-Function Linkage Database (Pegg et al. 2006), MACiE (Holliday et al. 2007), the Catalytic Site Atlas (Porter et al. 2004), and EzCatDB (Nagano 2005).

Automated analyses (Shah et al. 1997; Todd et al. 1999; Schmidt, Sunyaev et al. 2003) have used enzyme classification systems, like the Enzyme Commission (EC) system (Tipton et al. 2000), to represent functional properties and determine what properties are conserved. The EC system represents a large proportion of known enzyme reactions, classifying each enzyme with a hierarchical set of four numbers that uniquely identify a reaction, and is easy to use for large-scale analyses. However, this system, developed before analyses of enzyme evolution were common, does not provide a detailed description of enzyme function or substrates at the atomic level (Rison et al. 2000). Moreover, the EC classification of function often does not correspond with either the aspects of function that are conserved or those that can change during evolution. These issues make this system unsuitable for evaluating how enzyme function evolves, especially when evolutionary relationships are distant (Babbitt 2003). For enzymes, the Gene Ontology (GO) system's (Ashburner et al. 2000) molecular function classifications, also often used to describe and analyze function, largely recapitulate the EC system. Several groups have analyzed enzyme relationships and evolution on a large scale while using substrate and reaction similarities (Nobeli et al. 2005; Keiser et al. 2007; O'Boyle et al. 2007). Although these similarity metrics are useful, especially for clustering enzymes by their substrate similarities, they are not informative about what specific aspects of function are conserved.

To do large-scale and detailed studies of the general principles behind enzyme evolution, we first need a way to describe enzyme function that is both systematic and detailed. With such a description, we can begin to look at how function is conserved during enzyme evolution, within specific superfamilies and among enzymes in general.

1.6. Outline of Thesis

The overarching goal of this body of work is to examine how enzyme functions evolve by focusing on the molecules transformed by the enzymes. Because the enzymes themselves also contribute to the story of how they evolve, enzyme sequences and structures are also the object of study in this thesis. In addition providing new perspectives and information to improve our understanding of the constraints that drive enzyme evolution, our goal is also to improve our ability to predict enzyme function and engineer enzymes to perform new functions. The following paragraphs outline the sections of this thesis and how the work in each of these sections contributes to these goals.

The next chapter (Chapter 2) of this thesis demonstrates how traditional analyses of sequence and structure can be used to study how enzyme function evolves. For a well-studied enzyme superfamily, I report the results of two studies to trace the evolutionary history of individual families and analyze the conservation in sequence and structure to find specificity determinants. The second study was done in collaboration with Dr. Margaret Glasner, who was the primary contributor. For both studies, we were interested in how functions, especially functional promiscuity, vary within families and how that is determined by sequence and structure. The results demonstrated that the situation is complicated. To leverage additional information to clarify this complicated situation, we next turn to studying enzyme functions directly.

We have extended analyses of conservation in sequence and structure to examine the conservation in enzyme substrates (Chapter 3). We have used graph isomorphism algorithms to find the substructures that are conserved among all of the members of a

particular superfamily. These analyses have been automated to enable us to study a large set of enzyme superfamilies. The results for these superfamilies enable us to 1) probe general questions about how enzymes and their substrates evolve and 2) improve our ability to engineer new enzyme functions, and 3) improve the precision of predictions of enzyme function.

To make this substructure information accessible and useful for researchers interested in enzyme evolution, function prediction, and enzyme engineering, we have created a data resource. In Chapter 4, we describe how the conserved substructure information can be explored through this data resource. Additionally, this resource allows researchers to examine function variation within one superfamily as well as for a particular enzyme.

Because experiments to determine enzyme function are time and resource intensive, computational methods are required to predict functions or at least direct experimental researchers to which substrates to test. The final chapter is focused on work to facilitate the prediction of enzyme substrates accurately in the absence of experimental information. The docking strategy (Kitchen et al. 2004), that involves calculating how well different molecules fit into an enzyme's active site and choosing the best-fitting molecules, requires the three-dimensional structure of the enzyme. When there is no experimentally determined structure, comparative modeling (Baker et al. 2001) can be used to predict the enzyme's structure based on the structure of a homologous enzyme. In Chapter 5, I describe a study to select targets for structural genomics efforts. Dr. Ursula Pieper is the primary contributor on this project. I contributed to the sequence and modeling analyses to find new superfamily members and select targets for crystallization.

The goal is to increase the number of experimental structures to increase the number of enzyme structures that can be modeled which then improves our ability to predict the functions of those enzymes through methods like docking.

Chapter 2

Sequence Analysis to Find Determinants of Enzyme Specificity

2.1. Introduction

Although majority of this work is focused on ligands and how substrate specificity changes during enzyme evolution, because the substrate specificity is determined by an enzyme's sequence and structure, it is also important to consider these pieces of the enzyme evolution picture. In this chapter, I present two studies of how functional specificity is determined by variation in sequence and structure. For the first study, I was the primary contributor. For the second study, Dr. Margaret Glasner was the primary contributor and I performed the evolutionary trace and sequence analyses. The results of this second study were published in 2006 (Glasner, Fayazmanesh et al. 2006) and the sections of the published work that relate to my contribution are included in this chapter.

Before describing the results of these studies, I first discuss the sequence analysis method that was used in these studies (Section 2.1.1) and the enzyme superfamily that is the focus of these analyses (Section 2.1.2).

2.1.1. Evolutionary trace

The evolutionary trace (ET) method (Lichtarge et al. 1996; Madabushi et al. 2004) is commonly used to identify conserved sequence elements at different levels of divergence in a group of related proteins. Based on the input of a multiple sequence alignment, the ET method finds class-specific residues that have been evolutionary conserved in and are specific to a superfamily, subgroup, or family. In other words, the class-specific residues are not only conserved within that particular class, they are also not conserved between different classes (Figure 2.1). When mapped to a protein's structure, the class-specific residues often correspond to functionally important residues (Madabushi et al. 2002). When the different ET classes correspond to different enzyme functions, class-specific residues often correspond to residues that mediate the differences in specificity and function. The documentation and code for our implementation of evolutionary evolution can be found in Appendix A.

Figure 2.1. Evolutionary trace example.

Residues in red are class-specific for Family 1. Residues in gold are class-specific for Family 2. The aspartic acid (D) at the 11th position is conserved but not class-specific, as the same amino acid residue is conserved across multiple families.

2.1.2. Evolution of the enolase superfamily

Studying protein evolution requires identification of homologous proteins that have evolved to perform different functions, such as those found in mechanistically diverse superfamilies. Mechanistically diverse superfamilies are defined as groups of homologous proteins which are unified by a common chemical attribute of catalysis, although overall reactions can be quite different.(Gerlt et al. 2001) Here, we focus on the enolase superfamily, which includes enzymes catalyzing at least 14 different reactions.(Gerlt et al. 2005) All enolase superfamily enzymes utilize a common partial reaction in which a proton alpha to a carboxylate is abstracted by a base, leading to a metal-stabilized enolate anion intermediate. Apart from this conserved partial reaction, the overall reactions catalyzed by enzymes in this superfamily are quite divergent, including racemization, β-elimination, and cycloisomerization. Very few residues are required for the superfamily partial reaction; three metal-binding residues are well conserved across the superfamily, but the identity and position of the general base is not universally conserved.

Enolase superfamily proteins are composed of two domains, a ~200 amino acid C-terminal modified $(\beta/\alpha)_8$-barrel domain ($((\beta/\alpha)_7\beta)$) and a ~100–150 amino acid α+β domain comprised of elements from both the N and C termini, which we call the capping domain (Figure 2.2). As with other $(\beta/\alpha)_8$-barrel domain proteins, the active site is nestled in a depression formed by the C-terminal ends of the β-strands of the barrel domain. The capping domain is structurally conserved among all members of the enolase superfamily and has not been found in combination with any other $(\beta/\alpha)_8$-barrel domain protein superfamily, with domains of other folds, or as a single domain protein. Thus, it

appears that the two domains have been co-evolving since the origin of the enolase superfamily. The capping domain closes the active site and appears to play a role in determining substrate specificity and conformational changes that occur upon substrate binding. These functions are thought to be primarily mediated by two N-terminal loops, centered around positions 20 and 50 (numbering defined relative to *Escherichia coli o*-succinylbenzoate synthase; PDB identifier 1FHV), which will be referred to as the 20s and 50s loops. In most enolase superfamily members, the 20s loop is disordered in the absence of ligand, and ordering of this loop upon substrate binding results in interactions with the ligand and shields the active site from solvent.(Lebioda et al. 1988; Neidhart et al. 1991; Landro et al. 1994; Wedekind et al. 1994; Gulick et al. 2000; Thompson et al. 2000) The domain structure and the 20s and 50s loops are the focus of Section 2.3.

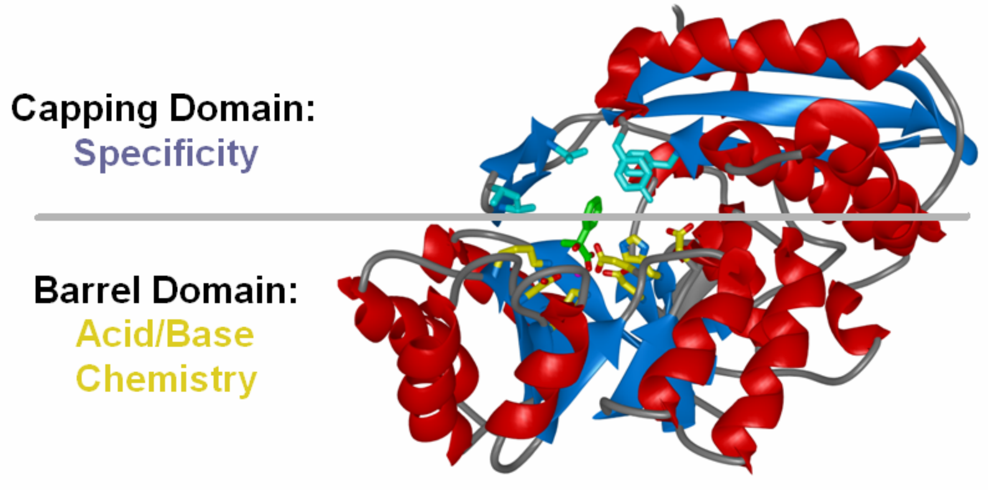


Figure 2.2. Capping and barrel domains in the enolase superfamily.

Categorizing superfamily members into families, or groups of proteins sharing the same function, is often accomplished by establishing a sequence similarity threshhold.(Todd et al. 1999; Devos et al. 2000; Wilson et al. 2000; Rost 2002; Tian et al. 2003) However, families in the enolase superfamily, as in other superfamilies, have most

likely diverged at different rates or at different times during evolutionary history, making it difficult to define a similarity score cutoff that separates different isofunctional families. The *o*-succinylbenzoate synthase (OSBS) family poses a particularly thorny problem. First, sequence similarity between some OSBSs barely exceeds random similarity scores expected between unrelated proteins, making it impossible to define a similarity score that encompasses all OSBSs but excludes proteins of other functions. Second, a promiscuous protein from *Amycolatopsis* sp. T-1-60 that shares 42% identity with the OSBS from *Bacillus subtilis* catalyzes both OSB synthesis and N-acylamino acid racemization (Palmer et al. 1999). Even experimental characterization does not adequately determine the physiological function of this enzyme, since it catalyzes OSB synthesis and racemization of N-succinylphenylglycine at equivalent rates.(Taylor Ringia et al. 2004) Thus, the OSBS/N-acylamino acid racemase (NAAAR) family is an especially interesting subject for investigating protein evolution because it includes both extremely divergent enzymes having the same function and very similar enzymes having different functions and is the focus of the following section (Section 2.2).

2.2. Evolution of Structure and Function in the *o*-Succinylbenzoate Synthase/N-Acylamino Acid Racemase Family

2.2.1. Introduction

The evolution of new protein functions is a major puzzle in biochemistry. Given that closely related proteins can have different functions, and distantly related proteins can have the same function, what kinds of structural alterations are required or tolerated during protein evolution? In addition, what characteristics of a particular protein

determine its degree of evolvability, or the likelihood that it will evolve a new function? Some previous work has indicated that evolution often proceeds through promiscuous intermediates(Ycas 1974; Jensen 1976; Hughes 1994; O'Brien et al. 1999; Schultes et al. 2000; Gerlt et al. 2001; Matsumura et al. 2001; Copley 2003; Schmidt, Mundorff et al. 2003; Aharoni et al. 2005) and that conformational flexibility of surface loops near the active site might contribute to promiscuous substrate binding and hence to the evolution of promiscuous functions(James et al. 2003). Unfortunately, there are still few proteins whose evolution, structure, and function have been analyzed in enough detail to fully evaluate these hypotheses. With the advent of large-scale genomic sequencing we are poised to answer these questions. Understanding how proteins evolve will help address several longstanding problems in biochemistry, including how to redesign proteins in the laboratory and how to predict function from sequence and structure.

Here, we have studied the evolution of the OSBS/NAAAR family. This study begins to answer several questions about how function and structure evolve in extremely divergent protein families. First, what sequence and structural features must be conserved to maintain function in extremely divergent families? Second, by what mechanisms do proteins evolve new functions? And finally, what functional and structural characteristics of a protein make it more or less capable of evolving a new function? Our study of the OSBS/NAAAR family’s evolution demonstrates that sequence, structure, and modes of substrate binding are surprisingly malleable. In addition, we have identified a number of proteins of unknown function whose experimental characterization would be valuable for understanding evolutionary relationships and structural determinants of catalysis in the enolase superfamily. We also demonstrated that the accuracy and extent of functional

annotation could be improved using rigorous phylogenetic reconstruction accompanied by analysis of genomic context. Lastly, our in depth analysis of the evolution, structure and function of the OSBS/NAAAR family identified several characteristics of *Amycolatopsis* OSBS/NAAAR which might enhance its evolvability relative to other OSBSs.

2.2.2. Methods

Identification of menaquinone pathway genes

Menaquinone biosynthesis genes were identified in complete and incomplete genomes using the Seed Annotation and Analysis Tool from the Fellowship for Interpretation of Genomes (FIG).(Overbeek et al. 2004) Genes were initially annotated as menaquinone pathway genes if the percent identity of a pairwise protein alignment covering >90% of the length of a characterized menaquinone pathway protein was >40%. Experimentally characterized menaquinone pathway proteins include all pathway proteins from *E. coli*; menB, menC, menD, menE, and menF from *B. subtilis*; ubiE from *Geobacillus stearothermophilus*; and menA and menB from *Synechocystis* sp. PCC 6803.(Meganathan et al. 1981; Taber et al. 1981; Driscoll et al. 1992; Rowland et al. 1995; Koike-Takeshita et al. 1997; Palmer et al. 1999; Johnson et al. 2000; Meganathan 2001) As a second criterion, genes were annotated as encoding a menaquinone pathway protein if they were five or fewer genes distant from another menaquinone pathway gene and their proteins had BLAST expectation values $<10^{-20}$ relative to reliably annotated menaquinone pathway proteins when searching the nr database. Most of the remaining genes were provisionally assigned functions if their proteins share ~25%–40% identity with a characterized menaquinone pathway protein and nearly all proteins identified as

being similar (BLAST E-values $<10^{-5}$ using the nr database) are annotated as having that function.

Identification of MLE subgroup members

The initial enolase superfamily data set was downloaded from the Structure-Function Linkage Database (SFLD).(Pegg et al. 2005; Pegg et al. 2006) Additional superfamily members were identified using a subset of the superfamily filtered to include only proteins sharing <35% identity as input for Shotgun.(Pegg et al. 1999) This program performs a BLAST search (Altschul et al. 1990) of each input sequence and outputs a score indicating the number of input sequences that find a given BLAST hit, allowing homologs which have barely significant BLAST E-value scores to be identified. These sequences were then manually screened to remove fragments and to verify that they contained the canonical catalytic residues of the enolase superfamily. The final enolase superfamily data set was compared to HMMs from the SFLD to classify sequences into subgroups and isofunctional families. All further analyses were performed using protein sequences matching the MLE subgroup HMM with expectation values $<10^{-18}$ and any other enolase superfamily sequences, which could not be classified into a subgroup or family by the HMMs.

Phylogenetic analysis

The MLE subgroup and outlying enolase superfamily members were aligned using Muscle v.3.52. (Edgar 2004) The initial alignment was manually refined using structural alignments of muconate lactonizing enzyme (1MUC), L-Ala-D/L-Glu epimerase (1JPM and 1JPD), N-acylamino acid racemase (1SJB and 1XS2), and OSBS (1FHV and *B. bacteriovorus* OSBS). Structural alignments were generated by MinRMS

(Jewett et al. 2003) and the structure matching and alignment feature of UCSF Chimera from the Resource for Biocomputing, Visualization, and Informatics at the University of California, San Francisco (supported by NIH P41 RR-01081). (Pettersen et al. 2004) Phylogenetic reconstruction was performed using Bayesian and distance methods. Bayesian trees were constructed with MrBayes v3.1.1(Ronquist et al. 2003; Altekar et al. 2004) under the WAG amino acid substitution model(Whelan et al. 2001) using a gamma distribution to approximate rate variation among sites.

Distance trees were constructed using the NEIGHBOR program in PHYLIP(Felsenstein 2004) under the JTT amino acid substitution model(Jones et al. 1992) and a gamma distribution of rate variation among sites using the alpha parameter estimated in the Bayesian analysis. Trees produced by the two methods were similar, although the Bayesian method produced trees with higher resolution and branch confidence values. Accession numbers of sequences and species abbreviations used for phylogenetic analysis are listed in the supplementary data (Tables 1, 2, 3, 4) of (Glasner, Fayazmanesh et al. 2006). In general, species names are abbreviated using the first three letters of the genus and first two letters of the species. The strain is indicated if multiple strains of the same species were used in the analysis, and *Bacteroides* is abbreviated with ‘‘Bct’’ to avoid confusion with *Bacillus*.

Sequence analysis

Sequence conservation was analyzed by comparing the aligned OSBS/NAAAR, MLE, and AEE families. Family assignments of MLE and AEE proteins were taken from the SFLD, which uses HMMs and information from the literature to assign proteins to families. Conserved positions were defined as those in which >90% of family or

subfamily members have the same amino acid residue. Phenylalanine and tyrosine or aspartate and glutamate were treated as equivalent. Conserved residues were mapped onto the structures of 1FHV (E. coli OSBS) and 1SJB (Amycolatopsis OSBS/NAAAR) in Chimera.(Pettersen et al. 2004)

Structural analysis

Structural superpositions of the whole proteins, capping domains, and barrel domains of 1SJB, 1FHV, B. bacteriovorus OSBS, and 1MUC were generated from the structure-based sequence alignment of the MLE subgroup using the Match feature of Chimera or Combinatorial Extension (CE)(Shindyalov et al. 1998).

2.2.3. Results

Summary of phylogenetic analysis results

To understand the evolution of the OSBS/NAAAR family, we began by identifying species which must have OSBS activity. We identified 127 strains in which at least five of the eight menaquinone pathway genes could be identified (Figure 2.3). In organisms in which most menaquinone pathway genes were identified, some or all are colocalized in the genome and are likely to be coregulated as operons.

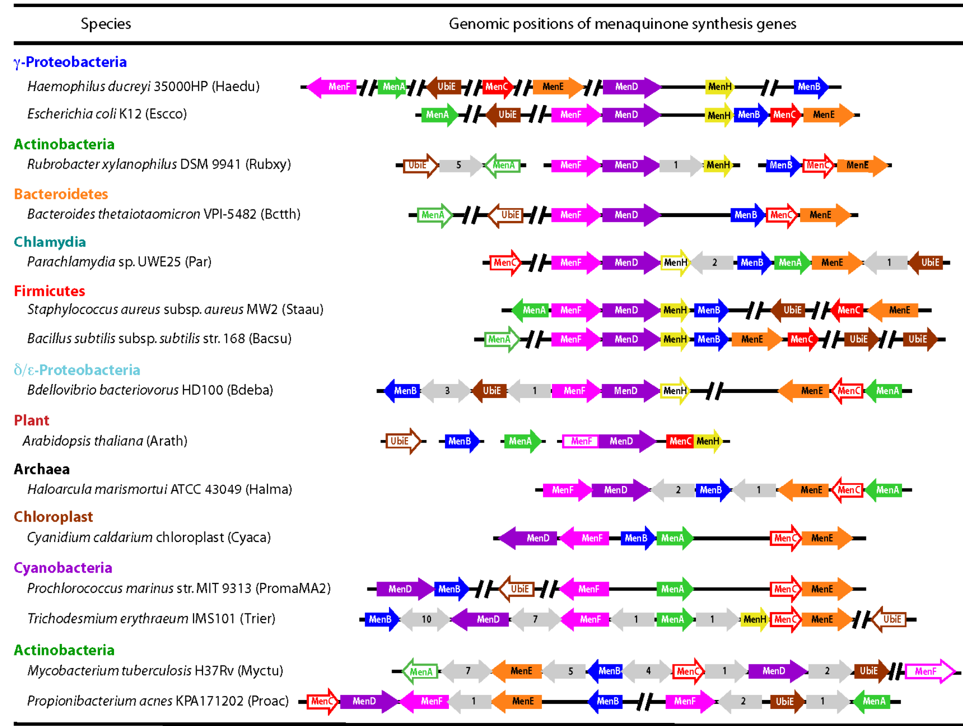


Figure 2.3. Genomic context of menaquinone biosynthesis genes.

All identified menaquinone synthesis genes are shown as arrows; hollow arrows indicate provisional assignments, as defined in Section 2.2.2. Menaquinone synthesis genes have been aligned to show similarities in gene order; as a result, spaces between genes are not proportional to the length of the DNA separating the genes. Each horizontal segment indicates a contiguous DNA segment. The genomes of some species have multiple chromosomes or have not been completely assembled, as indicated by gaps between segments. Hash marks indicate an intervening region encoding > 40 genes. Smaller intervening regions are shown as light grey arrows with the number of intervening genes and their orientation on the chromosome indicated.

The difficulty of unequivocally identifying OSBSs based on sequence similarity and genome context is in agreement with the observation of Palmer et al. that OSBSs are extremely divergent and can share < 15% identity(Palmer et al. 1999). In fact, some putative OSBSs are barely recognizable as enolase superfamily members. For instance, sequence similarity searches using the OSBS from *Bdellovibrio bacteriovorus* as a query

identifies another very divergent, putative OSBS as the best match, but the E-value (0.05) is barely significant. Thus, we speculated that OSBS activity might have evolved multiple times within the enolase superfamily. To investigate this hypothesis and to understand how the NAAAR-like proteins from organisms lacking menaquinone are related to OSBS, we examined the phylogeny of a subset of the enolase superfamily comprised of 288 sequences which includes all OSBS candidates, the rest of the MLE subgroup, and any other enolase superfamily members which could not be assigned to a subgroup or family by Hidden Markov Models (HMMs) created to describe OSBS and other enolase superfamily members in the Structure-Function Linkage Database (SFLD) (Pegg et al. 2005; Pegg et al. 2006). Contrary to our hypothesis, the phylogenetic tree of a representative subset of these sequences demonstrated that all OSBSs and NAAAR-like proteins are included in a single clade (Figure 2.4). Although the resolution at many interior nodes is low, the branch confidence value separating the OSBS/NAAAR family from the rest of the MLE subgroup is 1.00. This result confirms that the OSBSs identified by sequence similarity and genomic context, including those that are too divergent to match the MLE subgroup HMM and those that are not encoded near other menaquinone pathway genes, belong to the OSBS/NAAAR family. In addition, this result strongly suggests that this family had a single evolutionary origin, because rooting the tree with MLE or AEE, the closest known paralogs of the OSBS/NAAAR family(Babbitt et al. 1996), leaves the family as a monophyletic group.

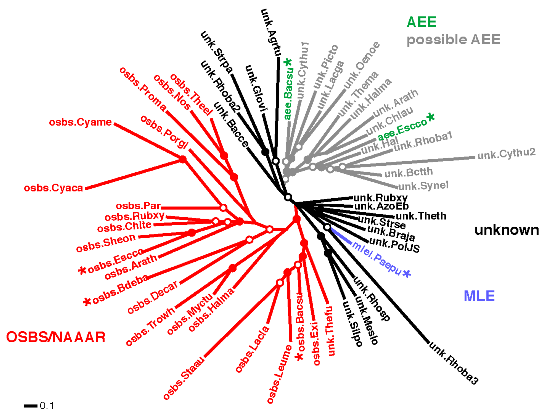


Figure 2.4. Bayesian phylogenetic tree of proteins in the MLE subgroup.

A representative set of 54 proteins was selected from the 288-protein subgroup by using only proteins sharing < 40% identity. The predicted or verified function is indicated by the prefix "osbs", "aee", or "mleI", and characterized proteins are indicated with an asterisk (*). Proteins of unknown function are prefixed by "unk". OSBS/NAAAR family members are shown in red, characterized AEEs are in green, and MLE I is in blue. Other possible AEEs are in gray, but they cluster with the characterized AEEs with only moderate statistical support. Proteins of unknown function are in black. Branch confidence values are indicated as solid circles (≥ 0.95), hollow circles (0.7-0.94), or no indication (0.5-0.7).

Diversity in the OSBS/NAAAR family

Having performed a comprehensive survey of the distribution of the OSBS/NAAAR family, we were interested in reevaluating the family's diversity to discover whether it is unusually divergent compared to other protein families, as suggested previously (Palmer et al. 1999). Initially, we compared lengths of OSBS/NAAAR family trees to tree lengths of other families in the menaquinone pathway

or enolase superfamily. Tree length (measured as substitutions per site) is expected to be the most accurate measure of sequence divergence, because it corrects for multiple substitutions per site. In comparisons of trees built using sequences from the same set of species, the length of OSBS/NAAAR trees were usually at least twice as long as those of other protein families, indicating that the OSBS/NAAAR family has indeed evolved at a much faster rate (data not shown). However, the topology of the OSBS/NAAAR tree was similar but rarely identical to the topology of trees built using other families, even when using subsets of the OSBS/NAAAR family that are well resolved on the phylogenetic tree.

Because the significance of comparing lengths of trees that have different topologies is uncertain, we also calculated pairwise percent sequence identities, even though these are a more approximate measure of evolutionary distance. Comparison of OSBSs and menBs from a wide taxonomic distribution agree well with those previously reported, with menB proteins generally sharing > 40% identity while OSBSs from the same set of species generally share < 30% identity (Palmer et al. 1999). To gain a better perspective concerning the divergence of the OSBS family, we compared minimum and average percent identities of the OSBS family to other families in the enolase superfamily and menaquinone pathway (Table 2.1). For each comparison, the set of OSBSs and the set of proteins from the compared family were taken from the same set of species. Compared to other families in the enolase superfamily, the OSBS family is unusually divergent. However, comparison to other proteins in the menaquinone pathway reveals a different picture. Although MenB is extremely well-conserved, the sequence divergence of MenD and MenE is more similar to OSBS. On average, the OSBS family is slightly

more divergent than the MenD or MenE families, but because percent identity is only a rough approximation of evolutionary distance, it is unclear whether the OSBS family is significantly more divergent than these proteins. Thus, although the OSBS family is unusually divergent for the enolase superfamily, it is less extraordinary compared to other proteins in its pathway.

Table 2.1. Relative divergence of the OSBS family

Family for comparison	Number of species[a]	Compared family[b]		OSBS[b]	
		Average % identity	Minimum % identity	Average % identity	Minimum % identity
Enolase[c]	66	56	27	26	15
Galactonate dehydratase[c]	8	55	32	31	20
Glucarate dehydratase[c, d]	11	78	66	45	20
AEE[c]	30	38	24	33	18
MenB	67	58	35	26	14
MenD	66	32	21	26	14
MenE	67	27	14	26	14

[a]OSBSs were compared to proteins from a second family which were taken from the same set of species as the OSBSs.

[b]Percentage identities were calculated as number identical/length of the longer sequence from pair-wise alignments generated by ALIGN.83. E.W. Myers and W. Miller, Optimal alignments in linear space, Comput. Appl. Biosci. 4 (1988), pp. 11–17. View Record in Scopus | Cited By in Scopus (432)83

[c]Some NAAAR-like proteins not encoded in menaquinone operons are included in the OSBS family.

[d]Glucarate dehydratase related protein, which has an unknown function was excluded.

In addition to being more divergent than other families in the enolase superfamily, the OSBS/NAAAR family is unusual in that it includes proteins catalyzing at least two different reactions. Surprisingly, the NAAAR-like proteins are not among the more divergent proteins in the family, but are closely related to proteins identified as OSBS based on genomic context and experimental evidence(Palmer et al. 1999). As shown above, phylogenetic analysis failed to separate the NAAAR-like proteins into a separate

clade. In fact, most NAAAR-like proteins which are not encoded in menaquinone operons share > 40% identity with *B. subtilis* OSBS. Only the genomic position of the genes encoding NAAAR-like proteins hints that their function might differ from the menaquinone operon-encoded OSBSs.

Conservation of sequence in the OSBS/NAAAR family

Despite the high sequence divergence of the OSBS/NAAAR family, all proteins in the family form a single clade in the MLE subgroup phylogenetic tree, indicating that there must be conserved sequence information that differentiates this family from the rest of the MLE subgroup. To identify conserved residues specific to the OSBS/NAAAR family, we compared the pattern of sequence conservation among the OSBS/NAAAR, MLE, and AEE families. For this analysis, the OSBS/NAAAR family was treated as a single unit or divided into subfamilies representing clades containing at least five sequences (γ-Proteobacteria, Cyanobacteria, Bacteroidetes, Actinobacteria, and Firmicutes/NAAAR-like proteins), as indicated in Figure 2.5. Except for unk.Thefu (gi23018694 from *Thermobifida fusca*), the NAAAR-like proteins were included with the Firmicute OSBSs because they could not be cleanly separated based on phylogeny or the presence of the menaquinone operon. In addition, the AEEs were divided into two groups comprised of close relatives of characterized *E. coli* or *B. subtilis* epimerases because the clade including both groups had poor statistical support on the MLE subgroup phylogenetic tree (Figure 2.4).

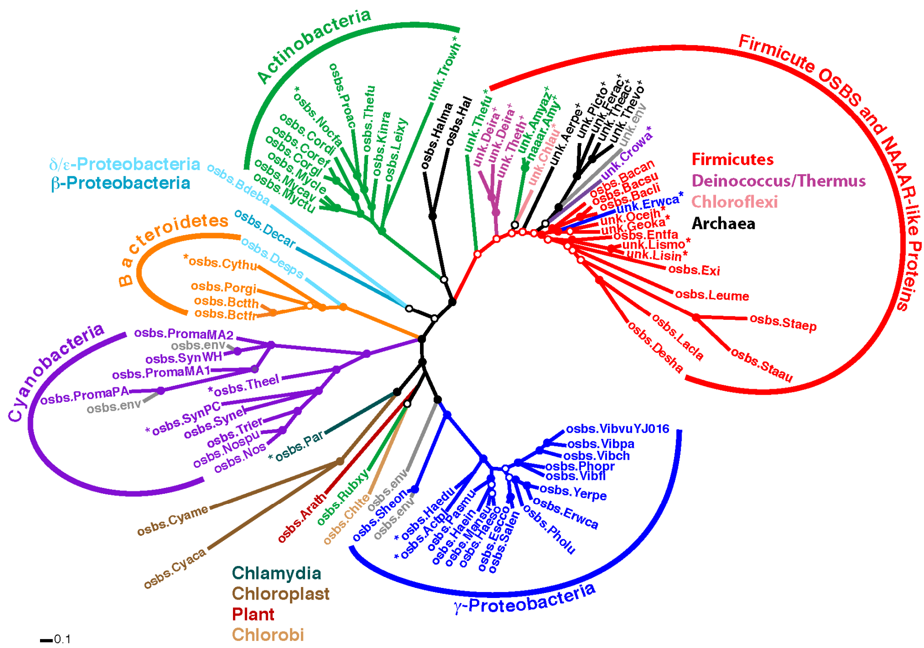


Figure 2.5. Bayesian phylogenetic tree of the proteins in the OSBS/NAAAR

Branch confidence values are shown as in Figure 2.4. A) The OSBS/NAAAR family. To build the tree, the full set of OSBS/NAAAR proteins was filtered to remove proteins sharing > 94% identity with any other in the set. Proteins are colored according to phylum, and arcs indicate the main subfamilies. Proteins in gray are environmental sequences derived from the Sargasso Sea data set (Venter et al. 2004). A plus sign (+) indicates NAAAR-like proteins found in strains in which menaquinone synthesis genes could not be identified. An asterisk (*) identifies proteins which are not encoded in menaquinone operons but are found in strains which have the menaquinone pathway.

The pattern of sequence conservation is summarized in Figure 2.6, in which residues conserved in > 90% of subfamily members are highlighted in magenta, and residues conserved in both > 90% of the subfamily and > 90% of the entire MLE subgroup are highlighted in black. The only residues conserved throughout the entire

MLE subgroup are the catalytic residues in the barrel domain, except for the lysine on barrel domain strand β6 (Bar- β6) which is replaced by tyrosine or arginine in some MLE subgroup members, including one branch of the Cyanobacteria OSBS subfamily. For these Cyanobacteria OSBSs, an arginine at this position might have little effect on catalysis, because the lysine at this position in *E. coli* OSBS appears to stabilize the enediolate intermediate rather than act as a general acid/base catalyst(Klenchin et al. 2003). The other highly conserved residues in the MLE subgroup appear to be involved in maintaining the structure. For instance, the conserved elements of capping domain strand β3 and helix α3 (Cap-β3 and Cap-α3) are adjacent and probably important for capping domain structure, and the glycine before Bar-β6 is located in a tight turn. Other than these residues, the pattern of sequence conservation is somewhat variable. Although some groups appear to have greater numbers of conserved residues, this is mostly because these groups are small (e.g. the Bacteroidetes group) or include sequences of limited diversity (e.g. MLE and AEE groups, in which sequences share > 40% identity). In comparison, the Firmicutes/NAAAR-like subfamily includes more divergent sequences; it should be noted that the most divergent sequences in this group (osbs.Staau, osbs.Staep, osbs.Lacla, osbs.Desha, osbs.Leume, and osbs.Exi) are menaquinone operon-encoded OSBSs, not NAAAR-like proteins.

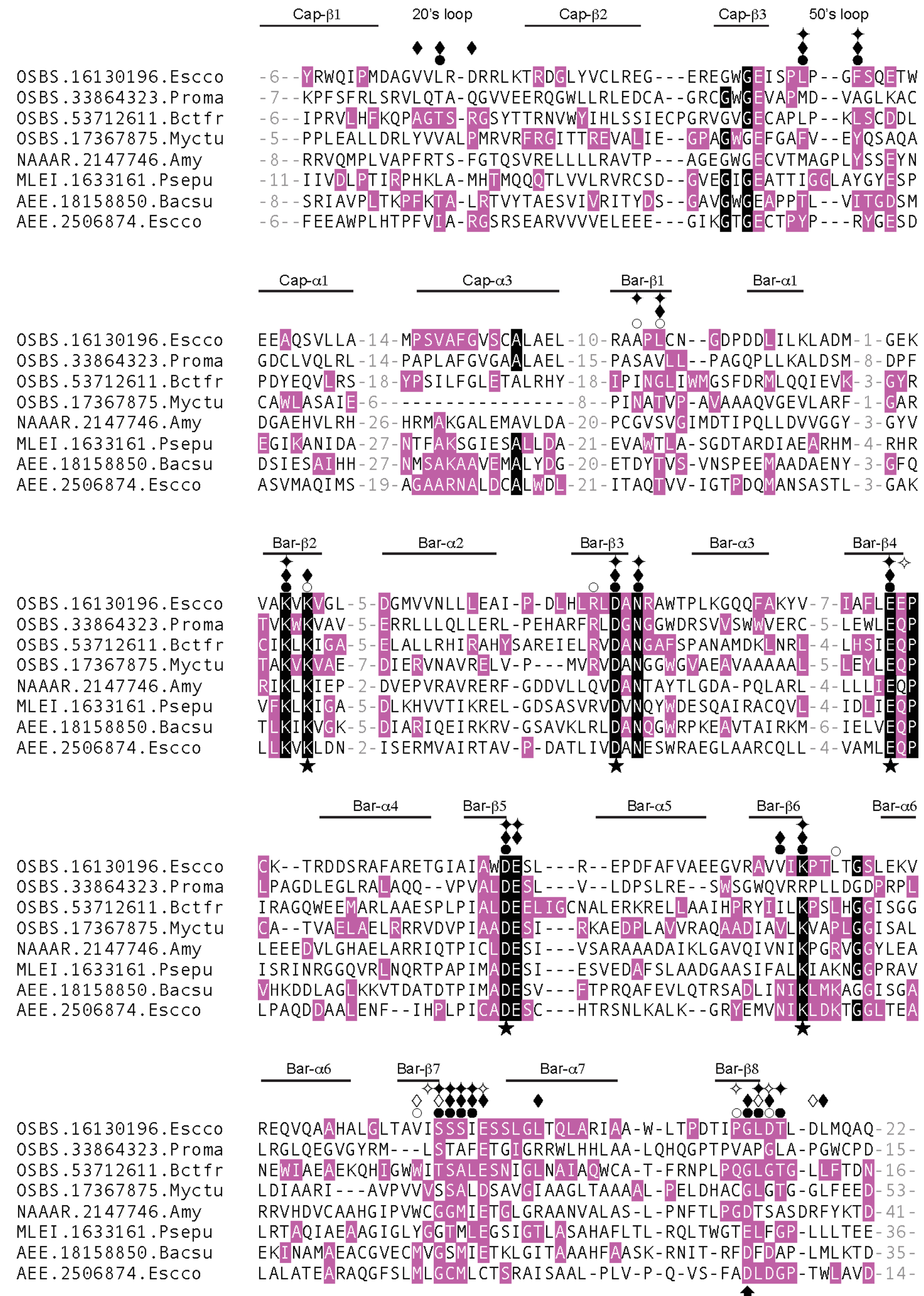


Figure 2.6. Analysis of sequence conservation in the OSBS/NAAAR family

The sequence alignment shows representatives of each of the five OSBS/NAAAR subfamilies, the MLE family, and two AEE subfamilies. The membership of each OSBS/NAAAR subfamily is shown in Figure 2.5, as indicated by the arcs, with the exception that the NAAAR-like *T. fusca* protein (unk.Thefu) was not included in this analysis. γ-Proteobacteria is represented by

OSBS.16130196.Escco, Cyanobacteria by OSBS.33864323.Proma, Bacteroidetes by OSBS.53712611.Bctfr, Actinobacteria by OSBS.17367875.Myctu, and the Firmicute/NAAAR-like protein subfamily by NAAAR.2147746.Amy. The membership of the AEE subfamilies and the MLE family consists of proteins sharing > 40% identity with each sequence that is shown. Magenta residues indicate conservation in > 90% of subfamily members, and black residues indicate conservation in both > 90% of the subfamily and > 90% of the entire MLE subgroup. Gray numbers indicate the length of segments that are not shown. Secondary structure of the capping and barrel domains are indicated by Cap- and Bar-, respectively. Catalytic residues are indicated by a five-pointed star below the sequences. Positions of residues lining the active site pocket are indicated for *E. coli* OSBS (●), *Amycolatopsis* OSBS/NAAAR (♦), and *B. bacteriovorus* OSBS (✦, sequence not shown). Solid symbols represent residues < 5 Å away from bound OSB, and open symbols indicate residues 5-6 Å away from the ligand. The arrow indicates the position of the glutamate or aspartate to glycine mutation that confers OSBS activity on *E. coli* AEE or *Pseudomonas* sp. P51 MLE II(Schmidt, Mundorff et al. 2003).

Surprisingly, the results of this analysis indicate that there are no conserved residues shared by all five OSBS/NAAAR subfamilies, other than residues also shared with the rest of the MLE subgroup. Conserved residues within subfamilies are most likely to fall in regions near the active site, either on two loops of the capping domain or on the strands or loops of the barrel domain. Although one or more OSBS/NAAAR subfamilies often has conserved residues at the same position, the identities of those residues are rarely the same. In cases where the residue identity is conserved, the same residue is often present in the MLE or AEE families. Thus, although the OSBS/NAAAR family is phylogenetically unified and most, if not all (including characterized NAAAR-like proteins) catalyze the OSBS reaction, there are no unique OSBS/NAAAR family motifs to differentiate them from other MLE subgroup members.

Summary of structural analysis results

To understand how substrate specificity is conserved with so little sequence conservation, we compared the structures of *E. coli* OSBS bound to the substrate or OSB (1FHV and 1R6W), *Amycolatopsis* OSBS/NAAAR bound to OSB (1SJB), and *B. bacteriovorus* OSBS bound to OSB (coordinates generously provided by Alexander Fedorov, Elena Fedorov and Dr. Steven Almo, Albert Einstein College of Medicine)(Thompson et al. 2000; Klenchin et al. 2003; Thoden et al. 2004). In all three structures, residues lining the active site pocket are in homologous positions, and these residues tend to be more highly conserved within and between subfamilies than regions distant from the active site (Figure 2.6). The structures exhibit similar hydrophobic interactions between the benzene ring of OSB and the 50s loop, in which at least one of the residues interacting with ligand is aromatic. Most members of the OSBS/NAAAR family (and many other members of the MLE subgroup) have aromatic residues at one or both positions, suggesting that this hydrophobic pocket is important for ligand binding.

In contrast to these similarities, there are also some striking differences in active site structure, which might contribute to differences in function and inherent evolvability. As previously reported, the conformation of OSB differs in the *Amycolatopsis* and *E. coli* enzymes(Thoden et al. 2004). In *Amycolatopsis*, the succinyl tail of OSB is extended, while it is bent in *E. coli* and *B. bacteriovorus* (Figure 2.7). Likewise, the succinyl or acetyl moieties of N-acylamino acid substrates also lie in extended conformations in *Amycolatopsis* OSBS/NAAAR. For N-succinyl-methionine, this conformation provides suitable hydrogen bond donors and acceptors, which are unavailable in *E. coli* OSBS,

accounting for the inability of *E. coli* OSBS to racemize this substrate(Thoden et al. 2004).

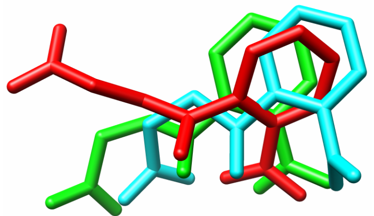


Figure 2.7. Comparison of OSB binding orientation

Amycolatopsis OSBS/NAAAR (1SJB) is red, *E. coli* OSBS (1FHV) is cyan, and *B. bacteriovorus* OSBS is green.

The second major difference among these structures is the position of the 20s loop (Figure 2.8, top). In spite of its proximity to the active site, the 20s loop is poorly conserved within and between different subfamilies. The lack of conservation might be explained by the necessity of compensatory mutations to accommodate other structural changes, such as shifts in the orientation between the two domains, although there might also be consequences for the catalytic activity (see below). In *Amycolatopsis* OSBS/NAAAR bound to OSB, the 20s loop contacts the catalytic lysine that acts as a general base (the second lysine in the KXK motif), sandwiching it between the loop and the barrel and orienting it appropriately for proton abstraction. In contrast, the 20s loop of *E. coli* OSBS bound to either substrate or product does not contact the barrel, leaving the active site slightly open and the catalytic lysine disordered and solvent accessible. Similarly, the catalytic lysine is also solvent accessible in *B. bacteriovorus* OSBS, although the 20s loop is disordered, even when OSB is bound (data not shown).

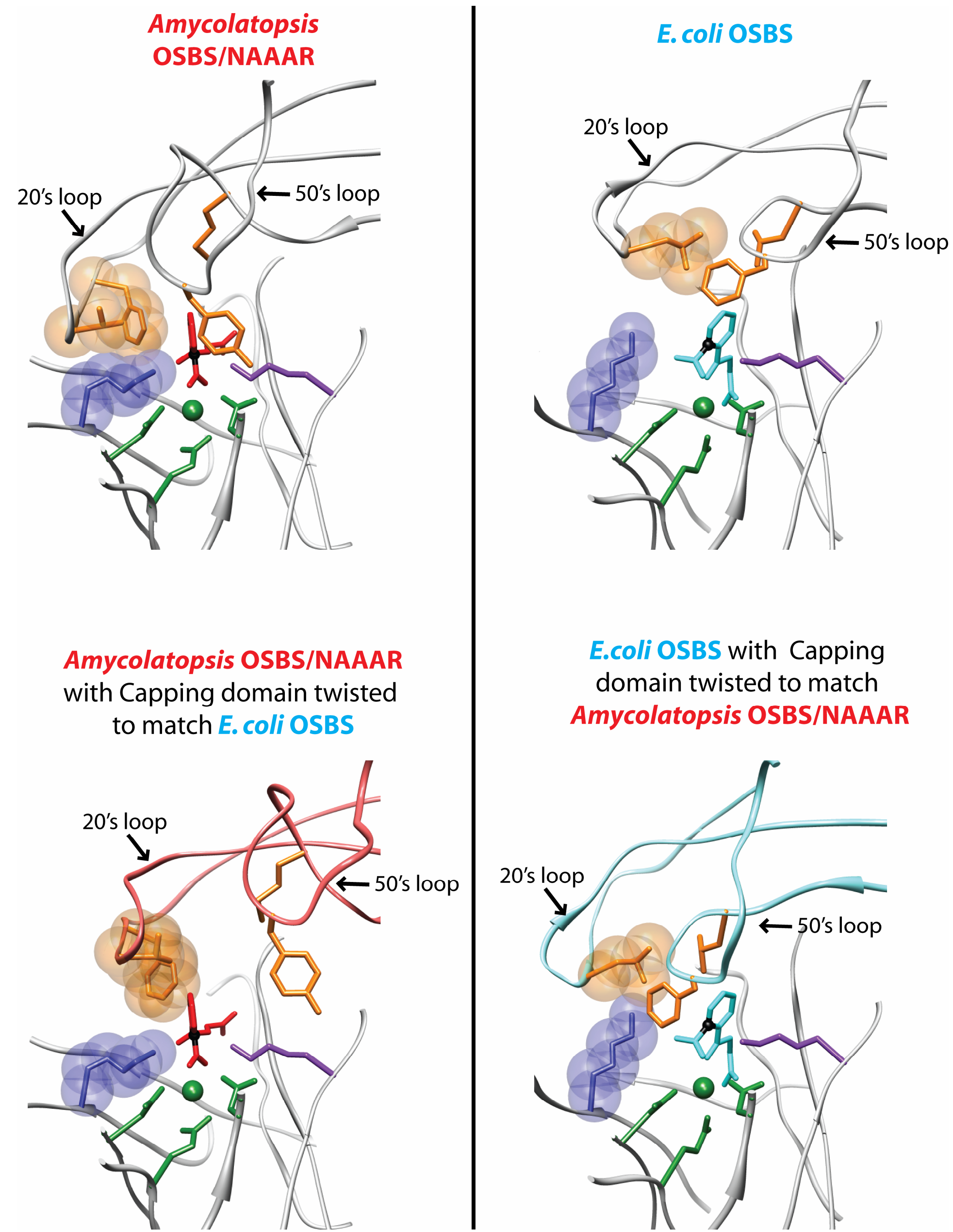


Figure 2.8. Comparison of the 20s and 50s loop positions in *E. coli* OSBS and *Amycolatopsis* OSBS/NAAAR

The native structures are shown in the top panels. In the bottom panels, the capping domain of *Amycolatopsis* OSBS/NAAAR has been rotated to match the position of the *E. coli* OSBS capping domain (left), and the *E. coli* OSBS capping domain has been rotated to match the *Amycolatopsis* OSBS/NAAAR capping domain (right). Metal binding residues and the metal ion are shown in green, the Bar-β2 lysine that acts as the general base is shown in blue, the Bar-β6 lysine required

for catalysis is purple, and residues on the 20s and 50s loops that contact the ligand are in orange. The carbon from which the proton is abstracted is shown in black.

We hypothesize that these structural differences might contribute to differences in binding specificity and catalysis among these enzymes, as well as to their capacities to evolve new functions, as discussed below.

In order to understand the consequences of domain orientation on the structure of the active site and the function of the enzymes, we analyzed the effect of twisting the *E. coli* OSBS capping domain to match the orientation of the *Amycolatopsis* OSBS/NAAAR capping domain (Figure 2.8, bottom). To do this, the capping and barrel domains were superimposed separately on the *Amycolatopsis* enzyme. Twisting the *E. coli* capping domain shifts the 20s and 50s loops ~ 6 Å down toward Bar-β2. As a result, the 20s loop is no longer in contact with the ligand. Instead, it now approaches the catalytic lysine of the KXK motif, which is disordered in the E. coli structures. Having the 20s loop in this position would prevent this lysine from adopting an extended conformation, possibly forcing it into the active site toward the substrate. When the converse experiment is performed and the *Amycolatopsis* capping domain is twisted to match that of E. coli, the 20s and 50s loops shift ~ 6 Å away from the barrel so that the 50s loop is no longer in contact with the ligand. In this position, the 20s loop barely contacts the second lysine of the KXK motif, leaving it mostly exposed to solvent outside the active site.

Although we have only shifted the orientations of the two domains and have not refined the models to ameliorate steric hindrances or reposition loop residues into more favorable conformations, these results suggest that proper orientation of the capping and barrel domains is required for positioning the catalytic lysine for catalysis in

Amycolatopsis OSBS/NAAAR. For *E. coli* OSBS, these results suggest two possibilities. First, perhaps the flexible lysine is resident in the active site often or long enough for catalysis. Second, it is also conceivable that the crystal structures of *E. coli* OSBS bound to either substrate or product do not capture the structure of the enzyme in the transition state. As in *Amycolatopsis* OSBS/NAAAR, repositioning the 20s loop through domain rotation or other conformation changes might be required in order to correctly position the lysine for catalysis. The fact that the 20s loop is disordered in *B. bacteriovorus* OSBS in the presence of ligand provides some support for the latter possibility.

2.2.4. Discussion

Changes in protein structure during evolution

Investigating the evolutionary relationships among the OSBS and NAAAR-like proteins of the enolase superfamily uncovered several surprising observations. The most remarkable are that these proteins exhibit significant structural variation and that sequence motifs unique to the OSBS/NAAAR family which distinguish it from other families in the enolase superfamily could not be identified, in spite of the fact that OSBS activity has been conserved and the family appears to have a single evolutionary origin.

This raises the question of how enzyme specificity can be maintained over the course of evolution. Some structural differences would be expected between *Amycolatopsis* OSBS/NAAAR and the other two OSBSs, since the *Amycolatopsis* enzyme has an additional activity. However, structural differences as exemplified by both RMSD and domain orientation are at least as great between *E. coli* and *B. bacteriovorus* OSBSs. One way in which specificity might be maintained during evolution is through compensatory mutations and structural flexibility of surface loops that close the active

site (James et al. 2003). In the three OSBS/NAAAR family structures, the function of the 50s loop appears to be conserved, since it is structurally well-aligned and forms a hydrophobic binding pocket for the benzene ring (Figure 2.6). The ring is anchored at one end by the carboxyl group binding to the metal ion and by the 50s loop at the other. Mutations that affect the orientation of the benzene ring could be accommodated by structural reorganization and mutations of the 50s loop, such as the small insertion observed in the *Amycolatopsis* enzyme.

The 20s loop is also likely to play an important role in maintaining, and perhaps altering enzyme specificity. In most enolase superfamily members, this loop is disordered in the absence of ligand (Lebioda et al. 1988; Neidhart et al. 1991; Landro et al. 1994; Wedekind et al. 1994; Gulick et al. 2000; Thompson et al. 2000). In addition to being less well-conserved than the 50s loop, the 20s loop is not well-aligned in the structures of *Amycolatopsis* OSBS/NAAAR and *E. coli* OSBS bound to OSB, and it is disordered in *B. bacteriovorus* OSBS bound to OSB. The flexibility and apparent mutability of this loop suggest that it could have coevolved with other sequence and structure elements (such as those determining domain orientation) to maintain substrate binding. In addition, the flexibility of this loop might allow promiscuous binding and reactions with new substrates without impairing OSBS activity, leading to the evolution of new protein functions, such as NAAAR activity(James et al. 2003; Aharoni et al. 2005).

While the role of flexible loops in maintaining OSBS activity is somewhat speculative, it has also been proposed that structural requirements for catalysis are relatively permissive because the OSBS reaction is highly exergonic and can proceed uncatalyzed at significant rates(Palmer et al. 1999; Taylor et al. 2001). In all three

OSBS/NAAAR family structures, interactions with OSB are largely hydrophobic, and most hydrogen bonds are formed with water or residues conserved in the whole MLE subgroup (Alexander Fedorov, Elena Fedorov and Dr. Steven Almo, unpublished)(Thompson et al. 2000; Thoden et al. 2004). Thus, it appears that interaction with subgroup-conserved residues is sufficient for correctly orienting the substrate for catalysis, and the only additional requirement is a hydrophobic cavity of an appropriate size and shape. Additional evidence for this is supplied by single point mutations in *Pseudomonas* sp. P51 MLE II and *E. coli* AEE which confer OSBS activity on these enzymes(Schmidt, Mundorff et al. 2003). These mutations are located at the same position in Bar-β8 and exchange an aspartate or glutamate for a glycine, creating space to accommodate the succinyl tail of OSB if it is bound in the same conformation as in *E. coli* OSBS (Figure 2.6 and Figure 2.7).

Ramifications for structure and function prediction in genomics

Two important contributions of genomics are to correctly annotate protein functions and identify proteins of unknown structure and function whose characterization will enhance biological understanding. As noted previously and shown here, simple sequence metrics are often inadequate for predicting protein function(Rost 2002; Tian et al. 2003). Perusal of GenBank annotations of the OSBS/NAAAR family reveals that only 60% are correctly annotated (43% excluding proteins misleadingly annotated as "*o*-succinylbenzoate-CoA synthases"). While only 7% of these annotations are completely incorrect, the remainder are incomplete or somewhat misleading, often assigning OSBS/NAAAR proteins to the wrong family or subgroup of the enolase superfamily. For example, several proteins are incorrectly annotated as muconate or chloromuconate

cycloisomerases. Many others are annotated as "COG4948: L-alanine-DL-glutamate epimerase and related enzymes of enolase superfamily", which correctly relates them to the MLE subgroup but also implies an incorrect function.

Functional annotation of the OSBS/NAAAR family is difficult for two reasons. First, some members of the family are so divergent that sequence similarity cannot be used to distinguish them. Outliers such as the *B. bacteriovorus* OSBS could only be identified using a combination of genomic context, phylogenetic analyses, and ultimately experimental validation. Second, the NAAAR-like proteins could not be separated from the OSBSs based on sequence similarity or position in the phylogenetic tree. Instead, their main characteristics are that they are closely related to *Amycolatopsis* OSBS/NAAAR and they are not encoded in menaquinone operons.

Given such complexities, it is not surprising that automated annotation methods have had so much difficulty with this family. The orthogonal information furnished by phylogenetic reconstruction and analysis of genome context not only provides stronger confidence in functional annotation, but it is also invaluable for identifying proteins whose functions cannot be predicted with certainty. Similarly rigorous application of these methods will probably be required for accurate annotation of other protein families which exhibit high sequence, structural, and functional divergence.

Detailed studies of the sort undertaken here are also useful for identifying candidates for experimental characterization and structural genomics projects. Not only is there significant functional diversity in the OSBS/NAAAR family, but we also discovered significant structural variation among the family's three crystallized members. As discussed above, it is expected that several other subfamilies, especially the

Actinobacteria subfamily, also exhibit structural variations. Solving the three-dimensional structures of representatives of other subfamilies will be valuable for understanding allowable variations in protein-substrate interactions in isofunctional proteins. In addition, our current and future studies of the structure and function of the NAAAR-like proteins will help elucidate how new protein functions evolve. Although our strategy is more labor-intensive than purely automated methods of target selection for structural genomics projects, it provides more context for understanding structure-function relationships and evolutionary mechanisms.

Concluding Remarks

Our analysis of the OSBS/NAAAR family revealed several insights into how protein function and structure evolve. First, highly divergent protein families can exhibit significant structural variations. Second, enzyme specificity can be maintained in spite of limited sequence conservation among ligand-contacting residues. Third, new activities can evolve through promiscuous intermediates, and there might be structural features of proteins that make them more or less prone to evolve promiscuous activities. Few analyses of protein structure, function, and evolution have been performed in this depth; thus, extending these studies to other protein families will be important for testing the generality of these conclusions.

2.3. Non-active Site Determinants of Enzyme Specificity

2.3.1. Introduction

Traditionally, efforts to study enzyme specificity and evolution have been focused on the active site region to explain how enzyme function has evolved. Our goal is to

investigate whether there are non-active site regions that are important for function and important for maintaining or altering function through evolution. Here, we apply evolutionary trace (ET) to examine how specificity may have evolved in the enolase superfamily, a group of related but diverse enzymes that share a common mechanistic step. The current view is that the C-terminal TIM $(\beta/\alpha)_8$ barrel domain mediates the common mechanistic step shared by the entire superfamily, and that the N-terminal capping domain determines the variation in specificity seen in the superfamily (Gerlt et al. 2003). However, these inferences are based on anecdotal observations. Structural analysis in the OSBS family (Section 2.2) suggests that domain orientation and both the 20s and 50s loops play a role in the substrate specificity. If these loops are important for mediating function and determining substrate specificity, we might expect to find function-specific sequence signatures in these loop regions. If domain orientation is important for determining substrate specificity, domain interface residues might be important for maintaining the proper domain orientation and we might expect to find function-specific sequence signatures in these domain interface regions. We test these ideas by examining the location of residues associated with specificity in relation to the two domains, the active site and active site loops, and the interdomain regions.

2.3.2. Methods

Sequences and multiple sequence alignment

The sequences and the quality of the multiple sequence alignment used for an evolutionary trace analysis will greatly affect the results. Therefore, we used sequences and a hand-curated alignment of the enolase superfamily from the Structure-Function Linkage Database (Pegg et al. 2006). The alignment includes sequences that are

experimentally characterized and sequences that, based on classification using curated Hidden Markov Models (HMMs), are predicted with high certainty to be members of the enolase superfamily (Brown et al. 2006). The alignment was generated using ClustalW and then manually refined using structural superpositions and to ensure that superfamily-conserved functionally important residues are aligned properly.

Evolutionary trace

The general methodology for evolutionary trace (ET) is described in Section 2.1.1. To find residues that determine functional and substrate specificity, the classes for the ET analysis were based on SFLD family classifications (Pegg et al. 2006). In the SFLD, sequences are grouped into isofunctional families based on their experimentally characterized function or, for uncharacterized sequences, based on how well they matched family-specific HMMs. We focused on the muconate lactonizing enzyme II, dipeptide epimerase, and the OSBS families. We divided the OSBS family into enzymes that only performed the OSBS function and enzymes that are also able to promiscuously catalyze the NAAAR reaction. We then found residues that were conserved within a class (a residue is present in > 90% of sequences within a class) and not conserved across the whole superfamily. The Python script for performing this analysis is included in Appendix A.

Identification of ligand-binding residues

Ligand-binding residues are defined based on annotations in LigBase, a database of ligand binding sites in protein structures (Stuart et al. 2002). A residue is defined as being in the binding site when at least one atom in that residue is within 5 Å of a ligand bound to the structure.

Identification of domain interface residues

PIBASE (Davis et al. 2005) is a database of interacting protein domain pairs and properties of their interfaces. We use the definition of domain interface residues from PIBASE for our analyses.

2.3.3. Results

We examined where the class-specific residues fall in the structures with respect to the 20s and 50s loops (Figure 2.9). We find very few class-specific residues in the 20s loop. In the dipeptide epimerase family, there are four residues that are class-specific, but there are no residues that are class-specific and in the 20s loops in the other three families. There are a few more class-specific residues in the 50s loop than in the 20s loop, but this region does not have the highest concentration of class-specific residues. The regions that have the most class-specific residues are highlighted as Regions 1, 2, and 3 in Figure 2.9.

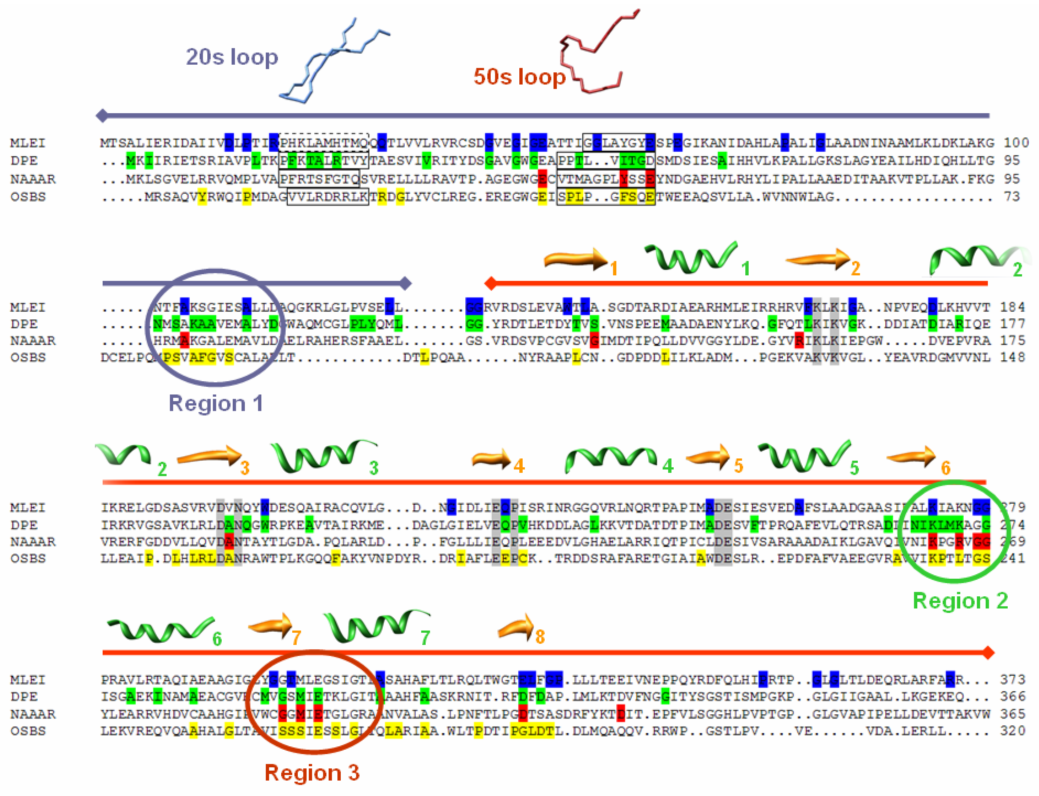


Figure 2.9. Evolutionary trace for four families in enolase superfamily.

Residues in blue are class-specific for the muconate lactonizing enzyme (MLEI) family. Residues in green are class-specific for the dipeptide epimerase (DPE) family. Residues in red are class-specific for N-acylamino acid racemases. Residues in yellow are class-specific for the *o*-succinyl benzoate synthase (OSBS) family. Residues in grey are conserved in the whole superfamily. Residues that are in the 20s and 50s loops are outlined in black. These loops and other secondary structure elements are shown above the sequence alignment. The N-terminal capping domain is indicated with the blue horizontal line above each section of the multiple sequence alignment and the C-terminal barrel domain with a red horizontal line.

Displaying these three regions on a representative structure from the superfamily shows that these regions are concentrated in the interface between the N-terminal capping domain and the C-terminal barrel domain (Figure 2.10). Region 1 is on an alpha helix of the N-terminal capping domain that faces the barrel domain. Regions 2 and 3 are on the

C-terminal end of the barrel, on two loops between beta strands and alpha helices of the barrel.

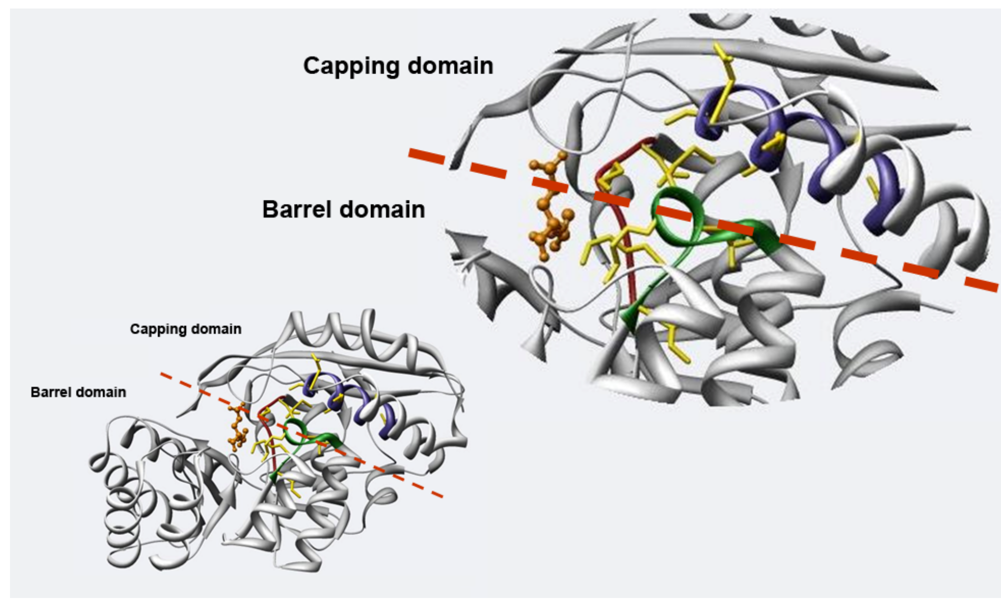


Figure 2.10. Class-conserved residues in domain interface.

Class-conserved residues that occur in Regions 1, 2, and 3 are displayed in yellow. The three regions with the highest concentration of class-conserved residues are shown in blue (Region 1), green (Region 2), and red (Region 3) and these correspond to the regions highlighted in Figure 2.9

To verify the visual observations, we compared the fraction of class-specific residues observed in a given region to what would be expected by chance (the fraction of all residues in that region) (Table 2.2). We focused on two regions of the structures – near the ligand (based on LigBase assignments) and near the domain interface (based on PIBASE assignments). Class-specific residues occur more often than expected by chance in the regions near the active site as well as near the interface region. The frequency of class-specific residues occurring near the ligand varies between the four families, with NAAAR/OSBS enzymes having the highest frequency and the OSBS enzymes having the lowest frequency. In contrast, the frequency of class-specific residues in the interface

region, though more than what would be expected based on previous reports of the superfamily, does not vary greatly between families.

Table 2.2. Location of class-specific residues

	Near Ligand (from LigBase)				In Interface (from PIBASE)			
	# cs res near ligand	# cs res	% cs res near ligand	% cs_ligand/ % ligand	# cs res in interface	# cs res	% cs res in interface	% cs_interface/ % interface
MLE	1	42	2.38%	**1.22**	23	42	54.76%	**1.44**
	7 of 360 (1.94%) residues in active site				137 of 360 (38.06%) residues in interface			
DPE	11	66	16.67%	**2.60**	38	66	57.58%	**1.50**
	23 of 359 (6.41%) residues in active site				138 of 359 (38.44%) residues in interface			
NAAAR/ OSBS	5	16	31.25%	**4.99**	13	16	81.25%	**1.88**
	23 of 367 (6.27%) residues in active site				159 of 367 (43.32%) residues in interface			
OSBS	1	54	1.85%	**1.19**	30	54	55.56%	**1.54**
	5 of 322 (1.55%) residues in active site				116 of 322 (36.02%) residues in interface			

2.3.4. Discussion

We find class-specific residues occurring more frequently than expected both near the ligand and also near the interface. Contrary to what we expected from previous studies, we did not find a high concentration of class-specific residues in the 20s loop of the N-terminal capping domain. In the 50s loop of the N-terminal capping domain, we find some class-conserved residues, but not a high concentration. The regions with the highest concentration of class-specific residues occur near the interface between the N-terminal capping domain and the C-terminal barrel domain.

Previous studies have concluded that the 20s and 50s loops are involved in substrate specificity based on their position in "capping" the active site and because the position of these loops can vary from structure to structure (Gerlt et al. 2003; Glasner, Fayazmanesh et al. 2006). However, our observation that there are very few class-specific residues in these loops suggests another explanation for the role of these loops.

This lack of class-specific residues on these loops suggests that these loops may be more important for excluding water from the active site or for non-specific binding of the substrate. Alternatively, it is possible that the 20s and 50s loops may be involved in mediating substrate specificity through multiple sequence signatures that achieve the same specificity, despite the lack of class-conserved residues on these loops. Until mutation studies are completed, we cannot be sure of the exact nature these loops.

Our observation that residues at the interdomain residues are conserved within isofunctional families in the enolase superfamily further confirms the conclusion from work described in Section 2.2, that the domain orientation is important for achieving specificity.

Combining the observations of this study with those from the study described in Section 2.2 suggests that the size and overall shape of the binding pocket is important for achieving different specificities among the different families of the enolase superfamily. The 20s and 50s loops could contribute to the size and shape of the binding pocket in a non-specific manner and functions can be conserved through by maintaining same size of loop. The contribution of the interdomain orientation to the size and shape of the pocket can similarly be maintained through the conservation of residues at the interface, as we observe in this study. However, the enzymes in this superfamily are often fairly specific, turning over only their own substrate and not similarly sized substrates and the loops and interdomain orientation may not completely explain the different specificities within the superfamily. The class-specific residues that we find close to ligand, of which most are on the C-terminal domain, could be responsible for fine-tuning the specificity that is generally shaped by the 20s and 50s loops and the interdomain orientation.

Previous studies (Gerlt et al. 2003) have suggested that the functions performed by members of the superfamily are segregated into different domains, with chemistry mediated by the C-terminal barrel domain and the substrate specificity determined by the N-terminal capping domain. Based on the results from this study, substrate specificity is likely to be mediated by portions of both domains. Further studies to mutate individual residues and combinations of residues are required to determine exactly how the substrate specificity is determined in these enzymes and how this specificity is maintained or diverges through evolution. There is ongoing work to mutate the 20s and 50s loops to further explore their role in enzyme function.

2.4. Conclusion

To more fully understand the sequence, structure, and function relationship in superfamilies like this one will require more detailed analysis of functions. In addition to correlating conserved sequence signatures with conservation in structural elements and in inter-domain orientations, it will also be useful to correlate conserved sequence signatures with conserved aspects of function. The following chapters (Chapter 3, Chapter 4) describe work in this area to systematically describe function and determine which aspects of function are conserved among related proteins.

Chapter 3

Evolutionarily Conserved Substrate Substructures for Automated Annotation of Enzyme Superfamilies

3.1. Abstract

The evolution of enzymes affects how well a species can adapt to new environmental conditions. During enzyme evolution, certain aspects of molecular function are conserved while other aspects can vary. Aspects of function that are more difficult to change or that need to be reused in multiple contexts are often conserved, while those that vary may indicate functions that are more easily changed or that are no longer required. In analogy to the study of conservation patterns in enzyme sequences and structures, we have examined the patterns of conservation and variation in enzyme function by analyzing graph isomorphisms among enzyme substrates of a large number of enzyme superfamilies. This systematic analysis of substrate substructures establishes the conservation patterns that typify individual superfamilies. Specifically, we determined the chemical substructures that are conserved among all known substrates of

a superfamily and the substructures that are reacting in these substrates, and then examined the relationship between the two. Across the 42 superfamilies that were analyzed, substantial variation was found in how much of the conserved substructure is reacting, suggesting that superfamilies may not be easily grouped into discrete and separable categories. Instead, our results suggest that many superfamilies may need to be treated individually for analyses of evolution, function prediction, and to guide enzyme engineering strategies. Annotating superfamilies with these conserved and reacting substructure patterns provides information that is orthogonal to information provided by studies of conservation in superfamily sequences and structures, thereby improving the precision with which we can predict the functions of enzymes of unknown function and direct studies in enzyme engineering. Because the method is automated, it is suitable for large-scale characterization and comparison of fundamental functional capabilities of both characterized and uncharacterized enzyme superfamilies. This chapter is modified from a published report of this project (Chiang et al. 2008).

3.2. Introduction

Why are some aspects of function shared and others allowed to change? By examining which aspects of function are shared among contemporary enzymes, we can gain insight into the requirements and constraints that govern this evolutionary process.

The focus of most studies of enzyme evolution has been the examination of conservation in sequence and structure. The data available to conduct such studies is enormous and still increasing due to the multiplicity of ongoing genomic and metagenomic sequencing efforts (Riesenfeld et al. 2004). In tandem with the growth of sequence and structural data, a large number of new and sophisticated tools have been

developed to improve our ability to identify the divergent members of superfamilies, allowing us to analyze patterns of conservation in sequence and structure that shed light on how enzyme functions have evolved and diversified (for some examples, see (Frazer et al. 2003; Pearson et al. 2005; Marti-Renom et al. 2007)). But such studies only capture aspects of enzyme evolution that can be inferred from the machinery that enables enzymatic catalysis, the enzymes themselves. Far fewer studies have focused on the substrates and products of these reactions, with most of these focused on the requirements of metabolism (Alves et al. 2002; Light et al. 2004). In this work, our goal is to understand the details of how enzymes function and evolve by studying the conservation and variation in their substrates and products. In doing so, we aim for a more extensive view of enzyme evolution in order to improve our abilities to annotate enzymes of unknown function and to infer common aspects of function for superfamilies that have not yet been characterized.

As described in Section 1.5, the success of any study of the evolution of enzyme function depends on how function is defined and described. Previous studies fall into two categories: detailed analyses that are limited in their scope because of the labor-intensive nature of these analyses and automated analyses that have larger scope but lose detail in how they describe function. The goal of this project was to develop methods that can be used for automated analyses of enzyme function, but that also do not sacrifice the level of detail.

Here, we use graph isomorphism analyses to compare substrates of enzymes from 42 superfamilies to identify specific aspects of function conserved within each superfamily. We also use comparisons of substrates and their corresponding products to

determine whether and how much of the conserved substructure is involved in the reaction. This comparison of substrates and products is similar to an analysis performed for a previous study with a different purpose, to predict EC numbers (Kotera et al. 2004). To simplify the interpretation of results across the multiple superfamilies in this study, only enzymes comprised of single domains and that catalyze unimolecular reactions were investigated. Automation of the analysis allows us to describe overall trends in functional conservation and variation across a large number of superfamilies. A descriptive representation of conserved enzyme molecular functions using chemical structures and SMILES strings (Weininger 1988; Weininger et al. 1989) is also provided. This representation should be useful for annotating new members of superfamilies discovered in sequencing projects and for characterizing new superfamilies.

3.3. Methods

3.3.1. Dataset – Enzyme superfamilies

For our analyses, we used a subset of superfamilies from SCOP, a database of manually classified protein superfamilies, filtered based on criteria chosen to be most informative about enzyme evolution at high levels of functional divergence. We included only superfamilies of single-domain enzymes with significant functional information in SCOPEC, a subset of SCOP with verified EC numbers, and in BRENDA, the most comprehensive database of enzyme experimental results. Although many enzymes and proteins function as multi-domain units, the nature and organization of which can affect the specificity and regulation of enzymes (Bashton et al. 2007), for this study, we chose to use only single-domain enzymes as this allowed us to clearly assign a single function

to one domain. We included examples of enzymes known to have multiple structural domains only when the composite acts as a single functional unit (e.g., the enolase superfamily).

To ensure that the members of each superfamily were sufficiently divergent in function to analyze conservation of their substructures, only superfamilies annotated with at least two different EC numbers were investigated. Compared to unimolecular reactions, bimolecular reactions have considerably more complex chemical and kinetic mechanisms for how substrates interact with the enzyme's catalytic site (i.e. in what order different substrates bind). Because these variations would have greatly complicated the analysis, we excluded superfamilies with any reactions that were not unimolecular. Using the top level of the EC annotation, superfamilies were selected in which all the characterized members belong to any one of the following classes: hydrolases (EC numbers 3.x.x.x), lyases (EC numbers 4.x.x.x), and isomerases (EC numbers 5.x.x.x).

Experimentally verified substrate and product data were taken from the licensed version of the BRENDA database (release 6.2) (Barthelmes et al. 2007). Reactions were excluded in which 1) the product(s) had more than five (non-hydrogen) atoms more than the substrate or 2) substrates and products both had three or fewer (non-hydrogen) atoms. Reactions in the first category are likely to be erroneous because they are not properly balanced. Reactions in the second category are unlikely to be informative for the analysis because they contain so few atoms.

3.3.2. Definitions

A "conserved substructure" (Figure 3.1) contains the maximal sets of bonds in a substrate that are present in all the substrates of a superfamily, plus their adjacent atoms.

In all our analyses, we considered only bonds consisting of two atoms, neither of which is a hydrogen. The "unconserved substructure" is the set of bonds in a substrate that are not in the conserved substructure, plus their adjacent atoms. An atom can be in both the conserved and unconserved substructure if it is adjacent to both a bond in the conserved substructure and a bond in the unconserved substructure.

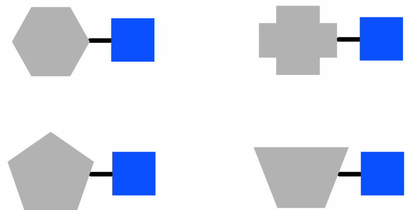


Figure 3.1. The conserved substructure (c) (blue square)

A "reacting substructure" (Figure 3.2) consists of the bonds in a substrate that are not present in the product, their adjacent atoms, and any atoms that become connected in new bonds in the product. In the case of a racemization reaction, in which the chirality of an atom center changes, the reacting substructure is defined as including the chiral atom that changes in the reaction, the four adjacent bonds and their adjacent atoms. The "nonreacting substructure" is the set of bonds in a substrate that are also present in the product and their adjacent atoms. An atom can be in both the reacting and nonreacting substructure if it is adjacent to both a bond in the reacting substructure and a bond in the nonreacting substructure.

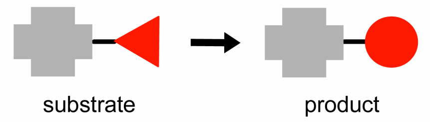


Figure 3.2. Reacting substructure (r) (red triangle)

3.3.3. Finding the conserved substrate substructure

The substrate substructure conserved among all characterized members of each superfamily was calculated using the maximal common substructure (MCS) algorithm implemented in the Chemistry Development Kit (CDK) (Steinbeck et al. 2003), an open source Java toolkit for manipulating small molecules. The molecules are represented as graphs in which the nodes represent atoms and the edges represent bonds. Each node is labeled with an atom type and each edge is labeled with the two atom types of the connected atoms and the bond order. This algorithm finds, for a pair of molecules, the maximum common substructure (MCS) present in both molecules. We extended this to find the MCS for the set of all known substrates for a superfamily. In this initial analysis, we treated different atoms as dissimilar as long as the element type was different and bonds as different when the bond order and the two pairs of connected atoms were not identical. The only exception to this rule was made for phosphate and sulfate groups, which we treated as similar in the substrate conservation analyses. Our code allowed for the possibility of multiple unconnected MCSs by representing them as an unconnected graph with each connected portion corresponding to one MCS. Although some of the pairwise MCSs contain multiple unconnected subgraphs, none of the superfamily-conserved substructures contain such multiple unconnected MCSs. Finally, each substrate has a unique unconserved substructure defined as the set of edges not present in the conserved substructure and the atoms adjacent to these edges.

3.3.4. Finding the reacting substrate substructure

For each enzymatic reaction in which both the substrate and its corresponding product(s) are known, we calculated the non-reacting substructure by finding the MCS

between the substrate and the product(s). The reacting substructure is the set of edges in the substrate that are not present in the product, plus the atoms adjacent to these edges. The reacting substructure also includes atoms that form new bonds in the product.

3.3.5. Overlap between reacting and conserved substructures

To quantify the overlap between the reacting and conserved substructures, for each reaction in our dataset, we calculate f_c (Figure 3.3A), the fraction of the conserved substructure that is reacting and f_r (Figure 3.3B), the fraction of the reacting substructure that is conserved.

Figure 3.3. Measures of overlap between reacting and conserved substructures

A) fc is the fraction of the conserved substructure (blue square) that is reacting (red triangle overlap) B) fr is the fraction of the reacting substructure (red triangle) that is conserved (blue square overlap)

The values for f_c and f_r are calculated in two ways, using atoms or bonds and the results for both are reported as they provide different but useful views of the data. f_c for bonds is determined by dividing the number of bonds that are in both the conserved and the reacting substructures ($r \cap c$) by the number of bonds in only the conserved substructure. f_c for atoms is determined similarly, using the number of atoms instead of bonds. Likewise, f_r for bonds is determined by dividing the number of bonds that are in both the conserved and the reacting substructures by the number of bonds in only the reacting substructure; this value was also calculated using atoms. For each enzyme in the BRENDA database, there may be multiple substrates with corresponding reactions that

have been characterized. For these cases, the values of f_c and f_r were obtained by averaging all the substrates of each enzyme and then these values were averaged for all the enzymes in each superfamily. We also determined the standard deviation in f_c and f_r for the enzymes of each superfamily.

3.3.6. Variation in which substructure is reacting

To determine whether the same part of the superfamily-conserved substructure was used in the different reactions of the superfamily, every pair of reactions was analyzed in each of the superfamilies in our dataset. Each reaction has a substrate substructure that is both conserved and reacting ($r \cap c$). For each pair of reactions, we calculated how much overlap is observed among the two ($r \cap c$) substructures and normalized each of these overlaps by the smallest ($r \cap c$) of each pair. The resulting measure of overlap ($o_{r \cap c}$) was then averaged over every pair of reactions in each superfamily.

3.4. Results

The 42 superfamilies that meet our criteria and for which there is sufficient data in Brenda include representatives of six of the seven SCOP fold classes; the only fold class not represented is the membrane proteins class. The enzymes of these 42 superfamilies represent a substantial proportion of the diversity of enzyme function, covering 25.4% of EC classes defined by the first two digits (subclasses) and 18.7% of EC classes defined by the first three digits (sub-subclasses). Conservation patterns were examined using only substrates and products as the data available in BRENDA were not sufficient to consider other aspects of reactions, such as transition states and intermediates.

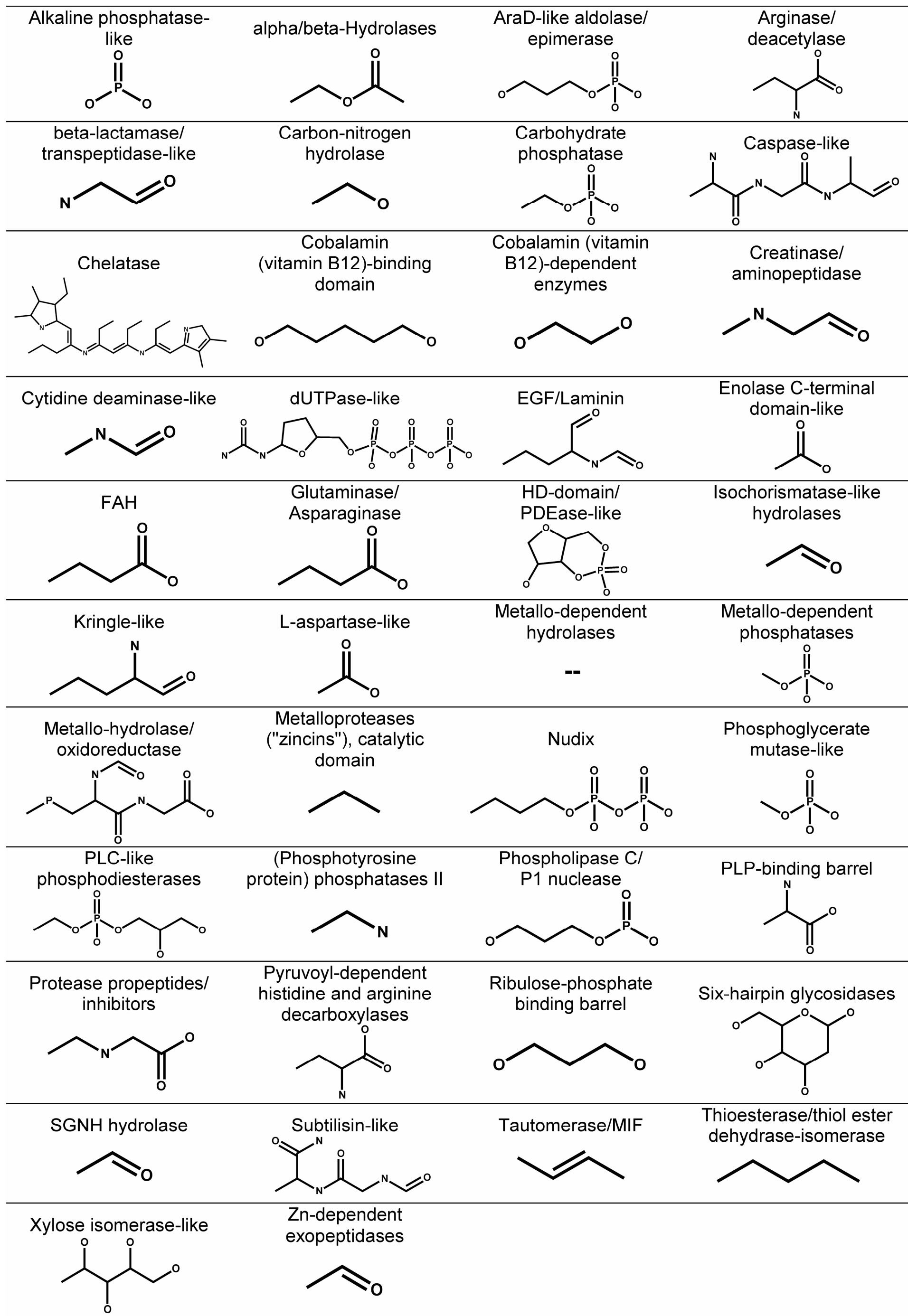


Figure 3.4. Summary of superfamilies and their conserved substrate substructures

Because the portion of the conserved substructure that is reacting often varies among members within one superfamily, we do not highlight the reacting substructure in this figure. (See Figure 3.7 for plots of the distribution of this variation over all superfamilies and Table 3.1 for values of variation for each superfamily.)

Our goal was to determine the molecular features that the substrates of a superfamily share and whether the shared features are involved in the reactions catalyzed by that superfamily. These conserved substructures for the 42 superfamilies in our dataset are shown in Figure 3.4.

Additional information about the diversity and conservation of functions in these superfamilies is provided in a hyperlinked table (Table S1 in Chiang et al. 2008). Moreover, for each enzyme's substrate(s), we found the reacting substructure and then determined whether the conserved substructure overlaps with the reacting substructure and by how much. Results for these measures of overlap are presented with respect to both the number of atoms and the number of bonds.

For a given superfamily, the average f_c and f_r calculated using atoms often differ from the values obtained using bonds (Table 3.1). This difference arises because the number of bonds is frequently not proportional to the number of atoms in molecular structures (e.g., one bond consists of two atoms while three atoms can be connected by three bonds; a cyclic structure will have a different number of bonds compared to non-cyclic structure with the same number of atoms). In addition, different types of reactions vary in the ratio of atoms and bonds that are involved in the reaction (e.g., a lyase may break one bond involving two atoms while an intramolecular transferase may involve one bond and three atoms). Because both are valid measures of substructure size, both are provided in this report.

Table 3.1. Overlap between reacting and conserved substructures (f_c and f_r)

Overlap between reacting and conserved substructures (f_c and f_r). The superfamilies in this table are sorted by [average f_c(atoms) plus f_c(bonds)]. *The metallo-dependent hydrolases superfamily does not have a substrate substructure that is conserved in all members of the superfamily. Thus, for this superfamily, f_c, the fraction of the conserved substructure that is reacting, cannot be calculated.

Superfamily	SCOP ID	f_c Avg. Atoms	f_c Avg. Bonds	f_c Std. Dev. Atoms	f_c Std. Dev. Bonds	f_r Avg. Atoms	f_r Avg. Bonds	f_r Std. Dev. Atoms	f_r Std. Dev. Bonds
Alkaline phosphatase-like	c.76.1	0.98	0.98	0.04	0.05	0.66	0.64	0.06	0.05
SGNH hydrolase	c.23.10	0.95	0.92	0.02	0.03	0.65	0.55	0.04	0.01
Nitrile hydratase alpha chain	d.159.1	0.72	0.68	0.16	0.17	0.82	0.81	0.26	0.27
Carbohydrate phosphatase	e.7.1	0.72	0.67	0.01	0	0.98	0.97	0.03	0.03
Cobalamin (vitamin B12)-dependent enzymes	c.1.19	0.81	0.54	0.09	0.06	0.81	0.81	0.09	0.09
Phosphoglycerate mutase-like	c.60.1	0.61	0.55	0.29	0.32	0.68	0.64	0.31	0.38
Six-hairpin glycosidases	a.102.1	0.56	0.48	0.14	0.15	0.65	0.67	0.13	0.18
alpha/beta-Hydrolases	c.23.9	0.55	0.48	0.23	0.21	0.64	0.63	0.45	0.47
PLP-binding barrel	c.1.6	0.56	0.47	0.1	0.12	1	1	0	0
Carbon-nitrogen hydrolase	d.160.1	0.5	0.5	0.71	0.71	0.07	0.04	0.09	0.06
Creatinase/aminopeptidase	d.127.1	0.55	0.44	0.24	0.3	0.56	0.45	0.23	0.31
Metalloproteases ("zincins"), catalytic domain	d.92.1	0.55	0.43	0.31	0.38	0.28	0.13	0.17	0.14
Nudix	d.113.1	0.5	0.46	0.2	0.22	0.46	0.40	0.2	0.18
Phospholipase C/P1 nuclease	a.124.1	0.5	0.43	0	0	0.52	0.48	0.21	0.22
Pyruvoyl-dependent histidine and arginine decarboxylases	d.155.1	0.47	0.38	0.06	0.07	0.93	0.95	0.09	0.07
PLC-like phosphodiesterases	c.1.18	0.43	0.36	0.15	0.16	0.45	0.39	0.1	0.07
dUTPase-like	b.85.4	0.41	0.35	0.03	0	0.92	0.9	0.12	0.14
Tautomerase/MIF	d.80.1	0.58	0.17	0.14	0.17	0.32	0.13	0.16	0.13
Xylose isomerase-like	c.1.15	0.43	0.23	0.26	0.24	0.71	0.49	0.19	0.24
Zn-dependent exopeptidases	c.56.5	0.42	0.19	0.18	0.13	0.37	0.07	0.16	0.04
Chelatase	c.92.1	0.33	0.23	0.09	0.08	0.68	0.47	0.02	0.11
L-aspartase-like	a.127.1	0.38	0.17	0.31	0.41	0.41	0.1	0.13	0.24
Protease propeptides/inhibitors	d.58.3	0.30	0.14	0.05	0.13	0.62	0.47	0.21	0.42
Ribulose-phosphate binding barrel	c.1.2	0.28	0.13	0.28	0.13	0.37	0.25	0.36	0.25
Metallo-hydrolase/oxidoreductase	d.157.1	0.25	0.15	0.15	0.11	0.92	0.88	0.12	0.18
Enolase C-terminal domain-like	c.1.11	0.31	0.08	0.1	0.13	0.35	0.07	0.07	0.12
Thioesterase/thiol ester dehydrase-isomerase	d.38.1	0.19	0.19	0.34	0.33	0.12	0.11	0.21	0.18
Cobalamin (vitamin B12)-binding domain	c.23.6	0.21	0.17	0.30	0.26	0.38	0.33	0.53	0.47
Subtilisin-like	c.41.1	0.23	0.14	0.09	0.06	0.57	0.52	0.36	0.42
Kringle-like	g.14.1	0.22	0.13	0.24	0.25	0.3	0.11	0.19	0.21
beta-lactamase/ transpeptidase-like	e.3.1	0.29	0.06	0.26	0.1	0.28	0.03	0.26	0.05
(Phosphotyrosine protein) phosphatases II	c.45.1	0.2	0.13	0.19	0.18	0.07	0.04	0.06	0.05
FAH	d.177.1	0.22	0.07	0.1	0.12	0.34	0.04	0.15	0.07

HD-domain/PDEase-like	a.211.1	0.17	0.08	0	0	0.79	0.81	0.02	0.04
Cytidine deaminase-like	c.97.1	0.20	0	0.05	0	0.26	0	0.07	0
Isochorismatase-like hydrolases	c.33.1	0.17	0	0.24	0	0.25	0	0.35	0
Glutaminase/Asparaginase	c.88.1	0.13	0.03	0.06	0.04	0.35	0.11	0.14	0.15
Caspase-like	c.17.1	0.11	0.03	0.08	0.04	0.33	0.11	0.11	0.16
AraD-like aldolase/epimerase	c.74.1	0.12	0.01	0.02	0.01	0.38	0.01	0.16	0.02
EGF/Laminin	g.3.11	0.04	0	0.04	0	0.06	0	0.06	0
Arginase/deacetylase	c.42.1	0.04	0	0.05	0	0.06	0	0.09	0
*Metallo-dependent hydrolases	c.1.9	--	--	--	--	0	0	0	0

The distribution of average f_c for the set of superfamilies (Figure 3.5) indicates that there is a continuum among the superfamilies in how much of the conserved substructure is reacting, with superfamilies ranging from having little to having most of the conserved substructure participating in the reaction. This trend is observed regardless of whether we use atoms or bonds in our calculations of average f_c. The results also show that all superfamilies with a conserved substructure have an average f_c above zero, indicating that at least part of the conserved substructure is involved in the reaction.

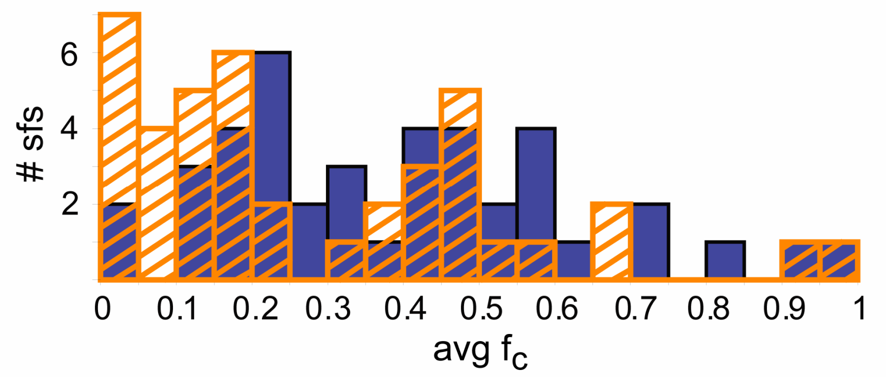


Figure 3.5. Distribution of average fraction of conserved substructure that is reacting.

For bonds (orange stripe) and for atoms (blue solid)

Only one superfamily in our study set, the superfamily defined by SCOP as the metallo-dependent hydrolase superfamily, also known as the amidohydrolase superfamily (Holm et al. 1997; Gerlt et al. 2003), has substrates so diverse that they do not share a common substructure of even a single conserved bond. Detailed analysis of the superfamily, including analysis of differences in the overall functions, how active site motifs are used for catalysis, and other factors such as metal ion dependence, suggests that this group may be more properly considered as multiple superfamilies (Brown and Babbitt, in preparation).

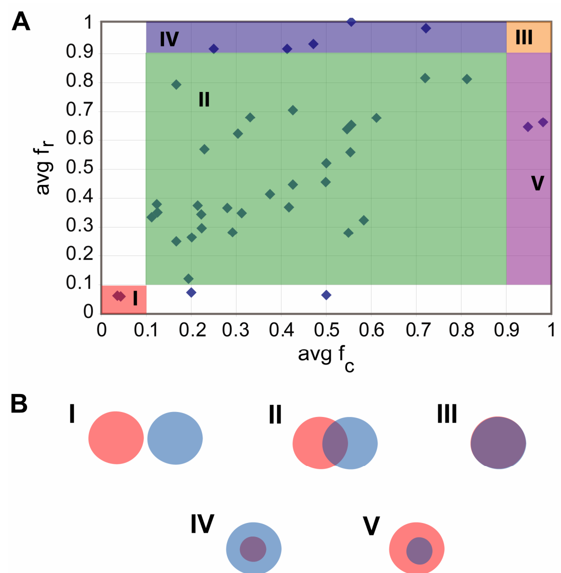


Figure 3.6. Patterns of overlap between reacting and conserved substructures

A) Scatter plot of average fr versus fc. Each superfamily is represented by a blue diamond. The plot is colored to orient the reader within the plot and to roughly indicate where the different overlap patterns fall. The regions labeled with Roman numerals correspond to the overlap patterns in part B of the figure. B) Five types of overlap patterns. (I) Completely nonoverlapping (red); (II) partially overlapping (green), (III) completely overlapping (orange), (IV) reacting is part of conserved substructure (blue), (V) conserved is part of reacting substructure (purple).

Plotting f_r, the fraction of the reacting substructure that is conserved, against f_c illustrates the distribution of superfamilies (Figure 3.6A) across different patterns of overlap (Figure 3.6B) in the reacting and conserved substructures. For simplicity, only the data calculated using atoms is provided in Figure 3.6A. The values for each superfamily, calculated using both atoms and bonds, are provided in Table 3.1. The different regions in Figure 3.6A are intended merely to orient the reader to the range of variation across multiple superfamilies rather than to infer distinct categories implying fundamental differences between the superfamilies in different regions.

Most superfamilies have little variation in how much of the conserved substructure is reacting (variation of f_c) (Table 3.1, Figure 3.7A). However, there are a few superfamilies with substantial variation in f_c. We also evaluated the level of variation in which part of a superfamily's conserved substructure is used among the different reactions ($o_{r \cap c}$). A flatter distribution and more variation was observed among the superfamilies for the average $o_{r \cap c}$ (Figure 3.7B) than for the standard deviation of f_c. The superfamilies that rank highest both in variation in f_c and $o_{r \cap c}$ include the carbon-nitrogen hydrolase, metalloproteases ("zincins") (catalytic domain), and the thioesterase/thiol ester dehydrase-isomerase superfamilies. Superfamilies that have low variation in f_c and $o_{r \cap c}$ include the HD-domain/PDEase-like, dUTPase-like, and carbohydrate phosphatase superfamilies.

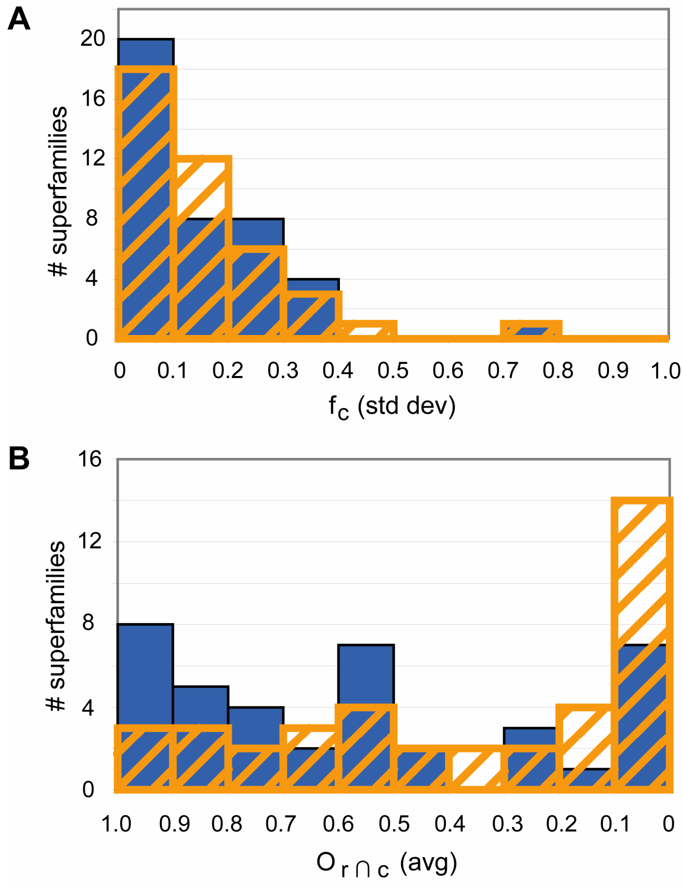


Figure 3.7. A) Variation in the fraction of the conserved substructure that is reacting. B) Variation in which part of conserved substructure is reacting.

A) Distribution of the observed standard deviation in fc within each superfamily, for bonds (orange stripe) and atoms (blue solid). B) Average pairwise overlap in the reacting and conserved substructure ($o_{r \cap c}$), for bonds (orange stripe) and atoms (blue solid). In both plots, superfamilies with less variation can be found on the left side of the distributions and those with more variation are found on the right.

From these examples of superfamilies with high and low variation in f_c and $o_{r \cap c}$, we observe that the superfamilies with high variation tend to have smaller conserved substructures while superfamilies with low variation tend to have larger conserved substructures, though the correlation is not perfect. The superfamilies in the low variation group have phosphate groups in the conserved substructure. These tendencies

may arise because different superfamilies and different types of reactions have different propensities for variation and conservation through evolution. Alternatively, variation in how different superfamilies are defined in SCOP may lead to some of the variation observed among these superfamilies. We also note that the set of reactions surveyed in this work represents only a subset of enzyme superfamilies, making it difficult to definitively address these hypotheses and questions. More extensive analyses will be required to confirm and further explore these initial observations.

As new superfamily members are characterized, modifications of these substructure conservation patterns may be required. To provide updates of this information, work is underway to incorporate this information into a searchable resource within our Structure-Function Linkage Database (http://sfld.rbvi.ucsf.edu/) (Pegg et al. 2006). Additional data generated in this study, including reacting substructures and how they overlap with conserved substructures for individual superfamily members, are available from the authors upon request. As described below (Section 3.5.2) , our method can also be used to determine conserved functional characteristics for superfamilies that have not yet been characterized. Programs and scripts required to perform these analyses are also available upon request.

3.5. Discussion

Our analysis of the conservation of substrate substructures in enzyme superfamilies precisely determines aspects of chemical transformations that are conserved during divergent evolution. As such, it provides a view of conservation and divergence different from the view afforded by more common types of studies focused on enzyme sequences and structures.

3.5.1. Patterns of substrate conservation across many superfamilies

While our dataset of superfamilies and their associated substrates, products, and reactions is large, it is still limited as only single domain and unimolecular enzymes and superfamilies with sufficient data available were considered. Nevertheless, the results suggest a continuum in how enzyme superfamilies have evolved, from the reacting substructure being mostly conserved to being only slightly conserved (Figure 3.5). Moreover, these superfamilies span a wide range in patterns of overlap (Figure 3.6).

Previously, both large-scale and focused studies of enzyme evolution have recognized two primary models of how function is conserved (Section 1.4). For the most part, the previous studies that have classified superfamilies into one or the other of these categories have been limited either in their scope (see the review by Glasner et al. for examples (Glasner, Gerlt et al. 2006)) or in the type of data used (Todd et al. 1999; Alves et al. 2002; Schmidt, Sunyaev et al. 2003; Light et al. 2004). Although our current work cannot be directly compared with these previous analyses because of differences in methodologies, our results suggest that the evolution of enzyme function is too complex to be described by a few distinct categories. Instead, we see large variations in the patterns of substrate conservation across the set of superfamilies investigated in this study. Also, in these superfamilies, conserved substructures are not entirely reacting nor are they entirely non-reacting. This observation also suggests that the reacting and non-reacting substructures, the latter often including the part of the substrate that has binding interactions with the enzyme, are simultaneously relevant to the evolutionary process and should be analyzed together. Consistent with our observations, a recent network-based analysis of the evolution of metabolism concludes that the two models previously used to

describe enzyme evolution are not mutually exclusive or independent (Diaz-Mejia et al. 2007).

Variations observed within individual superfamilies suggest additional complexity in the evolution of function and how conserved substrate substructures are used in catalysis. Although within most of the superfamilies we studied there is little variation in the extent to which conserved substructures are involved in the reaction (Figure 3.7), the observation of some variation, and in a few cases, considerable variation, demonstrates that even members of the same superfamily may not proceed with the same pattern of evolution.

As discussed in the sections below, these results also suggest potentially important implications for the analysis of individual superfamilies, functional annotation, and value of evolutionary information in providing guidance for enzyme engineering.

3.5.2. Functional annotation of superfamilies and enzymes

By automating the analysis of enzyme substrates and reactions, the methodology introduced in this work facilitates the analysis of previously unstudied enzyme superfamilies. This effort contrasts with previous analyses of enzyme superfamilies to determine patterns of functional conservation that have been highly labor-intensive, involving extensive manual analysis of reactions and literature-based curation of functional properties (see the SFLD, http://sfld.rbvi.ucsf.edu/, for examples). The substructures conserved among the substrates of all members of a superfamily (Figure 3.4) provide annotation information that describes how function has been conserved in each of these superfamilies. The certainty of these superfamily annotations will depend, however, on how well the range of substrates in each superfamily has been sampled.

Thorough substrate sampling may be especially critical for complex superfamilies that include many different catalytic functions. While we have used all available reaction information in our analyses, the sampling of superfamily reactions may still be incomplete. As new reactions are discovered through the sequencing of new genomes and metagenomes, these results can be updated and improved.

Despite these limitations, the characterization of superfamily-conserved substructures presented here facilitates the annotation of individual sequences on a large scale, helping to address the need for new strategies for automated function annotation. This issue has become more pressing as the number of sequenced genomes increases and the era of metagenomics moves into high gear (Friedberg 2006). Sequences that can be classified into a superfamily but not into a specific family can be annotated with the substructure common to all characterized members. In these cases, often found in complex superfamilies exhibiting broad diversity in enzyme function, this may be the only level at which accurate annotation can be achieved, as insufficient information may be available to support annotation of a specific reaction or substrate specificity.

While substructure-based annotation does not by itself suggest a specific enzyme function, this information can be used as a starting point for additional analyses to determine specific function. For example, many structures have been solved through structural genomics efforts, but their functions remain unknown (Gerlt 2007). We have compiled a list of structures that have been classified into the SCOP superfamilies analyzed in this study, but have unknown functions. These structures, many of them from structural genomics projects, can be at least minimally annotated with the substructure identified here as conserved across that superfamily, illustrated by the

examples given in Figure 3.8 (see Table S3 in (Chiang et al. 2008) for the complete list). Using this information, characteristics of ligands likely to be bound or turned over by these proteins can be inferred, providing guidance for biochemical studies to determine specificity. These data also provide information about classes of small molecules that may be useful for co-crystallization trials to aid in solving the structures of these proteins or to capture them in functionally relevant conformations.

The variation found within superfamilies presents a caveat to be considered when using these substructures for function annotation. While most of the superfamilies analyzed here have conserved substructures that are used consistently among the different superfamily members (Figure 3.7), there are a few superfamilies that have significant variation in the degree to which the conserved substructure is used in the reactions. These superfamilies can be expected to be more difficult cases for function prediction since their variability makes it more difficult to determine conserved aspects of function. In contrast, superfamilies with less variation in the degree to which the conserved substructure is used in the reaction are expected to be more straightforward cases for function prediction.

Superfamily and conserved substructure	SCOP ID	PDB ID	Current Annotation
alpha/beta-Hydrolases	c.69.1	1vkh	Putative serine hydrolase Ydr428c
		1pv1	Hypothetical esterase YJL068C
		1r3d	Hypothetical protein VC1974
		1vk9	Hypothetical protein TM1506
Enolase C-terminal domain-like	c.1.11	1rvk	Hypothetical protein Atu3453
		1zz, 2dw6, 2dw7	Hypothetical protein Bll6730
		2gdq, 2gge,	Hypothetical protein YitF
		2gl5	Putative dehydratase protein STM2273
HD-domain/PDEase-like	a.211.1	1ynb, 1yoy	Hypothetical protein AF1432
		2hek	Hypothetical protein aq_1910
		2o6i	Hypothetical protein EF1143
		1vqr	Hypothetical protein Cj0248
Metallo-dependent phosphatases	d.159.1	1nmw	Hypothetical protein PF1291
		1uf3	Hypothetical protein TT1561
		1s3m, 1s3l, 1s3n, 2ahd	Putative phosphodiesterase MJ0936
		1xm7	Hypothetical protein aq_1666
		1t70	Putative phosphatase DR1281
		1t71	Hypothetical protein MPN349
		2cv9	Hypothetical protein TTHA0625
Metallo-hydrolase/oxidoreductase	d.157.1	1vjn	Hypothetical protein TM0207
		1zkp	Hypothetical protein BA1088 (BAS1016)
		2az4	Hypothetical protein EF2904
		1ztc	Hypothetical protein TM0894
Nudix	d.113.1	1sjy, 1sz3, 1su2, 1soi	Hypothetical protein DR1025
		1k2e, 1k26, 1jrk	Hypothetical protein PAE3301
		2azw	Hypothetical protein EF1141
		2b0v	Hypothetical protein NE0184
		2b06	Hypothetical protein SP1235 (spr1115)
		1q27	Hypothetical protein DR0079
		2fkb	Hypothetical protein YfcD
		2fml	Hypothetical protein EF2700, N-term. domain
		2fbl	Hypothetical protein BT0354, N-term. domain
Xylose isomerase-like	c.1.15	1i60, 1i6n	Hypothetical protein IolI
		2g0w	Hypothetical protein Lmo2234
		1k77	Hypothetical protein YgbM (EC1530)
		1yx1	Hypothetical protein PA2260

Figure 3.8. Protein structures with unknown function can be annotated with superfamily-conserved substructures.

This partial list includes superfamilies with between four and nine proteins of unknown function. See Table S3 in (Chiang et al. 2008) for the full list.

3.5.3. Guidance for protein engineering

Understanding the patterns of functional conservation associated with the evolution of functionally diverse enzyme superfamilies can provide useful information for guiding enzyme engineering experiments in the laboratory (Glasner et al. 2007). Using as a starting template for design or engineering an enzyme that already "knows" how to perform a critical partial reaction or how to bind a required substrate substructure ensures that some of the machinery required to perform a desired function is already in place. Although still daunting, the task then simplifies to modifying the enzyme to bind and turn over a new substrate that contains the substructure consistent with the underlying capabilities of the superfamily. As a corollary, aspects of function that have been conserved in all members of a divergent superfamily may be difficult to modify by in vitro engineering (O'Loughlin et al. 2006; Glasner et al. 2007). Using such a strategy in a proof-of-concept study, two members of the enolase superfamily were successfully engineered to perform the reaction of a third superfamily member (Schmidt et al. 2001). As shown in Figure 3.9, the superfamily-conserved substructure and the partial reaction associated with that substructure were not changed in these experiments. Rather, engineering the template proteins to perform the target reaction involved changing each to accommodate binding the part of the substrate that is unique to the new reaction desired.

To allow for generalization of this approach, our analysis provides for all of the superfamilies that we investigated 1) the parts of an enzyme's substrate and reaction that are not conserved among related enzymes, which, provided they can be associated with regions of a target structure that interact with them, may point to structural features

amenable to engineering, and 2) the parts of the substrates that are conserved across all members of a superfamily, which may point to regions of the structure that may not be easily changed without loss of function or stability (Nagatani et al. 2007).

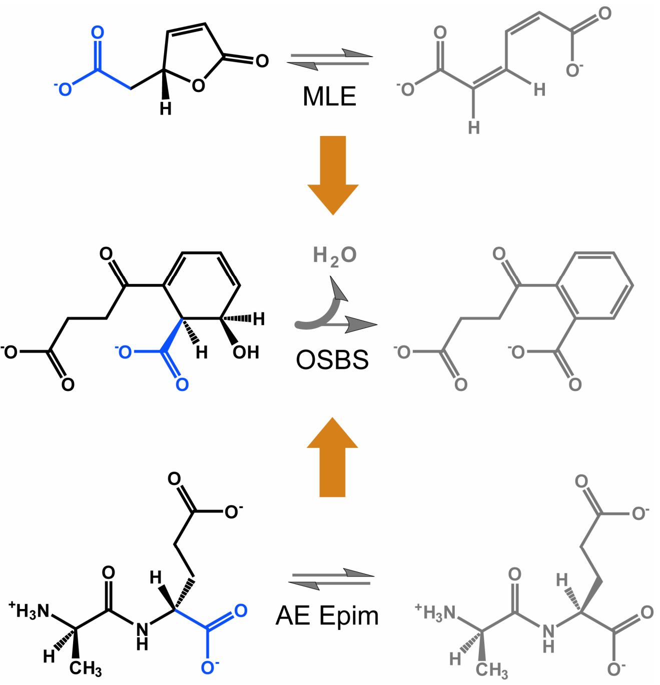


Figure 3.9. Enzyme engineering strategy.

Two previously demonstrated examples using superfamily analysis to guide engineering of enzymes to perform new functions (Schmidt, Mundorff et al. 2003). In the top example, error-prone PCR resulted in a single point mutation of muconate lactonizing II (MLE) enzyme, which enabled it to catalyze the o-succinylbenzoate synthase (OSBS) reaction (k_{cat}/K_M (M^{-1} sec^{-1}) = 2 x 10^3). In the lower example, a single mutation was rationally designed based on comparison of the active sites of Ala-Glu epimerase (AEE) and o-succinyl benzoate synthase (OSBS). The mutant that was generated enabled this enzyme to catalyze the OSBS reaction as well (k_{cat}/K_M (M^{-1} sec^{-1}) = 12.5). In both of these examples, the superfamily conserved substrate substructure (blue) and associated partial reaction were not changed during the engineering experiment. The changes in the reaction that were made are in the portion of the substrates that are not conserved in the superfamily (black). The diverse products of the native MLE, OSBS, and AEE reactions are also shown (grey).

3.5.4. Future directions for substructure analysis

In this study, requirements for a sufficiently large sample of enzyme reactions for a comprehensive analysis restricted us to using only substrates and products. However, enzyme substrates can undergo intermediate changes during catalysis that are not adequately captured by looking only at substrates and products. In some reactions, such as those in the enolase superfamily (Gerlt et al. 2005), some portions of the substrate change and revert back to their original configuration during the reaction; these types of transformations are undetectable in the study described here. The enolase superfamily represents a well-characterized example of chemistry-conserved evolution. However, because our analysis does not currently detect such substrate changes, the average f_c(atoms) for the enolase superfamily is 0.31 and the average f_c(bonds) for the enolase superfamily is 0.34, which places this superfamily in the middle of the distribution among our superfamilies for these measures of overlap. Being able to detect the full extent to which structures change during a reaction would provide a better picture of substructure conservation in superfamilies like the enolase superfamily. But this will require compilation of additional data required to capture all of the partial reactions involved in a given overall reaction, including structures of reaction intermediates. Emerging data resources, such as MACiE (Holliday et al. 2007) and the SFLD (Pegg et al. 2006), currently seek to catalog information about reaction steps and mechanisms. However, because this process is labor-intensive and often hampered by disagreement or ambiguity in the literature regarding the specific mechanisms of some reactions, these data resources are not yet sufficiently populated to support such broader analyses. As these types of resources grow, we are optimistic that the information required to analyze

reaction mechanisms more fully will become increasingly available. Although it is beyond the scope of this study, correlating the conservation patterns we see in enzyme substrates with the conservation patterns in the sequence and structures of the enzymes themselves would also be a valuable extension for these analyses.

Finally, recent progress has been made in using in silico docking of small molecules to enzyme structures to infer molecular function. In one such study, a library of high-energy reaction intermediates was generated and used to predict substrate specificity of enzymes in the amidohydrolase superfamily (Hermann et al. 2006). As these methodologies are further developed, incorporation of predicted reaction intermediates into substructure analysis could improve prediction of substructures that are reacting. In addition to benefiting from such recent advances in docking, the type of analysis presented here may in turn be used to improve applications of docking to predicting substrate specificity in enzymes. Several such studies have recently focused on predicting functional specificity in the enolase (Kalyanaraman et al. 2005; Song et al. 2007) and amidohydrolase (Hermann et al. 2007) superfamilies using knowledge about conserved substrate substructures from earlier analyses (Seibert et al. 2005; Pegg et al. 2006) to construct focused ligand libraries for docking. We expect that the set of conserved substructures generated by our analysis can be used similarly to guide the construction of chemical libraries of ligands to improve prediction of substrate specificity in other superfamilies.

3.5.5. Conclusions

This study presents an automated method for analysis of superfamilies to determine the conserved aspects of their functions, represented by patterns of substrate

conservation. Our results show that superfamilies do not fall into discrete and easily separable categories describing how their functions may have evolved. Rather, the conserved substructures determined in this analysis define superfamily-specific conservation patterns. These results enable precise prediction of functional characteristics at the superfamily level for complex superfamilies whose members perform many different but related reactions, even when the evidence is insufficient to support more specific annotations of overall reaction and substrate specificity. For applications in enzyme engineering, we expect that the identification of the aspects of function that have been most and least conserved during natural evolution will provide guidance for identifying the structural elements of a target scaffold that are most and least amenable to modification, thereby informing engineering strategies for improved success. The following chapter (Chapter 4) describes work to make the results of this work available for researchers interested in these applications.

Chapter 4

Substructures for Enzyme Evolution and Engineering Resource

4.1. Introduction

The goal of the study described in the previous chapter (Chapter 3) was to study the evolution of enzyme function in enzyme superfamilies. We used an automated graph isomorphism analysis to determine what substructures are conserved among the substrates of a superfamily (Chiang et al. 2008). Using the results of this analysis, we were able to study the evolution of function in multiple superfamilies and determined that enzyme evolution suggests more complex patterns of functional divergence than those that have been proposed by previous theories of enzyme evolution (Horowitz 1965; Babbitt et al. 1997) or that have been considered in previous studies (Todd et al. 1999; Schmidt, Sunyaev et al. 2003). The results of the substructure analyses can also be used to improve predictions of function and to guide enzyme engineering. The following sections (Sections 4.1.1, 4.1.2, and 4.1.3) describe how the conserved substructure information can be used to develop hypotheses and guide research in these areas.

4.1.1. Enzyme evolution and superfamilies

The conserved substructure is a representation of the conserved function of a particular superfamily. Based on the results of our previous study (Chiang et al. 2008), the conserved substructure is likely to be directly involved in or at least adjacent to the portion of the substrate undergoing the chemical conversion. Because the conserved substructure is present in all characterized members of a given superfamily, provided the diversity of enzyme substrates has been sufficiently characterized, new enzymes with unknown function can be expected to catalyze reactions on substrates that also contain the conserved substructure. Therefore, the conserved substructure provides initial annotation for the substrate of individual members of the superfamily, especially those members that cannot be classified into a specific family. The initial annotation can be confirmed and further explored through additional experiments.

4.1.2. Computational molecular docking to predict substrate specificity

Recently, researchers have been starting to apply computational molecular docking (Kitchen et al. 2004) methods to predict enzyme substrates (Kalyanaraman et al. 2005; Hermann et al. 2006; Hermann et al. 2007). In several of these successful cases, the ligand library was filtered for molecules that contained the substructures that were known to be conserved among substrates of the particular enzyme superfamily. Thus, during the docking analyses, fewer irrelevant molecules need to be screened and computational resources can be devoted to screening molecules that are more likely to be substrates. In these cases, the conserved substrate was determined through prior and extensive studies (Babbitt et al. 1996; Seibert et al. 2005). The results of the automated

substructure analysis allow researchers to customize their ligand libraries for enzymes in many additional superfamilies.

4.1.3. Enzyme engineering

Enzyme engineering can be used to develop new enzymes to catalyze reactions useful for biodefense, bioremediation, biofuels, and to facilitate the production of molecules important for human health and agriculture. What was conserved in substrates and products during evolution can be used to suggest which parts of the reaction are unlikely to be changed during engineering. Inversely, what has been variable during evolution can suggest the parts of the reaction that can be changed during engineering. The results of the conserved substructure analyses allow us use this strategy and to select enzymes that already partially perform the target reaction and that are likely to be changed to perform the target reaction. There are several criteria that can be used to select a superfamily or enzyme as a good starting point for engineering. For superfamilies, it may be advantageous to select those in which the superfamily conserved substructure is present in the desired new reaction and in which the range of reactions performed is diverse enough to suggest that the desired new reaction is evolvable. Just as it may be advantageous to select superfamilies that are more diverse, it may also be advantageous to select enzymes that are more promiscuous, as they are likely to more evolvable (Khersonsky et al. 2006).

4.1.4. Data resource

To facilitate these goals, we developed a database, the Substructures for Enzyme Evolution and Engineering Resource (SEEER), for researchers to access, interact with, and develop hypotheses from the data. This resource is currently available at a temporary

location, http://sfldtest.rbvi.ucsf.edu/seeer. Although the resource is not currently publicly available, it will eventually be made public.

4.2. Methods and Results

4.2.1. Data

The foundation of the SEEER is the data from the substrate substructure analysis described in the previous chapter (Chapter 3). The full details of the data and methods are described in the methods section of that chapter (Section 3.3). In summary, we used a subset of superfamilies from the Structural Classification of Proteins (SCOP) (Murzin et al. 1995) where the enzymes are all single-domain proteins and the reactions catalyzed by the enzymes of the superfamily are all unimolecular. For these 42 superfamilies, we found the conserved substrate substructures by extending a maximum common substructure algorithm, implemented in the Chemistry Development Kit (CDK) (Steinbeck et al. 2003), to find common substructures among more than two molecules at a time. To find the reacting substructures, we used the maximum common substructure algorithm; in this case, we found the substructures that differ between the substrate and product(s) of each reaction in the superfamilies. We developed a number of measures to describe the overlap between the conserved substructure and the reacting substructure: f_c, the fraction of the conserved substructure that is reacting, and f_r, the fraction of the reacting substructure that is conserved.

4.2.2. MySQL database structure

The underlying data is stored as a MySQL database. The main tables in the structure of database (Figure 4.1) describe superfamily, enzyme family, and reaction information. The number of entries in each of these tables are listed in Table 4.1.

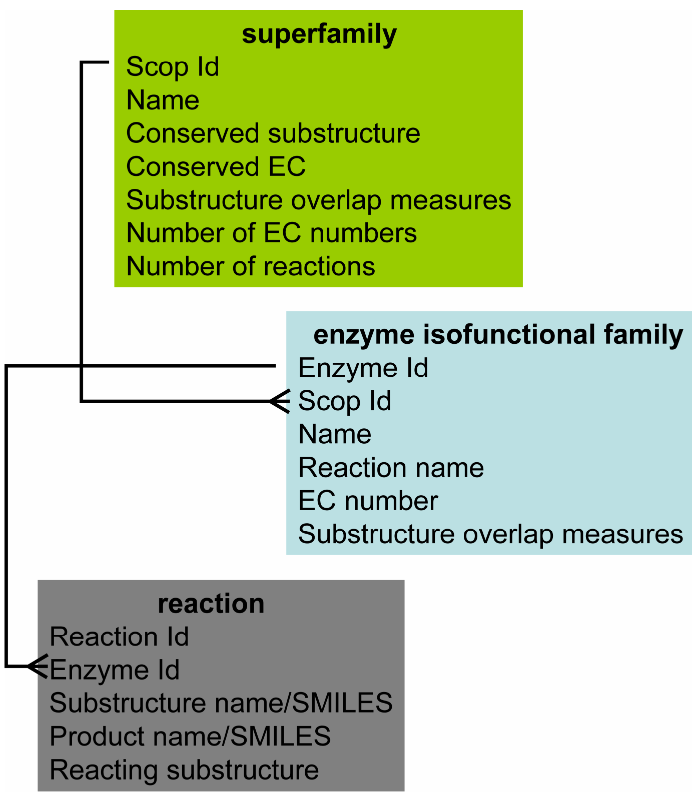


Figure 4.1. Summary of database schema.

Each colored box represents a different table in the MySQL database. The major fields is also listed in each colored box, with the name of the table in bold. The relationships between the tables are represented by lines. The ends of these lines represent the multiplicity of these relationships: one-to-many relationships are represented by the blank ends (one) and ends with "crows' feet." (many)

Table 4.1. Number of Entries in SEEER Tables

Table	Number of Entries
superfamily	42
enzyme family	149
reaction	822

4.2.3. Web interface

As of the date of submission of this thesis, the SEEER is available at http://sfldtest.rbvi.ucsf.edu/seeer. The interface is a combination of HTML pages and dynamic pages generated by Python cgi-scripts that query the MySQL database and display results. There are several ways of accessing the data: browsing superfamilies, searching by substructure, and searching by enzyme.

Selecting the option to "Browse by Superfamily" displays a summary table of all the superfamilies in the SEEER (Figure 4.2). For each superfamily, the table includes the superfamily's conserved function (both the conserved substructure and the EC digits shared by all members of the superfamily) as well as a summary of the range of reactions known to be performed by members of the superfamily (the number of unique EC numbers and the number of reactions). Because many of the enzymes in the SEEER are known to catalyze more than one reaction, either *in vivo* or *in vitro*, but an enzyme will often only be assigned one EC number, the number of reactions in a particular superfamily will often be greater than the number of unique EC numbers. Users may select a particular superfamily, which will open up a summary page describing the characteristics and reactions of that superfamily. The superfamily summary pages are described in more detail below.

Superfamilies in the SFLD					
Superfamily	SCOP ID	Conserved substructure	Conserved EC digits	Unique EC numbers	Reactions
(Phosphotyrosine protein) phosphatases II	c.45.1		3.1.3.	2	9
Alkaline phosphatase-like	c.76.1		3.1.	5	67
alpha/beta-Hydrolases	c.69.1		3.1.1.	3	13
AraD-like aldolase/epimerase	c.74.1		-	3	8

•
•
•

Figure 4.2. Database interface – Browse superfamilies.

To facilitate browsing through the superfamilies in SEEER, superfamilies and their conserved substructures as well as other summary information is displayed as a table.

A user interested in engineering an enzyme to perform a particular reaction may be interested in finding a superfamily or enzyme that is likely to be a good starting point for engineering. Selecting the "Search by Substructure" option allows the user to query the database with a SMILES string for the substrate or product of the desired reaction. The SMILES string will be used as a query against the superfamily conserved substructures and/or the reacting substructures, depending on which search options are selected. If the superfamily search option is selected, any superfamily whose conserved substructure is a substructure of the query SMILES will be returned and summarized in a

displayed table. If the enzyme search option is selected, any enzyme family with enzymes that have reacting substructures that are substructures of the query SMILES will be displayed as a table. Because an enzyme family may have multiple different reactions, there may be multiple matching reactions for a particular enzyme, but the number of matching reactions may only be a fraction of the total number of reactions performed by that enzyme. Therefore, both the number and fraction of matching reactions are reported and the results are sorted by the fraction of matching reactions.

A user may also have a particular enzyme as a starting point for entering the database. Such a user might be interested in exploring the range of reactions performed or learning which aspects of that enzymes function are conserved within its superfamily. By selecting the "Search by Enzyme" option, a user can enter in the name of the enzyme of interest or an EC number of the reaction performed by the enzyme. If a matching enzyme is found in the database, the results will be displayed as an enzyme summary table. From this table, the superfamily and other enzyme families in the superfamily are accessible through hypertext links. This table is discussed in more detail further below.

For each superfamily, users can view a superfamily summary page (Figure 4.3). The top of the page contains a table summarizing the range of reactions catalyzed by members of the superfamily and the functions conserved among members of the superfamily. This table also contains a summary of how much the conserved substructure and reacting substructures overlap. Below the summary table, the conserved substrate substructure and all of the reactions in the superfamily are displayed. The reactions can be viewed either all on one page (Show all reactions, default view) or by scrolling through one reaction at a time (Show one reaction at a time). Each reaction is

labeled with the name of the enzyme that performs that reaction and with the substrate and product(s) of the reaction and is linked to the summary pages for each enzyme family.

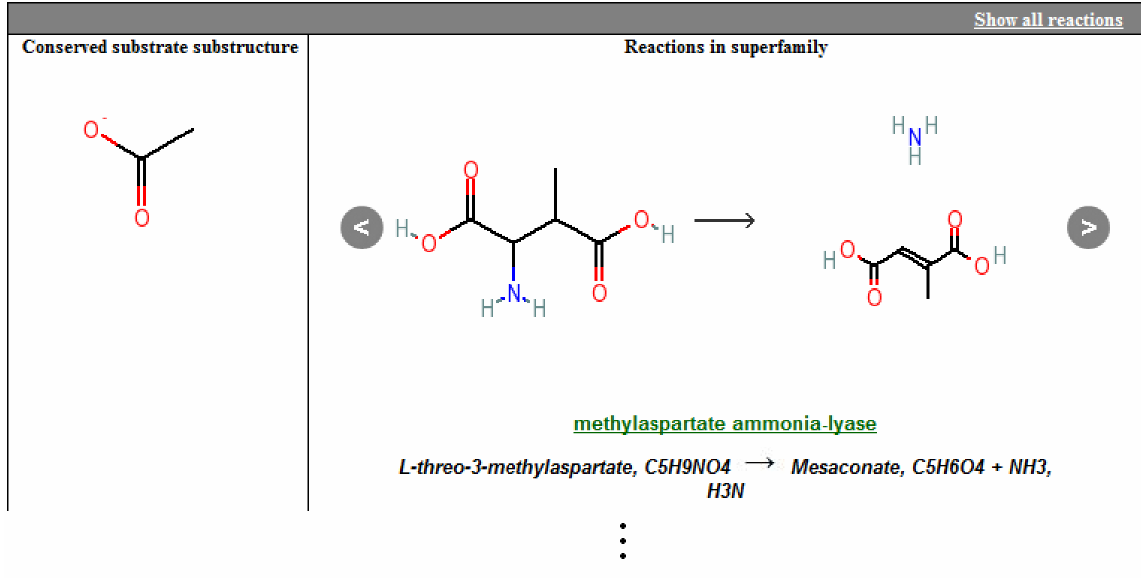


Figure 4.3. Database interface - Superfamily display page.

Superfamily summary pages display information about the superfamily including the conserved substructures, each reaction in the superfamily, the number of unique reactions in the superfamily, and measures that describe the overlap between conserved and reacting substructures.

Information for individual enzyme families in SEEER is also displayed in summary pages (Figure 4.4). The top of the page contains a table summarizing the canonical enzyme family reaction and the range of reactions catalyzed by this enzyme.

This table also contains a summary of how much the conserved substructure of the superfamily and the reacting substructures overlap. Below the summary table, there is a table showing all of the reactions catalyzed by members of the enzyme family. The reactions can be viewed either all on one page (Show all reactions, default view) or by scrolling through one reaction at a time (Show one reaction at a time). Each reaction is labeled with the substrate and product of the reaction.

Top Level			
SCOP Superfamily: Enolase C-terminal domain-like (c.1.11)			link to: SFLD SCOP
Enzyme family: muconate cycloisomerase (5.5.1.1)			link to: BRENDA
Summary of reactions in family:			
reaction:	2,5-dihydro-5-oxofuran-2-acetate = *cis,cis*-hexadienedioate		
# unique reactions in Brenda:	8		
Overlap between conserved and reacting substructures:			?
avg f_c (atoms)	0.25	avg f_c (bonds)	0.00
stdev f_c (atoms)	0.00	stdev f_c (bonds)	0.00
avg f_r (atoms)	0.32	avg f_r (bonds)	0.00
stdev f_r (atoms)	0.03	stdev f_r (bonds)	0.00

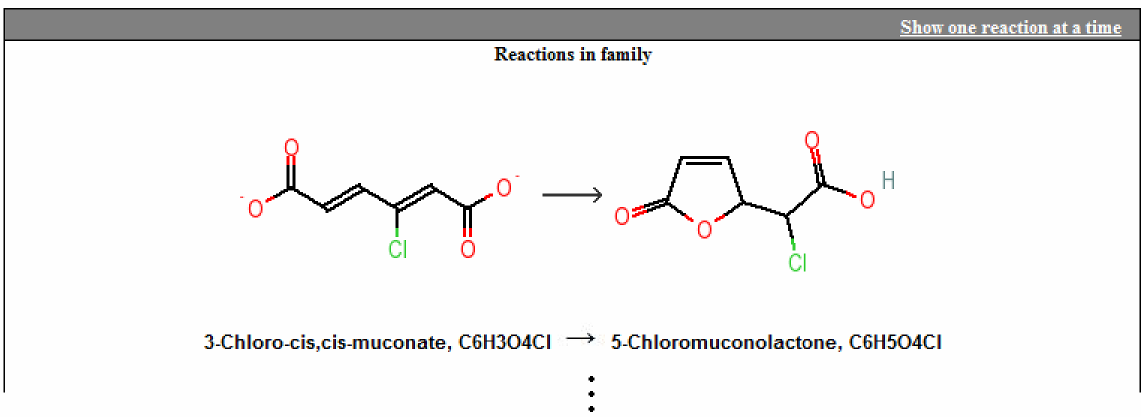


Figure 4.4. Database interface - Enzyme display page.

Enzyme summary pages show the reactions catalyzed by the particular family of enzymes and measures that describe the overlap between conserved and reacting substructures.

4.3. Conclusion

By developing this resource to share results of our prior substrate substructure analyses, we hope to facilitate the generation of hypotheses relating to enzyme evolution and for enzyme engineering. For researchers interested in a particular enzyme or superfamily, the SEEER allows for exploration of the conserved functions as well as the diversity of functions. In addition, the conserved substructures can be used as an initial prediction of function for enzymes that can be classified into a superfamily but not into a specific family. That initial prediction serves as a starting point for additional studies to determine the full and exact function of the enzyme. For enzyme engineering, the substrate and product of the desired reaction can be used to query the SEEER for superfamilies and enzymes that already perform reactions that share substructures with the desired reaction. By selecting such superfamilies and enzymes as starting points for evolution, fewer changes need to be made to achieve the desired reaction. In addition, researchers interested in predicting substrates using computational docking methodologies can use the conserved the substructure to filter ligand libraries for docking. The SEEER resource will be made publicly available to facilitate researchers interested in these goals.

Chapter 5

Target Selection and Annotation for the Structural Genomics of the Amidohydrolase and Enolase Superfamilies

5.1. Abstract

To study the physics and evolution of the substrate specificity of enzymes, we use the amidohydrolase and enolase superfamilies as model systems. Members of these superfamilies share a common TIM barrel fold and catalyze a wide range of chemical reactions. Here, we describe our work to maximize the structural coverage of the amidohydrolase and enolase superfamilies. Using sequence- and structure-based protein comparisons, we first selected 535 target proteins from a variety of genomes for high-throughput structure determination by X-ray crystallography; 63 of these targets were not previously annotated as superfamily members. To date, 20 unique structures in the amidohydrolase superfamily and 41 in the enolase superfamily have been determined, increasing the fraction of sequences in the two superfamilies that can be modeled based

on at least 30% sequence identity from 45% to 73%. The work in this chapter has been submitted for publication. The primary contributor of the full manuscript is Dr. Ursula Pieper. This chapter contains the sections that are related to my contribution to this work, the sequence and modeling analyses for structural genomics target selection.

5.2. Introduction

The goal of this work is to predict the substrate specificity of an enzyme based on its experimentally determined and/or modeled structure (Gerlt et al. 2001; Todd et al. 2001; Seibert et al. 2005; Glasner, Gerlt et al. 2006). Computational docking methodologies have been successfully used to predict enzyme specificity (Kalyanaraman et al. 2005; Hermann et al. 2007; Song et al. 2007). In the absence of an experimentally determined structure, comparative modeling can be used to predict an enzyme's structure, which then can be used for computational docking (McGovern et al. 2003). The quality of the models and the successive docking results depends on the availability of good template structures. Therefore, the successful prediction of enzyme specificity is enabled by structural genomics efforts to obtain crystallographic structures that thoroughly cover the space of enzyme sequences.

A particularly attractive opportunity to study the substrate specificity and enzymatic mechanisms from the evolutionary and physical perspectives is provided by the very large and diverse amidohydrolase and enolase superfamilies. These superfamilies are attractive targets because of significant existing knowledge about them, while there are still large areas of their sequence space where we don't have any structure or function information.

We have previously described the enolase superfamily (Section 2.1.2). The other superfamily central to this study is the amidohydrolase superfamily. The amidohydrolase superfamily members catalyze the hydrolysis of a wide range of substrates bearing amide or ester functional groups at carbon and phosphorus centers (Seibert et al. 2005). A common feature for this superfamily is a mononuclear or binuclear metal center coordinated in a $(\beta/\alpha)_8$-barrel (TIM barrel) polypeptide chain fold. The active site is formed by loops at the C-terminal ends of the β-strands. This superfamily is currently organized into 36 named families based on the experimentally verified catalytic reactions. The sequences are also clustered into 90 subgroups based on their chemical reaction catalyzed and active site similarities and a common sequence identity of at least 40% (Pegg et al. 2006).

NIH guidelines allow for 15% of Protein Structure Initiative (PSI) structures to be "community-nominated" targets (Norvell et al. 2007). A substantial fraction of the New York Research Center for Structural Genomics (NYSGXRC) community targets are members of the amidoydrolase and enolase superfamilies that we have nominated. To date, our work has resulted in 25 amidohydrolase superfamily and 50 enolase superfamily structures, contributing substantially to the total structures in the Protein Data Bank (PDB; 6/16/08) (amidohydrolase: 154 and enolase: 89) (Berman et al. 2007).

We begin by outlining the data sources and methods used for target selection and structure-based functional annotation (Methods, Section 5.3). Then, we present the results of the target selection process, the status of the selected targets in the structural genomics pipeline, and the improvement in the modeling of the amidohydrolase and enolase

superfamilies made possible by the new crystallographic structures (Results and Discussion, Section 5.4).

5.3. Methods

5.3.1. Target selection

Target selection begins by identifying known members of the superfamilies (seed sequences), followed by filtering to obtain an initial target list. To identify additional members, we applied sequence- and structure-based expansion methods, followed by filtering for preferred source organisms. Superfamily membership for the additional targets was verified by inspecting their sequences for probable catalytic sites.

Seed sequence sources

Verified amidohydrolase and enolase superfamily sequences (ie, seed sequences) were obtained from the SFLD, which has been described in more detail in Section 2.3.2. In June 2005, when our target list was constructed, SFLD contained 3701 sequences for the amidohydrolase superfamily and 1795 sequences in the enolase superfamily.

Filtering of seed sequences

To select targets that share ~30% or less amino acid sequence identity over at least 70% of their length to a known three-dimensional structure, the seed sequences were processed using the automated comparative modeling server MODWEB (http://salilab.org/modweb) (Eswar et al. 2003).

Sequence-based expansion of amidohydrolase and enolase superfamily members

We identified additional potential superfamily sequences that were not present in the seed sequence pools. For each seed sequence, homologous sequences in the

UNIPROT database (Wu et al. 2006) were identified by the BUILD_PROFILE routine of MODELLER-9 (Eswar et al. 2003). BUILD_PROFILE is an iterative database-searching tool that relies on local dynamic programming to generate alignments and a robust estimate of their statistical significance.

Structure-based expansion of amidohydrolase superfamily members

In addition to the SFLD entries, we also used the known amidohydrolase superfamily structures to find additional potential amidohydrolase superfamily members (this expansion was not performed for the enolase superfamily). We began by clustering 100 PDB files (only monomeric structure) containing known amidohydrolase superfamily structures (June 2005) at 80% sequence identity. The resulting 48 non-redundant structures were used for comparative modeling using the automated modeling server MODWEB (Eswar et al. 2003).

For each structure, PSIBLAST (Altschul et al. 1997) was used to find putative homologs in UNIPROT, which were modeled using the query structure as a template. Known amidohydrolase superfamily members were excluded. All models were deposited in our comprehensive MODBASE database of comparative protein structure models (http://salilab.org/modbase/; dataset model set ah_structures) (Pieper et al. 2006). To eliminate sequences that are likely members of other superfamilies, the homologs were subjected to standard comparative modeling with MODWEB using all non-redundant chains in the PDB as potential templates.

Filtering by organism

While seed sequences could come from any genome, the additional amidohydrolase superfamily sequences identified by sequence- and structure-based

expansions were filtered for ease of cloning to include only 79 organisms with genomic DNA available to NYSGXRC in 2005 and the marine metagenome from the Sargasso Sea sequencing project (Venter et al. 2004).

Verification of catalytic residues

The resulting putative amidohydrolase superfamily sequences were aligned to amidohydrolase superfamily HMMs in SFLD and manually inspected for probable catalytic residues. The final target list only includes sequences with at least 70% of the catalytic residues present.

5.3.2. Analysis of the target structures

The amidohydrolase and enolase superfamilies were annotated using several computational tools. Cytoscape clustering gives an overview of how the targets are distributed across the superfamily (Shannon et al. 2003). Finally, template-based modeling determines how many new sequences can be modeled with the new structural information (Eswar et al. 2003).

Sequence clustering by Cytoscape

All-by-all BLAST searches were performed for the superfamily members and these results were used to construct Cytoscape (Shannon et al. 2003) networks for the amidohydrolase superfamily and the mandelate racemase, glucarate dehydratase, mannonate dehydratase, and muconate cycloisomerase subgroups of the enolase superfamily. A node represents a single sequence and an edge is shown for the most significant BLAST e-value connecting two sequences when it is better than an e-value cutoff chosen empirically to achieve the best "visual" separation of the clusters (10^{-10} for

the amidohydrolase superfamily, 10^{-40} for the enolase superfamily). The nodes were arranged using the 'organic' layout.

Template-based modeling by MODWEB

The new NYSGXRC crystallographic structures were submitted to MODWEB (Eswar et al. 2003) to serve as templates to model all of the identifiable homologs of the input structure in the non-redundant database of protein sequences nr (Wheeler et al. 2008); these homologs were identified during ten PSI-BLAST iterations of the template sequence against nr (e-value cutoff is 0.0001) The results are available at http://salilab.org/modbase/models_nysgxrc_latest.html.

5.4. Results and Discussion

We first present the results of the target selection procedure. We also describe the current snapshot of the progress of the targets through our structural genomics pipeline (June 2008). We then indicate how the resulting crystallographic structures are distributed across the two superfamilies. Finally, we determine the number of protein sequences in the comprehensive sequence databases that are detectably related to these protein structures (ie, the modeling leverage).

5.4.1. Target selection

Given the capacities of NYSGXRC, the goal was to identify approximately 500 target sequences, approximately evenly distributed between the two superfamilies. These targets were obtained by selecting representatives from previously identified superfamily members as well as by identifying new superfamily members in a select set of genomes (See Methods, Section 5.3).

Targets for the amidohydrolase superfamily

From the SFLD, we obtained a list of 3701 amidohydrolase superfamily members. The first filtering step resulted in 1918 sequences with less than 30% sequence identity to a known structure and at least 250 amino acid residues in length, originating from 424 organisms. We chose a 30% sequence identity limit because at approximately this level of sequence identity, obtaining accurate target-template alignments becomes more difficult and homology modeling begins to incur significant errors (Sanchez et al. 1997; Vitkup et al. 2001).

These 1918 sequences were further filtered manually to obtain the reduced set of 224 target sequences. The selected amidohydrolase superfamily members are evenly distributed among the various clades of the superfamily, thus representing the diversity within the superfamily.

In addition to the known superfamily members, the sequence- and structure-based expansions detected 63 putative amidohydrolase superfamily members that were not initially in the SFLD (A table listing all amidohydrolase and enolase superfamily targets can be found at http://salilab.org/projects/enspec/target_list.html). These new potential targets fall into two categories: (i) divergent sequences that were detected by the sequence-based approach (Figure 5.1, blue box) and (ii) divergent sequences that were detected by the structure-based approach (Figure 5.1, pink box). Of the 63 putative amidohydrolase superfamily sequences, 50 were subsequently verified using the SFLD update procedure. The presence of probable catalytic residues for the remaining 13 targets was verified manually. Nine of these 13 sequences were detected by both the sequence- and structure-based approaches, and four sequences were only detected by the

structure-based approach. Thus, the sequence- and structure-based approaches yielded 13 additional targets that could not be identified with previously available protocols as amidohydrolase superfamily members (corresponding to 21% of the new putative members of the amidohydrolase superfamily).

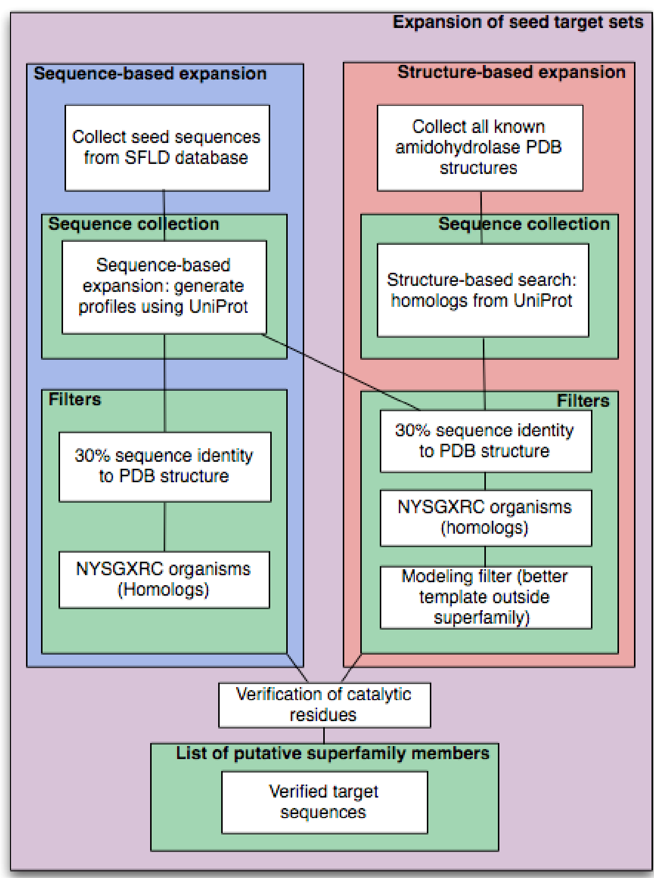


Figure 5.1. Flowchart of the target expansion strategy

Sequence-based target expansion (left) and structure-based target expansion (right).

The final amidohydrolase target list, including both previously identified and newly identified sequences, comprises 287 sequences from 53 organisms that cover 22 (61%) named families in the superfamily (Figure 5.2).

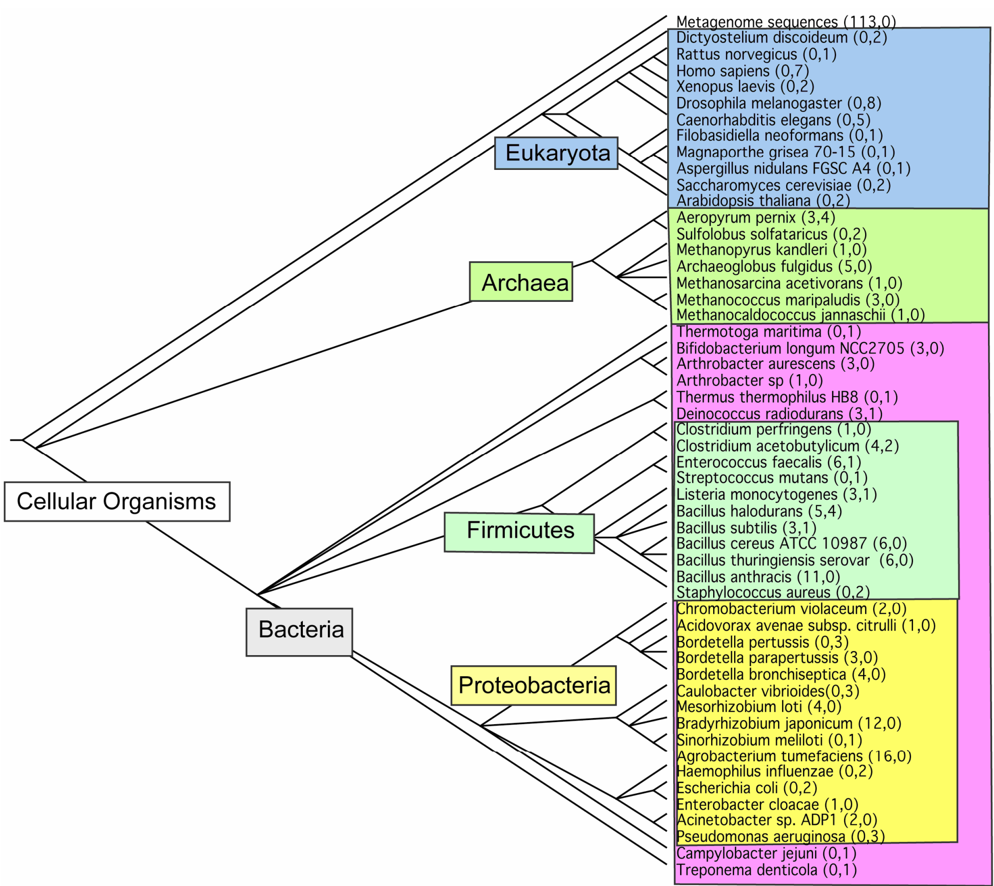


Figure 5.2. Phylogenetic tree of the organisms for the selected amidohydrolase targets

The numbers in parentheses represent the number of targets for confirmed (first number) and putative (second number) amidohydrolase superfamily members.

Targets for the enolase superfamily

We used a simpler selection scheme for the enolase superfamily members. Of the 1795 sequences already established as enolase superfamily members, we selected as targets the 255 sequences with less than 30% sequence identity to a known structure and at least 250 residues in length, originating from 98 organisms.

A complete list of the selected amidohydrolase and enolase superfamily targets can be found at http://salilab.org/projects/enspec/.

Structural genomics pipeline attrition

To date, structure determinations have been attempted for 254 amidohydrolase (88%) and 206 enolase (80%) superfamily members by the NYSGXRC/ENSPEC X-ray crystallographic structure determination pipeline. Progress to date and attrition rate at each stage of the pipeline are documented in Table 5.1 (June 2008). The project has not yet been completed, and a number of targets are still progressing through the pipeline. Therefore, the final overall success rate should be higher than that presented in Table 5.1.

Table 5.1. Success rates for the steps in the structural genomics pipeline as of June 2008.

Step	Amidohydrolase superfamily		Enolase superfamily		Both superfamilies	
	Total	Fraction [%]	Total	Fraction [%]	Total	Fraction [%]
Selected	279		222		501	
Cloned	254	91	206	93	460	92
Expressed	225	88	177	86	402	87
Soluble	167	74	112	63	279	69
Purified	110	66	67	60	177	63
Crystallized	~63	~57	~44	~66	~107	~60
Unique Structures	20	~32	41	~93	61	~57
All Structures	25		50		75	

5.4.2. Analysis of the resulting crystallographic structures

Leverage of new crystallographic structures by modeling

To determine the impact of a structure on the structural mapping of the protein sequence space, we determine how many known protein sequences can be modeled based on the structure (ie, the modeling leverage). Each enolase structure is a useful template for calculating comparative models for 2500 to 3200 other protein sequences in the Genbank nr database; a template is considered useful when the resulting model is based on a significant PSI-BLAST E-value (0.0001) or a favorable GA341 model score (>0.7). In contrast, the amidohydrolase superfamily structures fall into two categories: most are

detectably related to 3000 - 3800 other proteins, but five structures (PDB Codes: 2I5G, 2Q01, 2Q6E, 2RAG, and 3B40) are related to a significantly smaller number of sequences (approximately 300 - 1000).

The average number of models per structure is significantly higher for the amidohydrolase and enolase superfamilies than for all structures determined by NYSGXRC (2,681versus 1,964) (as of May 2007) (Table 5.2). This difference reflects the relatively large sizes of the amidohydrolase and enolase superfamilies. The number of sequences that can be modeled based on target-template sequence identity higher than 30% is significantly lower for the amidohydrolase and enolase superfamilies structures than for the full NYSGXRC structure set (3% versus 11%), due to the relatively high diversity in the amidohydrolase and enolase superfamilies.

Table 5.2. Comparison of template-based modeling statistics for the 63 ENSPEC/NYSGXRC structures and all 327 NYSGXRC structures (May 2007).

An acceptable model is defined to be based on a significant PSI-BLAST E-value (0.0001) or a favorable GA341 model score (>0.7).

	Amidohydrolase and enolase superfamily members	All
Average number of sequences with acceptable models	2681	1964
Minimum / maximum number of sequences with acceptable models	189/3693	30/6320
Average number of sequences with >50% sequence identity, at least 50% coverage	15	20
Average number of sequences with 30-50% sequence identity, at least 50% coverage	59	113
Average number of sequences with <30% sequence identity, at least 50% coverage	2572	1400

Upon initiation of this effort in June 2005, 45% of all known members of the amidohydrolase and enolase superfamilies were related to a known structure with a

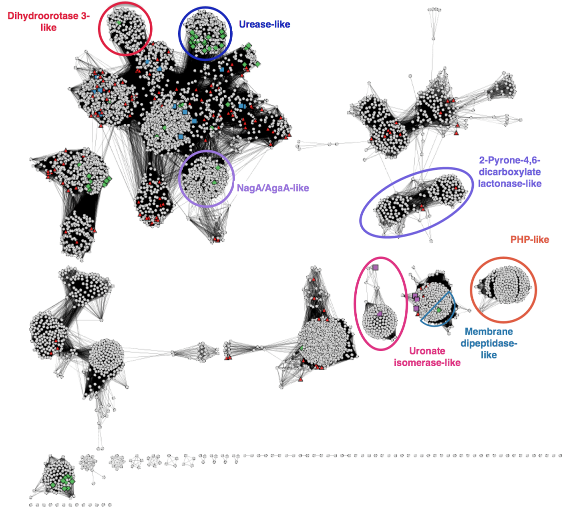
sequence identity higher than 30%. Due to the increased number of templates contributed by our consortia, this number increased to 73% from 45%.

The total number of unique sequences modeled using the new amidohydrolase and enolase superfamily structures is 11,097, approximately 30% more than the number of known sequences from the amidohydrolase and enolase superfamilies. Among these additional sequences, we expect both currently unidentified members of the amidohydrolase and enolase superfamilies as well as members of other superfamilies with the TIM barrel fold.

Distribution of targets over the amidohydrolase and enolase superfamilies

For large groups of related sequences, such as the amidohydrolase superfamily, a "network" visualization of their relationships is helpful in generating hypotheses on how various enzymes in the superfamily evolved, and on how closely the subgroups are related to each other. We have plotted Cytoscape networks for the amidohydrolase and enolase superfamilies, based on clustering by sequence similarity (Figure 5.3).

a



b

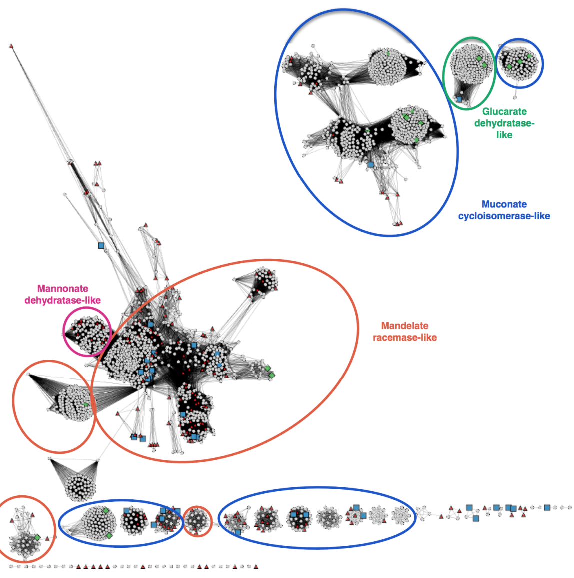


Figure 5.3. Cytoscape clustering for the amidohydrolase superfamily (a) and enolase superfamily (b). Green diamonds: Structures determined prior to the start of the ENSPEC/NYSGXRC project in June 2005. Red triangles: Superfamily members in the target list. Blue squares: All other structures determined by ENSPEC/NYSGXRC. (a) The most homogeneous subgroups have been named. Purple squares: Five divergent structures determined by ENSPEC/NYSGXRC. (b) Cytoscape clustering for the enolase superfamily. Subgroup clusters are marked for four subgroups.

Many subgroups in the large amidohydrolase superfamily, such as the urease-like subgroup and the uronate isomerase-like subgroup, are distinctly separated from the other superfamily members. For the enolase superfamily, we chose to generate a Cytoscape network that represents only four subgroups, containing the majority of the targets. The targets were mostly chosen from the mandelate racemase-like subgroup, because it is the largest subgroup with little previous structural coverage, and from the more divergent muconate cycloisomerase subgroup. The Cytoscape networks illustrate that the targets and the resulting structures are indeed concentrated in regions of superfamily sequence space that lacked structural characterization prior to the start of the project, as desired for our target selection.

5.5. Conclusion

We have made significant progress towards characterizing the structures in the amidohydrolase and enolase superfamilies. New members of the amidohydrolase superfamily have been identified through a combination of sequence- and structure-based expansions of the pool of known superfamily members. The structure-based expansion was particularly successful in identifying previously unrecognized superfamily members. The 63 crystallographic structures from the structural genomics pipeline increased the fraction of the sequences in these two superfamilies that can be modeled based on at least 30% sequence identity from 45% to 73%. This demonstrates the power of combining sequence- and structure-based approaches for the structural genomics of two large and diverse enzyme superfamilies.

Chapter 6

Conclusion and Future Directions

This dissertation has presented a body of work to study enzyme evolution by focusing on the substrates and products of the enzymes and how they vary over enzyme evolution. Our approaches also include sequence-based and structure-based analyses, which due to the interconnectedness of enzyme sequence, structure, and function, are closely tied to the primary focus of this work, the analyses of enzyme function. Using computational methods for these analyses ensures that our analyses can be conducted systematically and on large sets of data, which facilitates the determination of general patterns in enzyme evolution. In addition, the computational nature of the analyses facilitates the application of our results to other areas of research including function prediction and enzyme engineering.

The first studies described in this dissertation (Chapter 2) describe sequence and structure-based studies to trace the evolution of enzyme function in a particular superfamily. These two studies demonstrate the potential of phylogenetic, evolutionary trace, and structure comparison methods as well as some of their limitations, especially when the evolution of function is complicated.

The central piece of this body of work, described in Chapter 3, is the study on the conservation patterns in the substrates of enzyme superfamilies. Using newly developed automated methods to study many superfamilies, we determined that the previous theories of how enzymes evolve were inadequate for describing the range of variation seen among superfamilies. In addition, the results of this substrate substructure analyses can be used to improve the precision of protein function prediction and to guide efforts in enzyme engineering. The Substructures for Enzyme Evolution and Engineering Resource was developed to facilitate researchers interested in these applications (Chapter 4). This resource, which will be publicly accessible, allows researchers to search and explore the substrates, products, reactions, and how these are conserved among superfamilies and their enzymes. The resource can also be searched, using the target substrate or product, to find superfamilies and enzymes that are promising starting points for enzyme engineering.

Because the SEEER is a new resource, there are currently no examples of successful engineering of enzymes that have been based on hypotheses from the SEEER. We plan to make this resource available to researchers interested in engineering enzymes and hope that the information about how enzyme functions have evolved can be used as a model for successful engineering. In addition to engineering enzymes to perform different reactions and/or use different substrates, there are additional engineering strategies that can be combined to tightly control the function in biological systems (Arkin et al. 2006). For example, protein-protein interfaces can be engineered to modulate the functions of the component proteins (Kortemme et al. 2004). As progress is made in our ability to engineer different individual components of biological systems, we

can move toward more challenging engineering involving multiple interdependent components.

The study of enzyme function and evolution is enabled by knowing the functions of as many enzymes as possible, which is, in turn, enabled by having many structures. Experimentally determined structures can be used to determine how enzymes have evolved and thus, how their functions have evolved (For an example, see (Ojha et al. 2007)). Computational docking methods to predict enzyme substrates are enabled by studies like the one described in Chapter 3 as well as by having many structures. When there are no experimental structures available for docking, comparative modeling can be used to predict structures. For the study described in Chapter 5, the selection of targets for structural genomics efforts led to an increase in the number of structures available to guide studies to determine the functions. Additionally, selecting targets that were evenly distributed throughout the superfamilies enabled us to maximize the number of additional sequences that can be modeled.

Because the product of one enzyme is the substrate for the next enzyme in the pathway, enzymes in the same pathway will share similarities in their substrates. The goal of ongoing work is to leverage this similarity among substrates within pathways to improve our ability to predict enzyme substrate specificity (Figure 6.1). The first step in this strategy involves using docking methods to screen ligand libraries for substrates and products of multiple members of enzymes coded by a single operon. This requires either a crystal structure of the enzyme or of a homologous enzyme that can be used as a template for comparative modeling. For cases in which the order of the enzymes in the pathway is unknown, the SEA method (Keiser et al. 2007) to relate enzymes by ligand

list similarity can be used to cluster the enzymes' docking hit lists to predict whether the enzymes in the operon are in the same pathway as well as the likely order of enzymes. With the pathway order, substructure-based analyses can be used to find substructures that occur frequently in the hit lists of neighboring enzymes of the pathway to further enrich the docking results for true substrates and products.

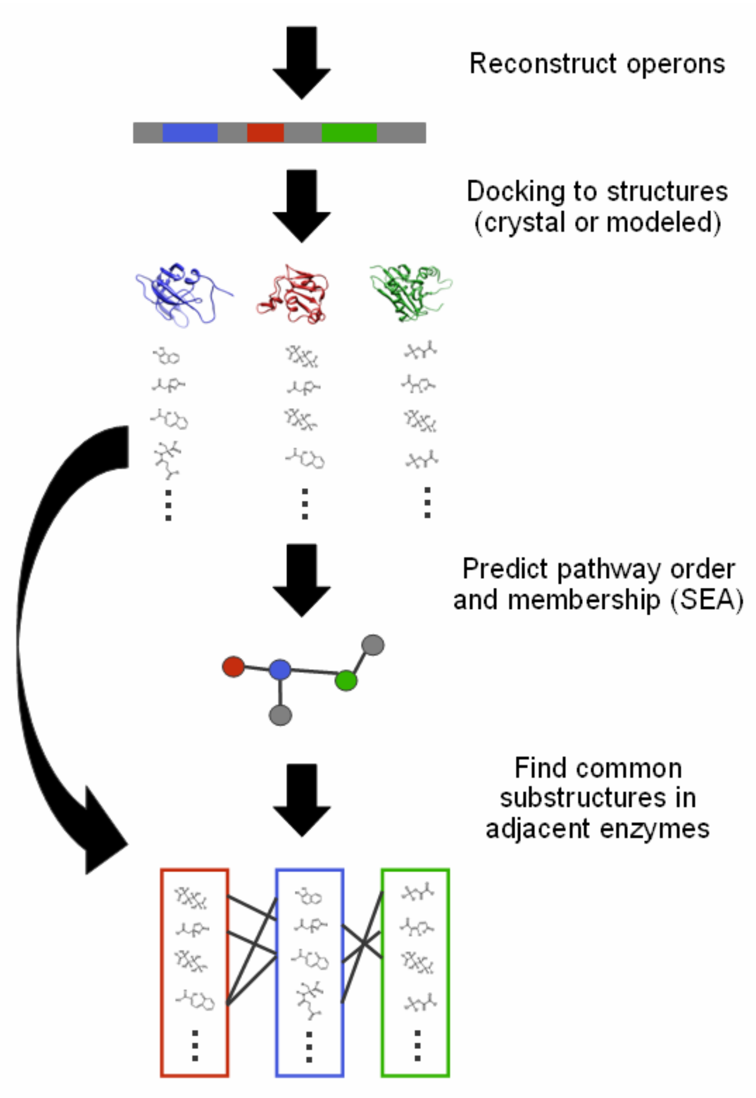


Figure 6.1. Flowchart of the substrate prediction strategy

Analysis of conservation in enzyme substrates and products represents an important first step in the study of the evolution of function. There are additional strategies that can supplement the initial strategy that we developed. For example, reaction steps can be described and compared to determine how enzyme functions are conserved or vary during enzyme evolution (O'Boyle et al. 2007). This type of methodology is currently being further developed and can be combined with the analysis of ligand structures to get a more complete view of how enzymes evolve. By combining these orthogonal strategies, we would get closer to a systematic and detailed representation of enzyme function that could replace the outdated EC system. This improved representation of enzyme function would facilitate the study, prediction, and annotation of enzyme function.

The more we study enzyme function – both focused and large-scale studies – the more we are finding that enzyme function is very complex. Highly dissimilar enzymes can share the same function (Glasner, Fayazmanesh et al. 2006), while highly similar enzymes can have differing functions (Seffernick et al. 2001). How many changes in sequence and structure can occur in an enzyme while maintaining the same function? Many enzymes are known to be promiscuous for multiple functions (O'Brien et al. 1999) and because of limits in our ability to test large ranges of possible functions, many additional enzymes are likely to be promiscuous. What role does promiscuity play in the evolution of new functions? Some case studies suggest that promiscuous proteins more likely to evolve different function, but this hypothesis has not been thoroughly tested. To answer these questions about the complex process of enzyme evolution will require more data, consistent and accurate database annotations, improved ways of encoding function,

and better ways to evaluate promiscuity (Nath et al. 2008). The work in this dissertation provides a foundation for further work in this field that will be necessary to develop comprehensive solutions to these challenges

References

Aharoni, A., L. Gaidukov, et al. (2005). "The 'evolvability' of promiscuous protein functions." Nat Genet **37**(1): 73-6.

Allen, K. N. and D. Dunaway-Mariano (2004). "Phosphoryl group transfer: evolution of a catalytic scaffold." Trends Biochem Sci **29**(9): 495-503.

Altekar, G., S. Dwarkadas, et al. (2004). "Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference." Bioinformatics **20**(3): 407-15.

Altschul, S. F., W. Gish, et al. (1990). "Basic local alignment search tool." J Mol Biol **215**(3): 403-10.

Altschul, S. F., T. L. Madden, et al. (1997). "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." Nucleic Acids Res **25**(17): 3389-402.

Alves, R., R. A. Chaleil, et al. (2002). "Evolution of enzymes in metabolism: a network perspective." J Mol Biol **320**(4): 751-70.

Arkin, A. P. and D. A. Fletcher (2006). "Fast, cheap and somewhat in control." Genome Biol **7**(8): 114.

Ashburner, M., C. A. Ball, et al. (2000). "Gene ontology: tool for the unification of biology. The Gene Ontology Consortium." Nat Genet **25**(1): 25-9.

Babbitt, P. C. (2003). "Definitions of enzyme function for the structural genomics era." Curr Opin Chem Biol **7**(2): 230-7.

Babbitt, P. C. and J. A. Gerlt (1997). "Understanding enzyme superfamilies. Chemistry As the fundamental determinant in the evolution of new catalytic activities." J Biol Chem **272**(49): 30591-4.

Babbitt, P. C., M. S. Hasson, et al. (1996). "The enolase superfamily: a general strategy for enzyme-catalyzed abstraction of the alpha-protons of carboxylic acids." Biochemistry **35**(51): 16489-501.

Baker, D. and A. Sali (2001). "Protein structure prediction and structural genomics." Science **294**(5540): 93-6.

Barthelmes, J., C. Ebeling, et al. (2007). "BRENDA, AMENDA and FRENDA: the enzyme information system in 2007." Nucleic Acids Res **35**(Database issue): D511-4.

Bashton, M. and C. Chothia (2007). "The generation of new protein functions by the combination of domains." Structure **15**(1): 85-99.

Bergmeyer, H. (1974). Methods of Enzymatic Analysis. New York, NY, Academic Press.

Berman, H., K. Henrick, et al. (2007). "The worldwide Protein Data Bank (wwPDB): ensuring a single, uniform archive of PDB data." Nucleic Acids Res **35**(Database issue): D301-3.

Bessman, M. J., D. N. Frick, et al. (1996). "The MutT proteins or "Nudix" hydrolases, a family of versatile, widely distributed, "housecleaning" enzymes." J Biol Chem **271**(41): 25059-62.

Brenner, S. E. (1999). "Errors in genome annotation." Trends Genet **15**(4): 132-3.

Brenner, S. E., C. Chothia, et al. (1998). "Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships." Proc Natl Acad Sci U S A **95**(11): 6073-8.

Briggs, G. E. and J. B. Haldane (1925). "A Note on the Kinetics of Enzyme Action." Biochem J **19**(2): 338-9.

Brown, S. D., J. A. Gerlt, et al. (2006). "A gold standard set of mechanistically diverse enzyme superfamilies." Genome Biol **7**(1): R8.

Chandonia, J. M. and S. E. Brenner (2006). "The impact of structural genomics: expectations and outcomes." Science **311**(5759): 347-51.

Chiang, R. A., A. Sali, et al. (2008). "Evolutionarily conserved substrate substructures for automated annotation of enzyme superfamilies." PLoS Comput Biol **4**(8): e1000142.

Copley, S. D. (2000). "Evolution of a metabolic pathway for degradation of a toxic xenobiotic: the patchwork approach." Trends Biochem Sci **25**(6): 261-5.

Copley, S. D. (2003). "Enzymes with extra talents: moonlighting functions and catalytic promiscuity." Curr Opin Chem Biol **7**(2): 265-72.

Davis, F. P. and A. Sali (2005). "PIBASE: a comprehensive database of structurally defined protein interfaces." Bioinformatics **21**(9): 1901-7.

Devos, D. and A. Valencia (2000). "Practical limits of function prediction." Proteins **41**(1): 98-107.

Devos, D. and A. Valencia (2001). "Intrinsic errors in genome annotation." Trends Genet **17**(8): 429-31.

Diaz-Mejia, J. J., E. Perez-Rueda, et al. (2007). "A network perspective on the evolution of metabolism by gene duplication." Genome Biol **8**(2): R26.

Driscoll, J. R. and H. W. Taber (1992). "Sequence organization and regulation of the Bacillus subtilis menBE operon." J Bacteriol **174**(15): 5063-71.

Edgar, R. C. (2004). "MUSCLE: multiple sequence alignment with high accuracy and high throughput." Nucleic Acids Res **32**(5): 1792-7.

Eswar, N., B. John, et al. (2003). "Tools for comparative protein structure modeling and analysis." Nucleic Acids Res **31**(13): 3375-80.

Felsenstein, J. (2004). "PHYLIP (Phylogeny Inference Package) version 3.6." Distributed by the author. Department of Genome Sciences, University of Washington, Seattle.

Fersht, A. (1985). Enzyme Structure and Mechanism (2nd ed). New York, W. H. Freeman and Co.

Frazer, K. A., L. Elnitski, et al. (2003). "Cross-species sequence comparisons: a review of methods and available resources." Genome Res **13**(1): 1-12.

Friedberg, I. (2006). "Automated protein function prediction--the genomic challenge." Brief Bioinform **7**(3): 225-42.

Gerlt, J. A. (2007). "A Protein Structure (or Function ?) Initiative." Structure **15**(11): 1353-6.

Gerlt, J. A. and P. C. Babbitt (1998). "Mechanistically diverse enzyme superfamilies: the importance of chemistry in the evolution of catalysis." Curr Opin Chem Biol **2**(5): 607-12.

Gerlt, J. A. and P. C. Babbitt (2001). "Divergent evolution of enzymatic function: mechanistically diverse superfamilies and functionally distinct suprafamilies." Annu Rev Biochem **70**: 209-46.

Gerlt, J. A., P. C. Babbitt, et al. (2005). "Divergent evolution in the enolase superfamily: the interplay of mechanism and specificity." Arch Biochem Biophys **433**(1): 59-70.

Gerlt, J. A. and F. M. Raushel (2003). "Evolution of function in (beta/alpha)8-barrel enzymes." Curr Opin Chem Biol **7**(2): 252-64.

Gilks, W. R., B. Audit, et al. (2002). "Modeling the percolation of annotation errors in a database of protein sequences." Bioinformatics **18**(12): 1641-9.

Glasner, M. E., N. Fayazmanesh, et al. (2006). "Evolution of structure and function in the o-succinylbenzoate synthase/N-acylamino acid racemase family of the enolase superfamily." J Mol Biol **360**(1): 228-50.

Glasner, M. E., J. A. Gerlt, et al. (2006). "Evolution of enzyme superfamilies." Curr Opin Chem Biol **10**(5): 492-7.

Glasner, M. E., J. A. Gerlt, et al. (2007). "Mechanisms of protein evolution and their application to protein engineering." Adv Enzymol Relat Areas Mol Biol **75**: 193-239, xii-xiii.

Gulick, A. M., B. K. Hubbard, et al. (2000). "Evolution of enzymatic activities in the enolase superfamily: crystallographic and mutagenesis studies of the reaction catalyzed by D-glucarate dehydratase from Escherichia coli." Biochemistry **39**(16): 4590-602.

Hegyi, H. and M. Gerstein (1999). "The relationship between protein structure and function: a comprehensive survey with application to the yeast genome." J Mol Biol **288**(1): 147-64.

Hermann, J. C., E. Ghanem, et al. (2006). "Predicting substrates by docking high-energy intermediates to enzyme structures." J Am Chem Soc **128**(49): 15882-91.

Hermann, J. C., R. Marti-Arbona, et al. (2007). "Structure-based activity prediction for an enzyme of unknown function." Nature **448**(7155): 775-9.

Holden, H. M., M. M. Benning, et al. (2001). "The crotonase superfamily: divergently related enzymes that catalyze different reactions involving acyl coenzyme a thioesters." Acc Chem Res **34**(2): 145-57.

Holliday, G. L., D. E. Almonacid, et al. (2007). "MACiE (Mechanism, Annotation and Classification in Enzymes): novel tools for searching catalytic mechanisms." Nucleic Acids Res **35**(Database issue): D515-20.

Holm, L. and C. Sander (1996). "Mapping the protein universe." Science **273**(5275): 595-603.

Holm, L. and C. Sander (1997). "An evolutionary treasure: unification of a broad set of amidohydrolases related to urease." Proteins **28**(1): 72-82.

Horowitz, N. H. (1945). "On the Evolution of Biochemical Syntheses." Proc Natl Acad Sci U S A **31**(6): 153-7.

Horowitz, N. H. (1965). The evolution of biochemical syntheses - retrospect and prospect. Evolving genes and proteins. V. Bryson and H. J. Vogel. New York, Academic Press**:** 15-.

Hughes, A. L. (1994). "The evolution of functionally novel proteins after gene duplication." Proc Biol Sci **256**(1346): 119-24.

James, L. C. and D. S. Tawfik (2003). "Conformational diversity and protein evolution--a 60-year-old hypothesis revisited." Trends Biochem Sci **28**(7): 361-8.

Jensen, R. A. (1976). "Enzyme recruitment in evolution of new function." Annu Rev Microbiol **30**: 409-25.

Jewett, A. I., C. C. Huang, et al. (2003). "MINRMS: an efficient algorithm for determining protein structure similarity using root-mean-squared-distance." Bioinformatics **19**(5): 625-34.

Johnson, T. W., G. Shen, et al. (2000). "Recruitment of a foreign quinone into the A(1) site of photosystem I. I. Genetic and physiological characterization of phylloquinone biosynthetic pathway mutants in Synechocystis sp. pcc 6803." J Biol Chem **275**(12): 8523-30.

Jones, D. T., W. R. Taylor, et al. (1992). "The rapid generation of mutation data matrices from protein sequences." Comput Appl Biosci **8**(3): 275-82.

Kalyanaraman, C., K. Bernacki, et al. (2005). "Virtual screening against highly charged active sites: identifying substrates of alpha-beta barrel enzymes." Biochemistry **44**(6): 2059-71.

Keiser, M. J., B. L. Roth, et al. (2007). "Relating protein pharmacology by ligand chemistry." Nat Biotechnol **25**(2): 197-206.

Khersonsky, O., C. Roodveldt, et al. (2006). "Enzyme promiscuity: evolutionary and mechanistic aspects." Curr Opin Chem Biol **10**(5): 498-508.

Kitchen, D. B., H. Decornez, et al. (2004). "Docking and scoring in virtual screening for drug discovery: methods and applications." Nat Rev Drug Discov **3**(11): 935-49.

Klenchin, V. A., E. A. Taylor Ringia, et al. (2003). "Evolution of enzymatic activity in the enolase superfamily: structural and mutagenic studies of the mechanism of the reaction catalyzed by o-succinylbenzoate synthase from Escherichia coli." Biochemistry **42**(49): 14427-33.

Koike-Takeshita, A., T. Koyama, et al. (1997). "Identification of a novel gene cluster participating in menaquinone (vitamin K2) biosynthesis. Cloning and sequence determination of the 2-heptaprenyl-1,4-naphthoquinone methyltransferase gene of Bacillus stearothermophilus." J Biol Chem **272**(19): 12380-3.

Kortemme, T. and D. Baker (2004). "Computational design of protein-protein interactions." Curr Opin Chem Biol **8**(1): 91-7.

Kotera, M., Y. Okuno, et al. (2004). "Computational assignment of the EC numbers for genomic-scale analysis of enzymatic reactions." J Am Chem Soc **126**(50): 16487-98.

Krishnamurthy, N. and K. V. Sjolander (2005). "Basic protein sequence analysis." Curr Protoc Mol Biol **Chapter 19**: Unit 19 5.

Landro, J. A., J. A. Gerlt, et al. (1994). "The role of lysine 166 in the mechanism of mandelate racemase from Pseudomonas putida: mechanistic and crystallographic evidence for stereospecific alkylation by (R)-alpha-phenylglycidate." Biochemistry **33**(3): 635-43.

Lebioda, L. and B. Stec (1988). "Crystal structure of enolase indicates that enolase and pyruvate kinase evolved from a common ancestor." Nature **333**(6174): 683-6.

Lichtarge, O., H. R. Bourne, et al. (1996). "An evolutionary trace method defines binding surfaces common to protein families." J Mol Biol **257**(2): 342-58.

Light, S. and P. Kraulis (2004). "Network analysis of metabolic enzyme evolution in Escherichia coli." BMC Bioinformatics **5**: 15.

Lupyan, D., A. Leo-Macias, et al. (2005). "A new progressive-iterative algorithm for multiple structure alignment." Bioinformatics **21**(15): 3255-63.

Madabushi, S., A. K. Gross, et al. (2004). "Evolutionary trace of G protein-coupled receptors reveals clusters of residues that determine global and class-specific functions." J Biol Chem **279**(9): 8126-32.

Madabushi, S., H. Yao, et al. (2002). "Structural clusters of evolutionary trace residues are statistically significant and common in proteins." J Mol Biol **316**(1): 139-54.

Marti-Renom, M. A., U. Pieper, et al. (2007). "DBAli tools: mining the protein structure space." Nucleic Acids Res **35**(Web Server issue): W393-7.

Matsumura, I. and A. D. Ellington (2001). "In vitro evolution of beta-glucuronidase into a beta-galactosidase proceeds through non-specific intermediates." J Mol Biol **305**(2): 331-9.

McGovern, S. L. and B. K. Shoichet (2003). "Information decay in molecular docking screens against holo, apo, and modeled conformations of enzymes." J Med Chem **46**(14): 2895-907.

Meganathan, R. (2001). "Biosynthesis of menaquinone (vitamin K2) and ubiquinone (coenzyme Q): a perspective on enzymatic mechanisms." Vitam Horm **61**: 173-218.

Meganathan, R., R. Bentley, et al. (1981). "Identification of Bacillus subtilis men mutants which lack O-succinylbenzoyl-coenzyme A synthetase and dihydroxynaphthoate synthase." J Bacteriol **145**(1): 328-32.

Mildvan, A. S., Z. Xia, et al. (2005). "Structures and mechanisms of Nudix hydrolases." Arch Biochem Biophys **433**(1): 129-43.

Murzin, A. G., S. E. Brenner, et al. (1995). "SCOP: a structural classification of proteins database for the investigation of sequences and structures." J Mol Biol **247**(4): 536-40.

Nagano, N. (2005). "EzCatDB: the Enzyme Catalytic-mechanism Database." Nucleic Acids Res **33**(Database issue): D407-12.

Nagatani, R. A., A. Gonzalez, et al. (2007). "Stability for function trade-offs in the enolase superfamily "catalytic module"." Biochemistry **46**(23): 6688-95.

Nath, A. and W. M. Atkins (2008). "A quantitative index of substrate promiscuity." Biochemistry **47**(1): 157-66.

Neidhart, D. J., P. L. Howell, et al. (1991). "Mechanism of the reaction catalyzed by mandelate racemase. 2. Crystal structure of mandelate racemase at 2.5-A resolution: identification of the active site and possible catalytic residues." Biochemistry **30**(38): 9264-73.

Nobeli, I., R. V. Spriggs, et al. (2005). "A ligand-centric analysis of the diversity and evolution of protein-ligand relationships in E.coli." J Mol Biol **347**(2): 415-36.

Norvell, J. C. and J. M. Berg (2007). "Update on the protein structure initiative." Structure **15**(12): 1519-22.

O'Boyle, N. M., G. L. Holliday, et al. (2007). "Using reaction mechanism to measure enzyme similarity." J Mol Biol **368**(5): 1484-99.

O'Brien, P. J. and D. Herschlag (1999). "Catalytic promiscuity and the evolution of new enzymatic activities." Chem Biol **6**(4): R91-R105.

O'Loughlin, T. L., W. M. Patrick, et al. (2006). "Natural history as a predictor of protein evolvability." Protein Eng Des Sel **19**(10): 439-42.

Ojha, S., E. C. Meng, et al. (2007). "Evolution of Function in the "Two Dinucleotide Binding Domains" Flavoproteins." PLoS Comput Biol **3**(7): e121.

Overbeek, R., T. Disz, et al. (2004). "The SEED: a peer-to-peer environment for genome annotation." Communications of the ACM **47**(11): 46-51.

Palmer, D. R., J. B. Garrett, et al. (1999). "Unexpected divergence of enzyme function and sequence: "N-acylamino acid racemase" is o-succinylbenzoate synthase." Biochemistry **38**(14): 4252-8.

Pearson, W. R. and M. L. Sierk (2005). "The limits of protein sequence comparison?" Curr Opin Struct Biol **15**(3): 254-60.

Pegg, S. C. and P. C. Babbitt (1999). "Shotgun: getting more from sequence similarity searches." Bioinformatics **15**(9): 729-40.

Pegg, S. C., S. Brown, et al. (2005). "Representing structure-function relationships in mechanistically diverse enzyme superfamilies." Pac Symp Biocomput: 358-69.

Pegg, S. C., S. D. Brown, et al. (2006). "Leveraging enzyme structure-function relationships for functional inference and experimental design: the structure-function linkage database." Biochemistry **45**(8): 2545-55.

Pettersen, E. F., T. D. Goddard, et al. (2004). "UCSF Chimera--a visualization system for exploratory research and analysis." J Comput Chem **25**(13): 1605-12.

Pieper, U., N. Eswar, et al. (2006). "MODBASE: a database of annotated comparative protein structure models and associated resources." Nucleic Acids Res **34**(Database issue): D291-5.

Porter, C. T., G. J. Bartlett, et al. (2004). "The Catalytic Site Atlas: a resource of catalytic sites and residues identified in enzymes using structural data." Nucleic Acids Res **32**(Database issue): D129-33.

Riesenfeld, C. S., P. D. Schloss, et al. (2004). "Metagenomics: genomic analysis of microbial communities." Annu Rev Genet **38**: 525-52.

Rison, S. C., T. C. Hodgman, et al. (2000). "Comparison of functional annotation schemes for genomes." Funct Integr Genomics **1**(1): 56-69.

Ronquist, F. and J. P. Huelsenbeck (2003). "MrBayes 3: Bayesian phylogenetic inference under mixed models." Bioinformatics **19**(12): 1572-4.

Rost, B. (2002). "Enzyme function less conserved than anticipated." J Mol Biol **318**(2): 595-608.

Rowland, B., K. Hill, et al. (1995). "Structural organization of a Bacillus subtilis operon encoding menaquinone biosynthetic enzymes." Gene **167**(1-2): 105-9.

Sanchez, R. and A. Sali (1997). "Evaluation of comparative protein structure modeling by MODELLER-3." Proteins **Suppl 1**: 50-8.

Sauder, J. M., J. W. Arthur, et al. (2000). "Large-scale comparison of protein sequence alignment algorithms with structure alignments." Proteins **40**(1): 6-22.

Schmidt, D. M., B. K. Hubbard, et al. (2001). "Evolution of enzymatic activities in the enolase superfamily: functional assignment of unknown proteins in Bacillus

subtilis and Escherichia coli as L-Ala-D/L-Glu epimerases." Biochemistry **40**(51): 15707-15.

Schmidt, D. M., E. C. Mundorff, et al. (2003). "Evolutionary potential of (beta/alpha)8-barrels: functional promiscuity produced by single substitutions in the enolase superfamily." Biochemistry **42**(28): 8387-93.

Schmidt, S., S. Sunyaev, et al. (2003). "Metabolites: a helping hand for pathway evolution?" Trends Biochem Sci **28**(6): 336-41.

Schultes, E. A. and D. P. Bartel (2000). "One sequence, two ribozymes: implications for the emergence of new ribozyme folds." Science **289**(5478): 448-52.

Seffernick, J. L., M. L. de Souza, et al. (2001). "Melamine deaminase and atrazine chlorohydrolase: 98 percent identical but functionally different." J Bacteriol **183**(8): 2405-10.

Seibert, C. M. and F. M. Raushel (2005). "Structural and catalytic diversity within the amidohydrolase superfamily." Biochemistry **44**(17): 6383-91.

Shah, I. and L. Hunter (1997). "Predicting enzyme function from sequence: a systematic appraisal." Proc Int Conf Intell Syst Mol Biol **5**: 276-83.

Shannon, P., A. Markiel, et al. (2003). "Cytoscape: a software environment for integrated models of biomolecular interaction networks." Genome Res **13**(11): 2498-504.

Shindyalov, I. N. and P. E. Bourne (1998). "Protein structure alignment by incremental combinatorial extension (CE) of the optimal path." Protein Eng **11**(9): 739-47.

Song, L., C. Kalyanaraman, et al. (2007). "Prediction and assignment of function for a divergent N-succinyl amino acid racemase." Nat Chem Biol **3**(8): 486-91.

Steinbeck, C., Y. Han, et al. (2003). "The Chemistry Development Kit (CDK): an open-source Java library for Chemo- and Bioinformatics." J Chem Inf Comput Sci **43**(2): 493-500.

Stuart, A. C., V. A. Ilyin, et al. (2002). "LigBase: a database of families of aligned ligand binding sites in known protein sequences and structures." Bioinformatics **18**(1): 200-1.

Taber, H. W., E. A. Dellers, et al. (1981). "Menaquinone biosynthesis in Bacillus subtilis: isolation of men mutants and evidence for clustering of men genes." J Bacteriol **145**(1): 321-7.

Taylor, E. A., D. R. Palmer, et al. (2001). "The lesser "burden borne" by o-succinylbenzoate synthase: an "easy" reaction involving a carboxylate carbon acid." J Am Chem Soc **123**(24): 5824-5.

Taylor Ringia, E. A., J. B. Garrett, et al. (2004). "Evolution of enzymatic activity in the enolase superfamily: functional studies of the promiscuous o-succinylbenzoate synthase from Amycolatopsis." Biochemistry **43**(1): 224-9.

Thoden, J. B., E. A. Taylor Ringia, et al. (2004). "Evolution of enzymatic activity in the enolase superfamily: structural studies of the promiscuous o-succinylbenzoate synthase from Amycolatopsis." Biochemistry **43**(19): 5716-27.

Thompson, T. B., J. B. Garrett, et al. (2000). "Evolution of enzymatic activity in the enolase superfamily: structure of o-succinylbenzoate synthase from Escherichia coli in complex with Mg2+ and o-succinylbenzoate." Biochemistry **39**(35): 10662-76.

Tian, W. and J. Skolnick (2003). "How well is enzyme function conserved as a function of pairwise sequence identity?" J Mol Biol **333**(4): 863-82.

Tipton, K. and S. Boyce (2000). "History of the enzyme nomenclature system." Bioinformatics **16**(1): 34-40.

Todd, A. E., C. A. Orengo, et al. (1999). "Evolution of protein function, from a structural perspective." Curr Opin Chem Biol **3**(5): 548-56.

Todd, A. E., C. A. Orengo, et al. (2001). "Evolution of function in protein superfamilies, from a structural perspective." J Mol Biol **307**(4): 1113-43.

Venter, J. C., K. Remington, et al. (2004). "Environmental genome shotgun sequencing of the Sargasso Sea." Science **304**(5667): 66-74.

Vitkup, D., E. Melamud, et al. (2001). "Completeness in structural genomics." Nat Struct Biol **8**(6): 559-66.

Wedekind, J. E., R. R. Poyner, et al. (1994). "Chelation of serine 39 to Mg2+ latches a gate at the active site of enolase: structure of the bis(Mg2+) complex of yeast enolase and the intermediate analog phosphonoacetohydroxamate at 2.1-A resolution." Biochemistry **33**(31): 9333-42.

Weininger, D., A. Weininger, et al. (1989). "SMILES.2. Algorithm for generation of unique SMILES notation." Jour. Chem. Info. Comp. Sci. **29**: 97-101.

Weininger, D. J. (1988). "SMILES.1. Introduction and encoding rules." Jour. Chem. Inf. Comput. Sci. **28**: 31-46.

Wheeler, D. L., T. Barrett, et al. (2008). "Database resources of the National Center for Biotechnology Information." Nucleic Acids Res **36**(Database issue): D13-21.

Whelan, S. and N. Goldman (2001). "A general empirical model of protein evolution derived from multiple protein families using a maximum-likelihood approach." Mol Biol Evol **18**(5): 691-9.

Wilson, C. A., J. Kreychman, et al. (2000). "Assessing annotation transfer for genomics: quantifying the relations between protein sequence, structure and function through traditional and probabilistic scores." J Mol Biol **297**(1): 233-49.

Wu, C. H., R. Apweiler, et al. (2006). "The Universal Protein Resource (UniProt): an expanding universe of protein information." Nucleic Acids Res **34**(Database issue): D187-91.

Ycas, M. (1974). "On earlier states of the biochemical system." J Theor Biol **44**(1): 145-60.

Appendix A. Evolutionary Trace

A.1.1. Usage

This section will detail the usage of et.py on the command line to find class-conserved and class-specific residue positions in a multiple sequence alignment.

Brief command description

General usage: **et.py --alignment=<alignment file> --classes=<classes file> [--cccs=<cc/cs>] [--conservationType=<complete/subs/<integer>> [--outputFormat=<list/seqsel>**

Example command: **et.py --alignment=msa.fasta --classes=classes.txt -cccs=cs --conservationType=90 -outputFormat=seqsel**

Requirements for running et.py. A version of Python (http://www.python.org/) should be installed in order to run this script.

Options.

`--alignment=<alignment file>` : Specify the file containing the multiple sequence alignment in FASTA format.

`--classes=<classes file>` : Specify the file containing class memberships (See below for file format)

`--cccs=<cc/cs>` : Specify either 'cc' to find class-conserved residues or 'cs' to find class-specific residues

`--conservationType=<complete/subs/<integer>>` : Specify the type of conservation for the class-conserved or class-specific residues. Specify 'complete' to require 100% conservation, 'subs' to allow glutamate-aspartate and phenylalanine-tyrosine substitutions, and an integer to specify the level of conservation required.

`--outputFormat=<list/seqsel>` : Optional. Default is 'list' option. Specify the output format. Specify 'list' to output text listing, for each class, residue position and amino acid type of class-conserved or class-specific residues. Specify 'seqsel' to output the results as a seqsel formatted file. A description of this file can be found at http://www.cgl.ucsf.edu/chimera/1.1700/docs/ContributedSoftware/msfviewer/seqsel.html. Files of this type can be opened in the MultAlignViewer of UCSF's Chimera program (http://www.cgl.ucsf.edu/chimera/) to color the class-conserved or class-specific positions in the alignment.

Definitions.

class-conserved : Positions in the alignment that are conserved within every class, but not necessarily with the same amino acid type across different classes.

class-specific : Positions in the alignment that are conserved within a particular class and that may or may not have conservation within other classes.

Input file format – class file. Classes with fewer than 2 members are not used for the analysis. Sequences, which can be in multiple classes, must be listed using the same identifier as the identifiers in the alignment file. If the 'seqsel' option is selected, a color must be specified for each class in as three integers (0 - 255) in RGB format (i.e. "255 10 0"). When opening the seqsel file in MultAlignViewer in Chimera, this color will be used to highlight the class-conserved or class-specific residues for that class. The items in the 'CLASSLEVEL' and 'NEWCLASS' lines are separated by tabs.

```
CLASSLEVEL  <name of class level A>
NEWCLASS    <name of class A1>      [RGB color]
seq_id1
seq_id2
NEWCLASS    <name of class A2>      [255 0 255]
seq_id3
CLASSLEVEL  <name of class level B>
NEWCLASS    <name of class B1>      [0 0 255]
seq_id4
...
```

Output file format.

```
<name of class level A>
<name of class A1>
<residue position> <amino acid type>
<residue position> <amino acid type>
...
<name of class A2>
<residue position> <amino acid type>
<residue position> <amino acid type>
...
<name of class level B>
<name of class B1>
<residue position> <amino acid type>
<residue position> <amino acid type>
...
```

A.1.2. Script Code

```
1   # et.py <alignment fasta format> <classes list> <specific or conserved>
    <type of conservation> [output format]
2   # Ranyee Chiang
3   # created May 5, 2005
4   # last edited September 25, 2005
5   #
6   # reads in an alignment of fasta format
7   # reads in classes organized as tab delimited file
8   # gets class-specific residues of classes based on alignment
9
10
11  #################################################################
12  # FORMAT FOR WRITING CLASS LEVELS AND GROUPINGS
13  #
14  # sequences can be in multiple classes
15  # you can have classes with one or zero members, but they will not
16  # be used
17  #
18  # CLASSLEVEL   name of class level
19  # NEWCLASS     name of class at this class level
20  # seq_id1
21  # seq_id2
22  # NEWCLASS     name of second class at this class level
23  # seq_id3
24  # CLASSLEVEL   name of second class level
25  # NEWCLASS     name of class at second class level
26  # seq_id4
27  #################################################################
28
29  import sys, string, getopt
30
31  GAPLIST = ['-', '.', '~']
32
33  ########################### usage() ###############################
34  # prints usage
35  def usage():
36      print "Proper syntax:"
37      print
38      print "et.py --alignment=<fasta alignment> --classes=<classes file>"
39      print "       [--cccs=<cc/cs>]
        [--conservationType=<complete/subs/<integer>>]"
40      print "       [--outputFormat=<list/seqsel>]"
41      print
42      print "--------------------------------------------------------"
43      print "Options"
44      print
45      print "--cccs:  'cc' = class-conserved (default), 'cs' = class-specific"
46      print "--conservationType: 'complete' = 100% conserved, 'subs' = D/E
        F/Y, <integer> = amt of conservation"
47      print "--outputFormat: 'list' = list (default), 'seqsel' = seqsel"
48      print
49      print "--------------------------------------------------------"
50      print "Input file formats"
```

```
51      print
52      print "Alignment file must be fasta format alignment"
53      print "See script text for class file description"
54
55
56  ######################### printError() ###########################
57  # prints an error if the syntax is incorrect and exits the program
58  def printError(s):
59      print "-------------------------------------------------------"
60      print "Error:", s
61      print "-------------------------------------------------------"
62      print
63      usage()
64      sys.exit()
65
66  CLASSSPECIFICTYPE = "cs"
67  CLASSCONSERVEDTYPE = "cc"
68
69  COMPLETECONSERVATIONTYPE = "complete"
70  SUBSTITUTIONSTYPE = "subs"
71  PARTIALCONSERVATIONTYPE = "partial"
72
73  LISTTYPE = "list"
74  SEQSELTYPE = "seqsel"
75
76  global etType
77  etType = CLASSCONSERVEDTYPE
78
79  global conservationType
80  conservationType = COMPLETECONSERVATIONTYPE
81  global conservationLevel
82  conservationLevel = 100
83
84  global alignmentFile
85  alignmentFile = ""
86  global classesFile
87  classesFile = ""
88
89  global outputType
90  outputType = LISTTYPE
91
92  ######################## handleArgs() #############################
93  # uses getopt to handle the command line arguments
94  # global variables are set here
95  def handleArgs():
96
97      global etType
98      global conservationType
99      global conservationLevel
100     global alignmentFile
101     global classesFile
102     global outputType
103
```

```
104     try:
105         opts, args = getopt.getopt(sys.argv[1:], "h", ["help", "alignment=",
        "classes=", "cccs=", "conservationType=", "outputFormat="])
106     except getopt.GetoptError:
107         printError("Incorrect parameter used")
108         sys.exit(2)
109     for o, a in opts:
110         if o in ["-h", "--help"]:
111             usage()
112             sys.exit()
113         elif o == "--cccs":  # class-conserved or class-specific
114             etType = a
115             if etType == CLASSSPECIFICTYPE or etType == CLASSCONSERVEDTYPE:
116                 pass
117             else:
118                 printError("Invalid option for cccs: only valid options are
                cs and cc")
119         elif o == "--alignment":
120             alignmentFile = a
121         elif o == "--classes":
122             classesFile = a
123         elif o == "--conservationType":
124             if a == SUBSTITUTIONSTYPE or a == COMPLETECONSERVATIONTYPE:
125                 conservationType = a
126             else:
127                 try:
128                     conservationLevel = int(a)
129                     conservationType = PARTIALCONSERVATIONTYPE
130                 except ValueError:
131                     printError("Invalid conservationType: only valid options
                    are 'complete', 'subs', <integer>")
132                 if conservationLevel > 100:
133                     printError("conservationType cannot be an integer over
                    100")
134         elif o == "--outputFormat":
135             if a == LISTTYPE or a == SEQSELTYPE:
136                 outputType = a
137             else:
138                 printError("Invalid output format: only valid options are
                'list' and 'seqsel'")
139
140 if __name__ == "__main__":
141     handleArgs()
142
143 ##################################################################
144 # parse class file
145 try:
146     lines = open(classesFile, 'r').readlines()
147 except IOError:
148     printError("Cannot open/read class file: %s" % (classesFile))
149
150 # classes are stored as a list (levels) of lists (classes) of lists
    (sequences)
```

```
151 CLASSLEVEL = "CLASSLEVEL"
152 NEWCLASS = "NEWCLASS"
153 classes = []      ## list of list of lists
154 levelNames = []   ## contains a list of level names
155 classNames = []   ## contains a list of class names for each level
156
157 currentLevelName = ""
158 currentClassNames = []
159 if outputType == SEQSELTYPE:
160     colors = []   ## list of colors of classes for each level
161     currentClassColors = []
162 currentLevel = []  # this is a list of classes
163 currentClass = []  # this is a list of sequences
164 firstLevel = 1     # this is to make sure that you don't add the first empty
    class
165 firstClass = 1     # this is to make sure that you don't add the first empty
    list of sequences
166 for l in lines:
167     if l[0] != "#":
168         tokens = (l.strip()).split('\t')
169         if tokens[0] == CLASSLEVEL:
170             if firstLevel:
171                 firstLevel = 0
172             else:
173                 currentLevel.append(currentClass)
174                 classes.append(currentLevel)
175                 classNames.append(currentClassNames)
176                 if outputType == SEQSELTYPE:
177                     colors.append(currentClassColors)
178             levelNames.append(tokens[1])
179             firstClass = 1
180             currentLevel = []
181             currentClassNames = []
182             if outputType == SEQSELTYPE:
183                 currentClassColors = []
184         elif tokens[0] == NEWCLASS:
185             if firstClass:
186                 firstClass = 0
187             else:
188                 currentLevel.append(currentClass)
189             currentClassNames.append(tokens[1])
190             if outputType == SEQSELTYPE:
191                 currentClassColors.append(tokens[2].split())
192             currentClass = []
193         else:
194             currentClass.append(tokens[0])
195 currentLevel.append(currentClass)
196 classes.append(currentLevel)
197 classNames.append(currentClassNames)
198 if outputType == SEQSELTYPE:
199     colors.append(currentClassColors)
200
201 ######################################################################
```

```
202 # parse alignment file
203 # this is for fasta format right now
204 try:
205     lines = open(alignmentFile, 'r').readlines()
206 except IOError:
207     printError("Cannot open/read alignment file: %s" % (alignmentFile))
208
209 sequences = {}
210 sequenceIndices = {}
211 currentSeqId = ""
212 currentSeq = ""
213 firstSeq = 1
214 count = 0
215 for l in lines:
216     if l[0] == '>':
217         if firstSeq:
218             firstSeq = 0
219         else:
220             sequences[currentSeqId] = currentSeq
221             sequenceIndices[currentSeqId] = count
222             count += 1
223         currentSeq = ""
224         tokens = (l.strip()).split('>')
225         currentSeqId = tokens[1]
226         #print currentSeqId
227     else:
228         currentSeq += l.strip()
229 sequences[currentSeqId] = currentSeq
230 sequenceIndices[currentSeqId] = count
231
232
233
234
235
236 ALLOWABLESUBSTITUTIONS = [['D','E'], ['F','Y']]
237
238 ############### areEquivalent(char1, char2) #######################
239 #
240 # function to determine whether two characters are allowable
241 # substitutions
242 #
243 # param char1       first character
244 # param char2       second character
245 # return 1          if two characters are equivalent, 0 otherwise
246 def areEquivalent(char1, char2):
247     if char1 == char2:
248         return 1
249     else:
250         for l in ALLOWABLESUBSTITUTIONS:
251             if (char1 in l) and (char2 in l):
252                 return 1
253     return 0
254
```

```
255 ######### isClassConserved(classSequences, position) ##############
256 #
257 # function to check if all positions within sequences have the same
258 # residue
259 # if all the sequences have a gap at a position, that doesn't count
260 #
261 # param classSequences   list of sequences in class
262 # param position         residue position to be checked
263 # return char identity of conserved aa at that position
264 # return 0 if it is not conserved
265 def isClassConserved(classSequences, position):
266     if conservationType != SUBSTITUTIONSTYPE:
267         conservedAADict = {}
268         for s in classSequences:
269             c = s[position]
270             if not c in conservedAADict.keys():
271                 conservedAADict[c] = 1
272             else:
273                 conservedAADict[c] += 1
274         conservedAA = ''
275         maxLength = 0
276         for c in conservedAADict.keys():
277             if conservedAADict[c] > maxLength:
278                 conservedAA = c
279                 maxLength = conservedAADict[c]
280         if conservedAA in GAPLIST:
281             return 0
282         if (maxLength+0.0)/len(classSequences) >=
        (conservationLevel+0.0)/100:
283             return conservedAA
284         else:
285             return 0
286     else:
287         conservedAA = classSequences[0][position]
288         if conservedAA in GAPLIST:
289             return 0
290         for s in classSequences[1:]:
291             if not areEquivalent(s[position], conservedAA):
292                 return 0
293         return conservedAA
294
295
296 ###### isSpecific(nonClassSequences, position, conservedAA) #######
297 #
298 # function to determine if all nonClassSequences at position position
299 # do not have conservedAA at that position
300 #
301 # param nonClassSequences    list of sequence identifiers that aren't in
    class
302 # param position             int position to check
303 # param conservedAA          char residue that should not be at the position
    in the sequence
304 # return 1 if all sequences do not have conservedAA at position
```

```
305 # return 0 otherwise
306 def isSpecific(nonClassSequences, position, conservedAA):
307     for s in nonClassSequences:
308         if s[position] == conservedAA:
309             return 0
310     return 1
311
312 ######### getClassSpecificResidues(classList, sequences) ##########
313 #
314 # function to find class-specific residues given class members
315 # full alignment
316 # also checks to see that len(classList) > 0
317 #
318 # param classList    list of sequences identifiers in class
319 # param sequences    dict of all sequences (key is sequence id, value is
    sequence)
320 # return list of class-specific positions
321 # return empty list if len(classList) > 0
322 def getClassSpecificResidues(classList, sequences):
323     if len(classList) > 1:
324         length = len(sequences[sequences.keys()[0]])
325         # gather all the sequences into one list
326         classSequences = []
327         for c in classList:
328             try:
329                 classSequences.append(sequences[c])
330             except KeyError:
331                 pass
332         if len(classSequences) < 3:
333             return []
334          # gather all nonclass sequences into one list
335         nonClassSequences = []
336         for s in sequences.keys():
337             if not s in classList:
338                 nonClassSequences.append(sequences[s])
339          # loop through all the positions
340         specificPositionsList = []
341         for p in range(length):
342             # check to see if all sequences in class have the same residue
343             conservedAA = isClassConserved(classSequences, p)
344             if conservedAA:
345                 if etType == CLASSSPECIFICTYPE:
346                     # check to see if all sequences not in class have a
                    different residu
347                     if isSpecific(nonClassSequences, p, conservedAA):
348                         specificPositionsList.append([p, conservedAA])
349                 else:
350                     specificPositionsList.append([p, conservedAA])
351         return specificPositionsList
352     else:
353         return []
354
355 ######################################################################
```

```
356 # go through classes and find class-specific residues
357 if outputType == LISTTYPE:
358     for l in range(len(levelNames)):
359         print levelNames[l]
360         for c in range(len(classNames[l])):
361             print " ", classNames[l][c]
362             residues = getClassSpecificResidues(classes[l][c], sequences)
363             print "    ",
364             for r in residues:
365                 print "%s%d" % (r[1], r[0]),
366             print
367 else:
368     # in seqsel format
369     # positions are numbered starting at 0
370     # sequences are numbered starting at 1
371     for l in range(len(levelNames)):
372         for c in range(len(classNames[l])):
373             residues = getClassSpecificResidues(classes[l][c], sequences)
374             # get sequence ranges
375             seqIndexList = []
376             min = -1
377             for seq in classes[l][c]:
378                 try:
379                     seqIndexList.append(sequenceIndices[seq])
380                 except KeyError:
381                     pass
382             seqIndexList.sort()
383             rangeList = []
384             if len(seqIndexList) > 2:
385                 min = seqIndexList[0]
386                 for s in range(len(seqIndexList))[1:]:
387                     if seqIndexList[s] != seqIndexList[s-1]+1:
388                         rangeList.append("%s %s" % (min+1,
                            seqIndexList[s-1]+1))
389                         min = seqIndexList[s]
390                 rangeList.append("%s %s" % (min+1,
                    seqIndexList[len(seqIndexList)-1]+1))
391                 for seqRange in rangeList:
392                     for r  in residues:
393                         print r[0], r[0], seqRange,
394                         for col in colors[l][c]:
395                             print col,
396                         print
397
398 ####################################################################
399 # output this information into file
400 # this file can later be converted into seqsel format to be visualized in
    Chimera
```

Appendix B. Reacting and Common Substructures

B.1.1. Usage

This section will detail the usage of the Java program RCSubstructures to calculate reacting and conserved structures for a single superfamily on the command line.

Command description

General usage: `java RCSubstructures <input molfile specification file> <output smiles file> > <output summary file>`

Example command: `java RCSubstructures a.102.1.mol.txt a.102.1.smiles.txt > a.102.1.sssummary.txt`

Requirements for running RCSubstructures. A version of Java's JDK (http://java.sun.com/javase/) should be installed in order to compile and run this program. In addition, the Chemistry Development Kit (http://sourceforge.net/projects/cdk) should be installed.

Requirements for input files. To be useful, multiple reactions should be specified for each superfamily. Reactions must be unimolecular (only one substrate), but multiple products can be specified. Molfiles for all substrates and products must be specified.

Notes about output files. If there are inconsistencies in the substrates and products of the specified reactions (i.e. the number of atoms in the substrate doesn't approximately equal the number of atoms in the products or the number of atoms in the substrate is very small), those reactions will not be used in the calculation. The output files only contain results for the valid reactions. To identify the reactions in the input file to which the outputted reactions correspond, the index of the reaction in the input file is specified (See below).

Input file format – Molfile specification file. This file specifies the location of the coordinate files for the substrate and products of the reactions in the superfamily. Each line in this file corresponds to one reaction.

```
	<molfile for substrate A> = <molfile for product A>
	<molfile for substrate B> = <molfile for product B1> + <molfile for product B2>
	...
```

Output file format – SMILES file. After the program is done running, this file will contain the SMILES string for the conserved substrate substructure for the superfamily. And for each reaction, this file will contain the SMILES strings for the substrate, product, reacting and nonreacting substructures, conserved and unconserved substructures, and the overlaps between all combinations of these substructures. The order of the reactions is the same as for the other output of RCSubstructures.

```
	<conserved substructure>
	CCCCCCCCCCCCCCC
	<substrate A>
	<product A>
	<reacting substructure A>
	<nonreacting substructure A>
	<conserved substructure A>
	<unconserved substructure A>
	<reacting conserved substructure overlap A>
	<reacting unconserved substructure overlap A>
	<nonreacting conserved substructure overlap A>
	<nonreacting unconserved substructure overlap A>
	CCCCCCCCCCCCCCC
	<substrate B>
	<product B>
	<reacting substructure B>
	<nonreacting substructure B>
	<conserved substructure B>
	<unconserved substructure B>
	<reacting conserved substructure overlap B>
	<reacting unconserved substructure overlap B>
	<nonreacting conserved substructure overlap B>
	<nonreacting unconserved substructure overlap B>
	...
```

Output file format – Summary file. After the program is done running, this file will contain the number of atoms and bonds in the various substructures that have been calculated. For each reaction, this file will contain the name of the substrate and product as specified in the molfiles. In addition, there will be an index that corresponds to the order of the reaction in the input file. The order of the reactions in this ouput is the same as the SMILES string output.

```
<substrate A name> = <product A name> <reaction A index>
substrate                     <# of atoms> <# of bonds>
product                       <# of atoms> <# of bonds>
reacting                      <# of atoms> <# of bonds>
nonreacting                   <# of atoms> <# of bonds>
conserved                     <# of atoms> <# of bonds>
unconserved                   <# of atoms> <# of bonds>
reacting+conserved            <# of atoms> <# of bonds>
reacting+unconserved          <# of atoms> <# of bonds>
nonreacting+conserved         <# of atoms> <# of bonds>
nonreacting+unconserved       <# of atoms> <# of bonds>
<substrate B name> = <product B name> <reaction B index>
substrate                     <# of atoms> <# of bonds>
product                       <# of atoms> <# of bonds>
reacting                      <# of atoms> <# of bonds>
nonreacting                   <# of atoms> <# of bonds>
conserved                     <# of atoms> <# of bonds>
unconserved                   <# of atoms> <# of bonds>
reacting+conserved            <# of atoms> <# of bonds>
reacting+unconserved          <# of atoms> <# of bonds>
nonreacting+conserved         <# of atoms> <# of bonds>
nonreacting+unconserved       <# of atoms> <# of bonds>
...
```

B.1.2. Program Code

```
1   /** RCSubstructures.java
2    *
3    * Usage: java RCSubstructures <input *.mol file> <output smiles file> >
    <output summary file>
4    *
5    * Ranyee Chiang
6    * December 15, 2006
7    * last updated January 9, 2007
8    *
9    * Reads in substrates and reactions for a superfamily
10   * Finds conserved substructure, reacting substructure
11   * Finds overlap
12   */
13
14  // For each substrate, get conserved and unconserved region by always
    putting in molecule in question or resulting substructure 1st into getMCS
15  // For each substrate with product info, get reacting and non-reacting
    regions - put substrate in first to MCS
16  // Then get MCS between conserved and reacting, etc but also check atom
    coordinates
17
18  //package RCSubstructures;
19
20  import java.lang.String;
21  import java.io.*;
22  import java.util.*;
23  import java.lang.*;
24  import java.lang.Math;
25  import org.openscience.cdk.Molecule;
26  import org.openscience.cdk.Atom;
27  import org.openscience.cdk.Bond;
28  import org.openscience.cdk.ChemModel;
29  import org.openscience.cdk.io.Mol2Reader;
30  import org.openscience.cdk.io.MDLReader;
31  import org.openscience.cdk.io.SMILESWriter;
32  import org.openscience.cdk.exception.CDKException;
33  import org.openscience.cdk.isomorphism.UniversalIsomorphismTester;
34  import org.openscience.cdk.isomorphism.mcss.RMap;
35  import org.openscience.cdk.graph.PathTools;
36  import org.openscience.cdk.AtomContainer;
37  import javax.vecmath.Point3d;
38  import javax.vecmath.Point2d;
39  //import RCsubstructures.RCAtomContainer;
40
41  public class RCSubstructures {
42
43      private static int MINATOMCOUNT = 3;
44      private static SMILESWriter sWriter;
45      private static AtomContainer blankMolecule = new AtomContainer();
46      private static UniversalIsomorphismTester uit = new
        UniversalIsomorphismTester();
47      private static int overlapCount = 0;
48      private static ArrayList acStack = new ArrayList();
```

```
49        private static ArrayList rmapStack = new ArrayList();
50
51        /* -----------------------------------------------------------
52         *       methods to read and initialize molecules from files
53         * -----------------------------------------------------------
54         */
55
56        /**
57         *           ----  getReactionsList ----
58         *
59         * param String filename - name of file containing paths of mol2 files
60         * return ArrayList - list of strings
61         *
62         * Reads and parses file with filename and returns list
63         * of mol2 file names
64         */
65        private static ArrayList getReactionsList(String filename) throws
          IOException {
66      FileReader fr = null;
67      try {fr = new FileReader(new File(filename));}
68      catch (FileNotFoundException e) {
69          System.out.println("file not found");
70          System.exit(1);
71      }
72      ArrayList filenameList = new ArrayList();
73      Reaction reaction = new Reaction(0);
74      int rxnCounter = 0;
75      String molName = new String();
76      int c;
77      boolean hasProduct = false;
78      while ((c = fr.read()) != -1) {
79          if (c == '-') {
80        hasProduct = true;
81        reaction = new Reaction(rxnCounter, molName);
82        rxnCounter++;
83        molName = new String();
84          } else if (c == '+') {
85        if (!hasProduct) {  // this case shouldn't happen but if it does
86            System.out.println("error with reading reactions with multiple
              products");
87        } else {
88            reaction.addProductFile(molName);
89        }
90        molName = new String();
91          } else if (c == '\n') {
92        if (hasProduct) {
93            reaction.addProductFile(molName);
94        } else {
95            reaction = new Reaction(rxnCounter, molName);
96            rxnCounter++;
97        }
98        filenameList.add(reaction);
99        molName = new String();
```

```
100       hasProduct = false;
101         } else if (c != '\n') {
102       molName = molName + (char)c;
103         }
104     }
105     // take care of situation where last line is /n
106     if (molName.length() > 1) {
107         if (hasProduct) {
108       reaction.addProductFile(molName);
109         } else {
110       reaction = new Reaction(rxnCounter, molName);
111       rxnCounter++;
112         }
113         filenameList.add(reaction);
114     }
115     return filenameList;
116       }
117
118       /**          --- getECNumList ---
119        * reads *.info.txt file to get EC numbers as a list
120        */
121       private static ArrayList getECNumList(String filename) throws
          IOException {
122
123     FileReader fr = null;
124     try {fr = new FileReader(new File(filename));}
125     catch (FileNotFoundException e) {
126         System.out.println("file not found");
127         System.exit(1);
128     }
129     ArrayList ECList = new ArrayList();
130     int rxnCounter = 0;
131     String ECNum = new String();
132     int c;
133     boolean pauseReading = false;
134     while ((c = fr.read()) != -1) {
135         if (c == ' ' && pauseReading == false) {
136       ECList.add(ECNum);
137       rxnCounter++;
138       pauseReading = true;
139       ECNum = new String();
140         } else if (c == '\n') {
141       pauseReading = false;
142         } else if (pauseReading == false) {
143       ECNum = ECNum + (char)c;
144         }
145     }
146     return ECList;
147
148       }
149
150
151       /**
```

```
152      *          ---- getMolecule ----
153      *
154      * param String filename - name of mol2 file
155      * return Molecule - converted from contents of mol2 file
156      *
157      * Reads in file with filename, creates and returns new Molecule object
158      */
159     private static Molecule getMolecule(String filename) {
160   FileReader file = null;
161   try {file = new FileReader(filename);}
162   catch (FileNotFoundException e) {
163       return new Molecule();
164   }
165   MDLReader testReader = new MDLReader(file);
166   Molecule mol1 = new Molecule();
167   ChemModel chemMod = new ChemModel();
168   try {
169       chemMod = (ChemModel)testReader.read(chemMod);
170       mol1 = chemMod.getSetOfMolecules().getMolecule(0);
171   }
172   catch (CDKException e) {
173       System.out.println("reader error: " + e.getMessage() + " " +
        filename);
174   }
175   return mol1;
176     }
177
178     /**
179      *           ---- removeHydrogensFromMolecule ----
180      *
181      * param AtomContainer ac
182      * returns AtomContainer
183      */
184     private static AtomContainer removeHydrogensFromMolecule(AtomContainer
        ac) {
185   Atom[] atomList = ac.getAtoms();
186   String HString = new String("H");
187
188   for (int i=0; i< atomList.length; i++) {
189       if (HString.equals(atomList[i].getSymbol())) {
190     ac.removeAtomAndConnectedElectronContainers(atomList[i]);
191       }
192   }
193   return ac;
194     }
195
196     /**       -------- changeSToP(AtomContainer ac)
197      *
198      * changes all S to P so that I find matches between S and P
199      */
200     private static AtomContainer changeSToP(AtomContainer ac) {
201   Atom[] atomList = ac.getAtoms();
202   String SString = new String("S");
```

```
203    String PString = new String("P");
204    char PChar = 'P';
205
206    for (int i = 0; i < atomList.length; i++) {
207        if (SString.equals(atomList[i].getSymbol())) {
208      String symbolString = atomList[i].getSymbol();
209      atomList[i].setSymbol(symbolString.replace('S', PChar));
210      atomList[i].setAtomicNumber(15);
211      atomList[i].setMassNumber(31);
212      atomList[i].setExactMass(30.9737634);
213        }
214    }
215    return ac;
216      }
217
218      /**
219       *            ---- getMoleculeName ----
220       *
221       * param String filename
222       * return String molecule name
223       *
224       * from file with filename, gets molecule name (second line of Brenda
       molfile)
225       */
226      private static String getMoleculeName(String filename) throws
      IOException {
227    FileReader fr = null;
228    try {fr = new FileReader(new File(filename));}
229    catch (FileNotFoundException e) {
230        return null;
231    }
232    ArrayList filenameList = new ArrayList();
233    String molName = new String();
234    int c;
235    int count = 0;
236    while ((c = fr.read()) != -1) {
237        if (c != '\n') {
238      molName = molName + (char)c;
239        }
240        else {
241      if (count == 1) {
242          return molName;
243      }
244      count = count + 1;
245      molName = new String();
246        }
247    }
248    return molName;
249      }
250
251      /**
252       *        ---- getMoleculesForList ----
253       *
```

```
254      * param ArrayList filenameList - list of mol2 file names
255      * return ArrayList - ArrayList of Molcules
256      *
257      * Reads in all mol2 files in filename List and returns ArrayList
258      * of Molecules
259      */
260     private static ArrayList getMoleculesForList(ArrayList reactionList)
        throws IOException, CDKException {
261     ArrayList molList = new ArrayList();
262     RCAtomContainer mol;
263     String molTitle;
264     AtomContainer ac;
265     Reaction reaction;
266     for (int i = 0; i < reactionList.size(); i++) {
267         ac = new AtomContainer();
268         molTitle = new String("");
269         reaction = (Reaction)reactionList.get(i);
270         // convert Reaction's substrate
271         String filename = reaction.getSubstrateFile();
272         filename = filename.trim();
273         // converts filename to AtomContainer, removes hydrogens, adds to
            RCAtomContainer
274         ac = getMolecule(filename);
275
276         if (ac != null) {
277       molTitle = getMoleculeName(filename);
278       ac = removeHydrogensFromMolecule(ac);
279       ac = changeSToP(ac);
280       if (molTitle != null) {
281           mol = new RCAtomContainer(ac, molTitle);
282           if (mol.getLigand().getAtomCount() > MINATOMCOUNT) {
283         reaction.setSubstrateAtomContainer(mol);
284           }
285       }
286         }
287
288         // convert Reaction's product
289         ArrayList filenameList = reaction.getProductFileList();
290         for (int p = 0; p < filenameList.size(); p++) {
291       filename = (String)filenameList.get(p);
292       if (filename != null) {
293           filename = filename.trim();
294           ac = getMolecule(filename);
295           ac = removeHydrogensFromMolecule(ac);
296           ac = changeSToP(ac);
297           if (ac != null) {
298         molTitle = getMoleculeName(filename);
299         mol = new RCAtomContainer(ac, molTitle);
300         if (mol == null) {
301         } else {
302             // accept molecules that are smaller than minatomcount
303             reaction.addProductAtomContainer(mol);
304         }
```

```
305             }
306         }
307           }
308     }
309     for (int i = reactionList.size()-1; i >= 0; i--) {
310         RCAtomContainer testSubstrate =
            ((Reaction)reactionList.get(i)).getSubstrateAtomContainer();
311         if (testSubstrate == null) {
312       // remove from list
313       reactionList.remove(i);
314           }
315     }
316     return reactionList;
317       }
318
319       /**
320        *       ---- initFileWriter ----
321        *
322        * param String filename - name of file to initialize for writing
323        * return FileWriter - Object which can be used to write to the file
324        *
325        * Initializes FileWriter for writing to file with filename
326        */
327       private static FileWriter initFileWriter(String filename) throws
          IOException, CDKException{
328     FileWriter fileWriter = null;
329     try {fileWriter = new FileWriter(filename);}
330     catch (FileNotFoundException e) {
331         System.out.println("File not found: " + filename);
332         System.exit(1);
333     }
334     return fileWriter;
335       }
336
337       /**
338        *            ----  readStereochemistryForList ----
339        *
340        * param ArrayList list - list of reactions
341        * return none
342        *
343        * Reads and parses information for stereochemistry and adds it to
344        * substrate and product in reactions
345        */
346       private static void readStereochemistryForList(ArrayList list) throws
          IOException {
347     Reaction reaction;
348     String scFilename;
349     ArrayList productFilenameList;
350     boolean fileIsPresent;
351     int c; float x = 0; float y = 0; float z = 0; int sc = 0; int
        coordinateCount = 0;
352     ArrayList currentCharArray = new ArrayList();
353     Object[] objArray;
```

```
354     char[] charArray;
355
356     for (int r = 0; r < list.size(); r++) {
357         reaction = (Reaction)list.get(r);
358         FileReader fr = null;
359         scFilename = reaction.getSubstrateFile().replace(".mol", "sc.txt");
360         scFilename =
            scFilename.replace("C:\\brenda_molfiles\\molfiles_out\\",
            "C:\\brenda_molfiles\\stereochemistry\\");
361
362         try {fr = new FileReader(new File(scFilename)); fileIsPresent =
            true;}
363         catch (FileNotFoundException e) {
364       fileIsPresent = false;
365         }
366
367         if (fileIsPresent) {
368       // read the file and set the RCAtomContainers to have the correct
          stereopariy
369       x = 0; y = 0; z = 0; sc = 0; coordinateCount = 0;
370 =     currentCharArray = new ArrayList();
371       while ((c = fr.read()) != -1) {
372           if (c == ' ') {   // are now getting ready to encounter new number,
              deal with previous number
373         objArray = currentCharArray.toArray();
374         charArray = new char[objArray.length];
375         for (int i = 0; i < charArray.length; i++) {
376             charArray[i] = ((Character)objArray[i]).charValue();
377         }
378         if (coordinateCount == 2) {z = new Float(new String(charArray));
            coordinateCount++;}
379         else if (coordinateCount == 1) {y = new Float(new String(charArray));
            coordinateCount++;}
380         else if (coordinateCount == 0) {x = new Float(new String(charArray));
            coordinateCount++;}
381         currentCharArray = new ArrayList();
382           } else if (c == '\n') {  // you've reached end of line
383         objArray = currentCharArray.toArray();
384         charArray = new char[objArray.length];
385         for (int i = 0; i < charArray.length; i++) {
386             charArray[i] = ((Character)objArray[i]).charValue();
387         }
388         if (charArray[0] == '1') {
389             sc = 1;
390         } else if (charArray[0] == '2') {
391             sc = 2;
392         }
393         reaction.getSubstrateAtomContainer().setStereoparity(sc, x, y, z);
394         coordinateCount = 0;
395         currentCharArray = new ArrayList();
396         sc = 0;
397           } else {
398         currentCharArray.add((char)c);
```

```
399             }
400         }
401         // read in product stereoparity information
402         productFilenameList = reaction.getProductFileList();
403         for (int p = 0; p < productFilenameList.size(); p++) {
404             scFilename = ((String)productFilenameList.get(p)).replace(".mol",
                "sc.txt");
405             scFilename =
                scFilename.replace("C:\\brenda_molfiles\\molfiles_out\\",
                "C:\\brenda_molfiles\\stereochemistry\\");
406             scFilename = scFilename.trim();
407
408             try {fr = new FileReader(new File(scFilename)); fileIsPresent =
                true;}
409             catch (FileNotFoundException e) {
410           fileIsPresent = false;
411             }
412
413             if (fileIsPresent) {
414           // read the file and set the RCAtomContainers to have the correct
              stereopariy
415           x = 0; y = 0; z = 0; sc = 0; coordinateCount = 0;
416           currentCharArray = new ArrayList();
417           while ((c = fr.read()) != -1) {
418               if (c == ' ') {  // are now getting ready to encounter new
                  number, deal with previous number
419             objArray = currentCharArray.toArray();
420             charArray = new char[objArray.length];
421             for (int i = 0; i < charArray.length; i++) {
422                 charArray[i] = ((Character)objArray[i]).charValue();
423             }
424             if (coordinateCount == 2) {z = new Float(new String(charArray));
                coordinateCount++;}
425             else if (coordinateCount == 1) {y = new Float(new
                String(charArray)); coordinateCount++;}
426             else if (coordinateCount == 0) {x = new Float(new
                String(charArray)); coordinateCount++;}
427             currentCharArray = new ArrayList();
428               } else if (c == '\n') {  // you've reached end of line
429             objArray = currentCharArray.toArray();
430             charArray = new char[objArray.length];
431             for (int i = 0; i < charArray.length; i++) {
432                 charArray[i] = ((Character)objArray[i]).charValue();
433             }
434             if (charArray[0] == '1') {
435                 sc = 1;
436
437             } else if (charArray[0] == '2') {
438                 sc = 2;
439             }
440             reaction.getProductAtomContainer().setStereoparity(sc, x, y, z);
441             coordinateCount = 0;
442             currentCharArray = new ArrayList();
```

```
443            } else {
444          currentCharArray.add((char)c);
445            }
446        }
447
448          }
449      }
450        }
451    }
452      }
453
454      /* ------------------------------------------------------------------
455       *          methods for debugging
456       * ------------------------------------------------------------------
457       */
458
459      /**
460       *           ---- printReactionsList ----
461       *
462       * debugging method
463       */
464      private static void printReactionsList(ArrayList list) throws
         IOException, CDKException{
465    Reaction reaction;
466    String pFile;
467    RCAtomContainer pAc;
468    for (int i = 0; i < list.size(); i++) {
469        reaction = (Reaction)list.get(i);
470        printSMILES(reaction.getSubstrateAtomContainer().getLigand());
471        pAc = reaction.getProductAtomContainer();
472        if (pAc == null) {
473      System.out.println("No product");
474        } else {
475      printSMILES(pAc.getLigand());
476        }
477    }
478      }
479
480      /**
481       *            ---- printSMILES ----
482       *
483       * param AtomContainer ac
484       * returns none
485       */
486      private static void printRCSMILES(RCAtomContainer ac)  throws
         CDKException, IOException{
487    sWriter.write(new Molecule(ac.getLigand()));
488      }
489
490      private static void printSMILES(AtomContainer ac)  throws CDKException,
         IOException{
491    if (ac != null) {
492        sWriter.write(new Molecule(ac));
```

```
493     }
494       }
495
496       private static void printSMILESList(ArrayList list) throws
          CDKException, IOException {
497     AtomContainer ac;
498     for (int i = 0; i < list.size(); i++) {
499         ac = (AtomContainer)list.get(i);
500         printSMILES(ac);
501     }
502       }
503
504       private static void printSMILESList(List list) throws CDKException,
          IOException {
505     RCAtomContainer ac;
506     for (int i = 0; i < list.size(); i++) {
507         ac = (RCAtomContainer)list.get(i);
508         printSMILES(ac.getLigand());
509     }
510       }
511
512
513       private static void printCoordinates(AtomContainer ac) {
514     Atom[] atoms = ac.getAtoms();
515     for (int j = 0; j < atoms.length; j++) {
516         Atom a = atoms[j];
517         if ((a.getX2d() == 0) && (a.getY2d() == 0)) {
518       System.out.print(a.getPoint3d());
519         } else {
520       System.out.print(a.getPoint2d());
521         }
522     }
523     System.out.println();
524       }
525
526       private static void printStereoparity(AtomContainer ac) {
527     Atom[] atoms = ac.getAtoms();
528     for (int j = 0; j < atoms.length; j++) {
529         Atom a = atoms[j];
530         System.out.print(a.getStereoParity());
531     }
532     System.out.println();
533       }
534
535       /* ------------------------------------------------------------------
536        *        private housekeeping  and utility methods
537        * ------------------------------------------------------------------
538        */
539
540       /**
541        *         ---- convertMoleculeList ----
542        *
543        * converts ArrayList of Reactions to ArrayList of AtomContainers
```

```
544       * returns ArrayList of the substrates
545       */
546      private static ArrayList convertMoleculeList(ArrayList molList) throws
         CDKException, IOException{
547     ArrayList newList = new ArrayList();
548     for (int i = 0; i < molList.size(); i++) {
549         AtomContainer ac =
            ((Reaction)molList.get(i)).getSubstrateAtomContainer().getLigand();
550         newList.add(ac);
551     }
552     return newList;
553      }
554
555      /**
556       *            ---- swap ----
557       *
558       * param ArrayList list - list with elements to be swapped
559       * param int i - index of first element to be swapped
560       * param int j - index of second element to be swapped
561       * return ArrayList - ArrayList with elements swapped
562       *
563       * switches the values of the ith and jth position in the ArrayList
564       */
565      private static ArrayList swap(ArrayList list, int i, int j) {
566     int temp = (Integer)list.get(i);
567     list.set(i, (Integer)list.get(j));
568     list.set(j, temp);
569     return list;
570      }
571
572      /**
573       *
574       *
575       * param int[] list - list with elements to be swapped
576       * param int i - index of first element to be swapped
577       * param int j - index of second element to be swapped
578       * return int[] - ArrayList with elements swapped
579       *
580       * switches the values of the ith and jth position in the ArrayList
581       */
582      private static double[] swapDoubles(double[] list, int i, int j) {
583     double temp = list[i];
584     list[i] = list[j];
585     list[j] = temp;
586     return list;
587      }
588
589      private static int[] swapInts(int[] list, int i, int j) {
590     int temp = list[i];
591     list[i] = list[j];
592     list[j] = temp;
593     return list;
594      }
```

```
595
596
597      private static ArrayList swapACs(ArrayList list, int i, int j) {
598    AtomContainer temp = (AtomContainer)list.get(i);
599    list.set(i, (AtomContainer)list.get(j));
600    list.set(j, temp);
601    return list;
602      }
603
604      private static ArrayList swapStrings(ArrayList list, int i, int j) {
605    String temp = (String)list.get(i);
606    list.set(i, (String)list.get(j));
607    list.set(j, temp);
608    return list;
609      }
610
611      private static ArrayList swapArrayLists(ArrayList list, int i, int j) {
612    ArrayList temp = (ArrayList)list.get(i);
613    list.set(i, (ArrayList)list.get(j));
614    list.set(j, temp);
615    return list;
616      }
617
618      /**
619       *          ---- sort ----
620       *
621       * param ArrayList list - list of Integers
622       * return ArrayList - sorted list of Integers
623       *
624       * sorts the Integers in list
625       */
626      private static ArrayList sort(ArrayList list) {
627    int j;
628    int int1;
629    int int2;
630    for (int i = 0; i < list.size()-1; i++) {
631        for (j = i+1; j < list.size(); j++) {
632      int1 = ((Integer)list.get(i)).intValue();
633      int2 = ((Integer)list.get(j)).intValue();
634      if (int1 > int2) {
635          list = swap(list, i, j);
636      }
637        }
638    }
639    return list;
640      }
641
642      /** gets sorted string of atom identities */
643      private static String getSortedAtomSymbols(AtomContainer ac) {
644    Atom[] atomList = ac.getAtoms();
645    ArrayList symbolsList = new ArrayList();
646    // get all symbols together
647    for (int a = 0; a < atomList.length; a++) {
```

```
648         symbolsList.add(atomList[a].getSymbol());
649     }
650     // sort the symbols
651     for (int i = 0; i < symbolsList.size()-1; i++) {
652         for (int j = i + 1; j < symbolsList.size(); j++) {
653       if (((String)symbolsList.get(i)).compareTo((String)symbolsList.get(j))
          > 0) {
654           symbolsList = swapStrings(symbolsList, i, j);
655       }
656         }
657     }
658     // convert the symbols to a string
659     String symbolsString = new String();
660     for (int i = 0; i < symbolsList.size(); i++) {
661         symbolsString = symbolsString + (String)symbolsList.get(i);
662     }
663     return symbolsString;
664       }
665 
666       /**         --- sortRMapList(ArrayList list) ----
667        * sorts ArrayList of rmaps so that the largest molecule is first
668        */
669       private static ArrayList sortRMapList(ArrayList list) {
670     int j;
671     for (int i = 0; i < list.size()-1; i++) {
672         for (j = i+1; j < list.size(); j++) {
673       int size1 = ((ArrayList)list.get(i)).size();
674       int size2 = ((ArrayList)list.get(j)).size();
675       if (size1 < size2) {
676           list = swapArrayLists(list, i, j);
677       }
678         }
679     }
680     return list;
681       }
682 
683       /** sorts ArrayList of AtomContainers so that the largest molecule is
          first */
684       private static ArrayList sortACList(ArrayList list) {
685     int j;
686     int int1;
687     int int2;
688     AtomContainer ac1;
689     AtomContainer ac2;
690 
691     // then sort by sum of bonds connected to atom
692     for (int i = 0; i < list.size()-1; i++) {
693         for (j = i+1; j < list.size(); j++) {
694       ac1 = (AtomContainer)list.get(i);
695       ac2 = (AtomContainer)list.get(j);
696 
697       int connectionsSum1 = 0;
698       int connectionsSum2 = 0;
```

```
699
700        Atom[] atomList = ac1.getAtoms();
701        for (int a = 0; a < atomList.length; a++) {
702            connectionsSum1 = connectionsSum1 +
               (ac1.getConnectedBonds(atomList[a])).length;
703        }
704        atomList = ac2.getAtoms();
705        for (int a = 0; a < atomList.length; a++) {
706            connectionsSum2 = connectionsSum2 +
               (ac2.getConnectedBonds(atomList[a])).length;
707        }
708
709        if (connectionsSum1 > connectionsSum2) {
710            list = swapACs(list, i, j);
711        }
712          }
713      }
714
715      // first sorting by sum of bond order
716      for (int i = 0; i < list.size()-1; i++) {
717          for (j = i+1; j < list.size(); j++) {
718        ac1 = (AtomContainer)list.get(i);
719        ac2 = (AtomContainer)list.get(j);
720
721        double bondOrderSum1 = 0;
722        double bondOrderSum2 = 0;
723
724        Bond[] bondList = ac1.getBonds();
725        for (int b = 0; b < bondList.length; b++) {
726            bondOrderSum1 = bondOrderSum1 + bondList[b].getOrder();
727        }
728        bondList = ac2.getBonds();
729        for (int b = 0; b < bondList.length; b++) {
730            bondOrderSum2 = bondOrderSum2 + bondList[b].getOrder();
731        }
732
733        if (bondOrderSum1 > bondOrderSum2) {
734            list = swapACs(list, i, j);
735        }
736          }
737      }
738
739      // sorting by # bonds, descending
740      for (int i = 0; i < list.size()-1; i++) {
741          for (j = i+1; j < list.size(); j++) {
742        int1 = ((AtomContainer)list.get(i)).getBondCount();
743        int2 = ((AtomContainer)list.get(j)).getBondCount();
744        if (int1 < int2) {
745            list = swapACs(list, i, j);
746        }
747          }
748      }
749
```

```
750     return list;
751       }
752 
753       /** sorts ArrayList of AtomContainers so that the largest molecule is
          first */
754       private static ArrayList sortACList(ArrayList list, ArrayList rmapList)
          {
755     int j;
756     int int1;
757     int int2;
758     AtomContainer ac1;
759     AtomContainer ac2;
760 
761     // then sort by sum of bonds connected to atom
762     for (int i = 0; i < list.size()-1; i++) {
763         for (j = i+1; j < list.size(); j++) {
764       ac1 = (AtomContainer)list.get(i);
765       ac2 = (AtomContainer)list.get(j);
766 
767       int connectionsSum1 = 0;
768       int connectionsSum2 = 0;
769 
770       Atom[] atomList = ac1.getAtoms();
771       for (int a = 0; a < atomList.length; a++) {
772           connectionsSum1 = connectionsSum1 +
              (ac1.getConnectedBonds(atomList[a])).length;
773       }
774       atomList = ac2.getAtoms();
775       for (int a = 0; a < atomList.length; a++) {
776           connectionsSum2 = connectionsSum2 +
              (ac2.getConnectedBonds(atomList[a])).length;
777       }
778 
779       if (connectionsSum1 > connectionsSum2) {
780           list = swapACs(list, i, j);
781           if (rmapList.size() > j) {
782         rmapList = swapArrayLists(rmapList, i, j);
783           }
784       }
785         }
786     }
787 
788     // first sorting by sum of bond order
789     for (int i = 0; i < list.size()-1; i++) {
790         for (j = i+1; j < list.size(); j++) {
791       ac1 = (AtomContainer)list.get(i);
792       ac2 = (AtomContainer)list.get(j);
793 
794       double bondOrderSum1 = 0;
795       double bondOrderSum2 = 0;
796 
797       Bond[] bondList = ac1.getBonds();
798       for (int b = 0; b < bondList.length; b++) {
```

```
799             bondOrderSum1 = bondOrderSum1 + bondList[b].getOrder();
800         }
801         bondList = ac2.getBonds();
802         for (int b = 0; b < bondList.length; b++) {
803             bondOrderSum2 = bondOrderSum2 + bondList[b].getOrder();
804         }
805
806         if (bondOrderSum1 > bondOrderSum2) {
807             list = swapACs(list, i, j);
808             if (rmapList.size() > j) {
809           rmapList = swapArrayLists(rmapList, i, j);
810             }
811         }
812           }
813       }
814
815       // sorting by # bonds, descending
816       for (int i = 0; i < list.size()-1; i++) {
817           for (j = i+1; j < list.size(); j++) {
818         int1 = ((AtomContainer)list.get(i)).getBondCount();
819         int2 = ((AtomContainer)list.get(j)).getBondCount();
820         if (int1 < int2) {
821             list = swapACs(list, i, j);
822             if (rmapList.size() > j) {
823           rmapList = swapArrayLists(rmapList, i, j);
824             }
825         }
826           }
827       }
828
829       return list;
830         }
831
832         /**          ---- getReorderedIndices ----
833          *
834          * param int[] numOverlapsList
835          * return int[] with indices of numOverlapsList in sorted order
836          */
837         private static int[] getReorderedIndices(double[] numOverlapsList) {
838       int[] reorderedIndices = new int[numOverlapsList.length];
839       // intialize indices list to 0, 1, 2, 3, 4, 5...
840       for (int i = 0; i < reorderedIndices.length; i++) {
841           reorderedIndices[i] = i;
842       }
843
844       for (int i = 0; i < numOverlapsList.length-1; i++) {
845           for (int j = i; j < numOverlapsList.length; j++) {
846         if (numOverlapsList[i] > numOverlapsList[j]) {
847             numOverlapsList = swapDoubles(numOverlapsList, i, j);
848             reorderedIndices = swapInts(reorderedIndices, i, j);
849         }
850           }
851       }
```

```
852
853  return reorderedIndices;
854    }
855
856
857    /**
858     *        ---- reverseSubstrate ----
859     *
860     * param Reaction
861     *
862     * reverses the atom order of the substrate in this reaction
863     */
864    private static void reverseSubstrate(Reaction reaction) throws
       IOException, CDKException{
865  RCAtomContainer rcac = reaction.getSubstrateAtomContainer();
866  rcac.reverseSubstrateAtomOrder();
867    }
868
869
870    /**
871     *        ---- reverseProduct ----
872     *
873     * param Reaction
874     *
875     * reverses the atom order of the product in this reaction
876     */
877    private static void reverseProduct(Reaction reaction) throws
       IOException, CDKException{
878  RCAtomContainer rcac = reaction.getProductAtomContainer();
879  rcac.reverseSubstrateAtomOrder();
880    }
881
882    /* -----------------------------------------------------------------
883     *         atom and molecule comparison methods
884     * -----------------------------------------------------------------
885     */
886
887    /**
888     *          ---- atomIsPresent ----
889     * param Atom a
890     * param AtomContainer ac
891     *
892     * returns boolean
893     *
894     * returns whether atom instance is present in atom container
895     */
896    private static boolean atomIsPresent(Atom a, AtomContainer ac) {
897  Atom[] atomList = ac.getAtoms();
898  for (int i=0; i < atomList.length; i++) {
899      if (atomList[i].compare(a)) {
900    return true;
901      }
902  }
```

```
903    return false;
904      }
905
906
907      /**
908       *          ---- getAtomInMolecule ----
909       *
910       * param Atom atom
911       * param AtomContainer molecule
912       *
913       * return boolean - true if atom with that coordinates, atom type are
       present, false otherwise
914       */
915      private static Atom getAtomInMolecule(Atom atom, AtomContainer
      molecule) {
916    Atom molAtom;
917    Atom[] atomList = molecule.getAtoms();
918    // loop through all atoms in molecule
919    for (int i = 0; i < atomList.length; i++) {
920        // check to see if this atom has same info as parameter atom
921        molAtom = atomList[i];
922        // check to see if it's using 3d or 2d coordinates
923        if (atom.getX2d() - molAtom.getX2d() == 0 && atom.getY2d() -
        molAtom.getY2d() == 0 && atom.getX2d() == 0 && molAtom.getY2d() == 0)
        {
924      if (atom.getX3d() == molAtom.getX3d() && atom.getY3d() ==
      molAtom.getY3d() && atom.getZ3d() == molAtom.getZ3d() &&
      atom.getSymbol().equals(molAtom.getSymbol())) {
925          return molAtom;
926      }
927        } else {
928      if (atom.getX2d() == molAtom.getX2d() && atom.getY2d() ==
      molAtom.getY2d() && atom.getSymbol().equals(molAtom.getSymbol())) {
929          return molAtom;
930      }
931        }
932
933    }
934    return null;
935      }
936
937      private static boolean atomsHaveIdenticalCoordinates(Atom a1, Atom a2)
      {
938    if (a1.getX2d() - a2.getX2d() == 0 && a1.getY2d() - a2.getY2d() == 0 &&
    a1.getX2d() == 0 && a2.getY2d() == 0) {
939        if (a1.getX3d() == a2.getX3d() && a1.getY3d() == a2.getY3d() &&
        a1.getZ3d() == a2.getZ3d() && a1.getSymbol().equals(a2.getSymbol()))
        {
940      return true;
941        }
942    } else {
943        if (a1.getX2d() == a2.getX2d() && a1.getY2d() == a2.getY2d() &&
        a1.getSymbol().equals(a2.getSymbol())) {
```

```
944      return true;
945        }
946    }
947    return false;
948      }
949
950      /**
951       *           ---- getBondInMolecule ----
952       *
953       * param Bond bond
954       * param AtomContainer molecule
955       *
956       * return Bond in molecule that has same info as bond
957       */
958      private static Bond getBondInMolecule(Bond bond, AtomContainer
         molecule) {
959    Bond molBond;
960    Bond[] bondList = molecule.getBonds();
961    Point2d zeropoint = new Point2d(0.0, 0.0);
962
963    Atom[] molBondAtoms;
964    Atom[] bondAtoms;
965
966    // loop through all bonds in molecule
967    for (int i = 0; i < bondList.length; i++) {
968        // check to see if this bond has the same info as parameter bond
969        molBond = bondList[i];
970        if ( (bond.get2DCenter().distance(zeropoint) == 0) &&
           (molBond.get2DCenter().distance(zeropoint) == 0) ) {
971      if (bond.get3DCenter().distance(molBond.get3DCenter()) == 0) {
972          // get both atoms of bond
973          bondAtoms = bond.getAtoms();
974          // get both atoms of molBond
975          molBondAtoms = molBond.getAtoms();
976          // if atomCoordinates are equal (first way or second way)
977          if ( (atomsHaveIdenticalCoordinates(bondAtoms[0], molBondAtoms[0])
             && atomsHaveIdenticalCoordinates(bondAtoms[1], molBondAtoms[1]) )
             || ( atomsHaveIdenticalCoordinates(bondAtoms[0], molBondAtoms[1])
             && atomsHaveIdenticalCoordinates(bondAtoms[1], molBondAtoms[0]))) {
978        if (bond.getOrder() == molBond.getOrder()) {
979            return molBond;
980        }
981          }
982      }
983        }
984        else {
985      if (bond.get2DCenter().distance(molBond.get2DCenter()) == 0) {
986          // get both atoms of bond
987          bondAtoms = bond.getAtoms();
988          // get both atoms of molBond
989          molBondAtoms = molBond.getAtoms();
990          // if atomCoordinates are equal (first way or second way)
```

```
991         if ( (atomsHaveIdenticalCoordinates(bondAtoms[0], molBondAtoms[0])
            && atomsHaveIdenticalCoordinates(bondAtoms[1], molBondAtoms[1]) )
            || ( atomsHaveIdenticalCoordinates(bondAtoms[0], molBondAtoms[1])
            && atomsHaveIdenticalCoordinates(bondAtoms[1], molBondAtoms[0]))) {
992       if (bond.getOrder() == molBond.getOrder()) {
993           return molBond;
994       }
995         }
996     }
997       }
998   }
999   return null;
1000    }
1001
1002    /**
1003     *         ---- countOverlappingAtoms ----
1004     *
1005     * return int number of ac has any of the same atoms as ac2
1006     */
1007    public static int countOverlappingAtoms(AtomContainer ac1,
        AtomContainer ac2) {
1008  Atom[] atomList = ac1.getAtoms();
1009  Atom testAtom;
1010  int count = 0;
1011  for(int a = 0; a < atomList.length; a++) {
1012      testAtom = getAtomInMolecule(atomList[a], ac2);
1013      if (testAtom != null) {
1014    count = count + 1;
1015      }
1016  }
1017  return count;
1018    }
1019
1020    /**
1021     *         ---- getOverlappingAtoms ----
1022     *
1023     * param AtomContainer ac1
1024     * param AtomContainer ac2
1025     *
1026     * returns AtomContainer with overlapping atoms
1027     * !warning! doesn't handle bonds now
1028     */
1029    public static AtomContainer getOverlappingAtoms(AtomContainer ac1,
        AtomContainer ac2) throws CDKException, IOException {
1030  Atom[] atomList = ac1.getAtoms();
1031  Atom testAtom;
1032  AtomContainer overlappingAC = new AtomContainer();
1033  for (int a = 0; a < atomList.length; a++) {
1034
1035      testAtom = getAtomInMolecule(atomList[a], ac2);
1036      if (testAtom != null) {
1037    overlappingAC.addAtom(testAtom);
1038      }
```

```
1039    }
1040    return overlappingAC;
1041      }
1042
1043      /**
1044       *       ---- areEquivalentAtomContainers ----
1045       *
1046       * param AtomContainer ac1
1047       * param AtomContainer ac2
1048       * return boolean whether AtomContainers have same number of atoms with
         equivalent coordinates
1049       *
1050       */
1051      private static boolean areEquivalentAtomContainers(AtomContainer ac1,
        AtomContainer ac2) {
1052    int numOverlappingAtoms = countOverlappingAtoms(ac1, ac2);
1053    if (numOverlappingAtoms == ac1.getAtomCount() && numOverlappingAtoms ==
    ac2.getAtomCount()) {
1054        return true;
1055    }
1056    return false;
1057      }
1058
1059      /**        --- getNonOverlappingSubstructure ----
1060       * getNonOverlappingSubstructure
1061       *
1062       * param AtomContainer molecule
1063       * param AtomContainer substructure
1064       *
1065       * returns AtomContainer part of molecule that is not contained in
         substructure
1066       */
1067      private static AtomContainer
        getNonOverlappingSubstructure(AtomContainer molecule, AtomContainer
        substructure) throws IOException, CDKException{
1068    Atom molAtom;
1069    Bond molBond;
1070    Vector atomVector;
1071    int i;
1072    // check that substructure is substructure of molecule
1073    // copy the molecule
1074    AtomContainer acCopy = (AtomContainer)molecule.clone();
1075    Bond[] substructureBondList = substructure.getBonds();
1076
1077    // loop through bonds of substructure
1078    for (i = 0; i < substructureBondList.length; i++) {
1079        molBond = getBondInMolecule(substructureBondList[i], acCopy);
1080        if (molBond != null) {
1081      // get atoms connected to this bond
1082      atomVector = molBond.getAtomsVector();
1083      // remove that bond
1084      acCopy.removeBond((Atom)atomVector.get(0), (Atom)atomVector.get(1));
1085        }
```

```
1086    }
1087    // remove disconnected atoms
1088    Atom[] acCopyAtomList = acCopy.getAtoms();
1089    for (i = 0; i < acCopyAtomList.length; i++) {
1090        if (acCopy.getBondCount(acCopyAtomList[i]) == 0) {
1091      acCopy.removeAtom(acCopyAtomList[i]);
1092        }
1093    }
1094    return acCopy;
1095      }
1096
1097
1098      /**           ---- getNonOverlappingSubstructureDiffCoord ----
1099       * getNonOverlappingSubstructureDiffCoord
1100       *
1101       * param AtomContainer molecule
1102       * param AtomContainer substructure
1103       *
1104       * returns AtomContainer part of molecule that is not contained in
         substructure
1105       */
1106      private static AtomContainer
          getNonOverlappingSubstructureDiffCoord(AtomContainer molecule,
          AtomContainer substructure) throws IOException, CDKException{
1107    List acList = uit.getOverlaps(molecule, substructure);
1108    AtomContainer matchingSubstructure;
1109    if (acList.size() != 1) {
1110        ArrayList betterList = sortACList(new ArrayList(acList));
1111        if (acList.size() > 0) {
1112      matchingSubstructure = (AtomContainer)betterList.get(0);
1113        } else {
1114      return new AtomContainer();
1115        }
1116    } else {
1117        matchingSubstructure = (AtomContainer)acList.get(0);
1118    }
1119    if (matchingSubstructure.getBondCount() == substructure.getBondCount()) {
1120        return getNonOverlappingSubstructure(molecule, matchingSubstructure);
1121    } else {
1122        return getNonOverlappingSubstructure(molecule, matchingSubstructure);
1123    }
1124
1125      }
1126
1127      /**            --- sortBondArray(Bond[] bonds) ---
1128       * sorts bond array by some measure
1129       * any measure, doesn't matter, just has to be consistent
1130       */
1131
1132      private static boolean areBondsEquivalent(Bond b1, Bond b2) {
1133    Point2d zeropoint = new Point2d(0.0, 0.0);
1134    if ( (b1.get2DCenter().distance(zeropoint) == 0) &&
        (b2.get2DCenter().distance(zeropoint) == 0) ) {
```

```
1135        if (b1.get3DCenter().distance(b2.get3DCenter()) == 0) {
1136      return true;
1137        }
1138   } else {
1139        if (b1.get2DCenter().distance(b2.get2DCenter()) == 0) {
1140      return true;
1141        }
1142   }
1143   return false;
1144     }
1145
1146     // given a list of atoms, checks to see if atom is in list
1147     // checks coordinates and stereoparity
1148     private static boolean atomInList(Atom atom, ArrayList aList) {
1149   Atom tempAtom;
1150   for (int a = 0; a < aList.size(); a++) {
1151        tempAtom = (Atom)aList.get(a);
1152        if ( (tempAtom.getX2d() == atom.getX2d()) && (tempAtom.getY2d() ==
            atom.getY2d()) && (tempAtom.getStereoParity() ==
            atom.getStereoParity()) ) {
1153      return true;
1154        }
1155   }
1156   return false;
1157     }
1158
1159     /**       --- getBondString ---
1160      * param Bond
1161      * returns String - alphebetized string with atoms on either side of
          bond with bond order in front
1162      */
1163     private static String getBondString(Bond b) {
1164   String bondString = new String();
1165   //add in bond order
1166   bondString = bondString.concat(String.valueOf(b.getOrder()));
1167   // get atoms, sort them, add them to bondString
1168   Atom[] atoms = b.getAtoms();
1169   String a0 = atoms[0].getSymbol();
1170   String a1 = atoms[1].getSymbol();
1171   if (a0.compareTo(a1) > 0) {
1172        bondString = bondString.concat(a1);
1173        bondString = bondString.concat(a0);
1174   } else {
1175        bondString = bondString.concat(a0);
1176        bondString = bondString.concat(a1);
1177   }
1178   return bondString;
1179     }
1180
1181     /**          --- sortStringList ---
1182      * param ArrayList - of Strings
1183      * returns ArrayList - alphabetized list of strings
1184      */
```

```
1185     private static String[] sortStringArray(String[] stringArray) {
1186   String str1;
1187   String str2;
1188   String tmpStr;
1189
1190   for (int i = 0; i < stringArray.length-1; i++) {
1191       for (int j = i+1; j < stringArray.length; j++) {
1192     str1 = stringArray[i];
1193     str2 = stringArray[j];
1194
1195     if (str1.compareTo(str2) > 0) {
1196         // swap
1197         stringArray[i] = str2;
1198         stringArray[j] = str1;
1199     }
1200       }
1201   }
1202   return stringArray;
1203     }
1204
1205     /**           --- getBondArrayString ---
1206      * param Bond[] - of Bonds
1207      * returns String - alphabetized String of all bonds concatenated
1208      */
1209     private static String getBondArrayString(Bond[] bondList) {
1210
1211   Bond bond;
1212   String bondString;
1213
1214   String[] stringArray = new String[bondList.length];
1215   String wholeString = new String();
1216   for (int b = 0; b < bondList.length; b++) {
1217       bond = bondList[0];
1218       bondString = getBondString(bond);
1219       stringArray[b] = bondString;
1220   }
1221   stringArray = sortStringArray(stringArray);
1222   for (int i = 0; i < stringArray.length; i++) {
1223       wholeString = wholeString.concat(stringArray[i]);
1224   }
1225   return wholeString;
1226     }
1227
1228     private static Bond[]
         getBondsConnectedToAtomNotIncludingCurrentBond(AtomContainer ac, Atom
         a, Bond b) {
1229   Bond[] allConnectedBonds = ac.getConnectedBonds(a);
1230   if (allConnectedBonds.length > 1) {
1231       Bond[] newConnectedBonds = new Bond[allConnectedBonds.length-1];
1232       int bondListLength = 0;
1233       if (newConnectedBonds.length == 0) {
1234     return new Bond[0];
1235       }
```

```
1236        for (int i = 0; i < allConnectedBonds.length; i++) {
1237      if (!areBondsEquivalent(allConnectedBonds[i], b)) {
1238          if (bondListLength < newConnectedBonds.length) {
1239        newConnectedBonds[bondListLength] = allConnectedBonds[i];
1240        bondListLength++;
1241          }
1242      }
1243        }
1244        return newConnectedBonds;
1245    } else {
1246        return new Bond[0];
1247    }
1248
1249      }
1250
1251      /**        --- areAtomsForBondSwitched ---
1252       * param Bond first bond
1253       * param Bond second bond
1254       * param AtomContainer substrateSubstructure - common substructure,
         uses substrate's coordinates
1255       * param AtomContainer productSubstructure - common substructure, uses
         product's coordinates
1256       * returns boolean false if a0-b0 and a1-b1, true if a0-b1 and a1-b0
1257       *
1258       * determines whether atom order needs to be switched to get
1259       * proper correspondence
1260       */
1261      private static boolean areAtomsForBondSwitched(Bond substrateBond, Bond
      productBond, Atom[] substrateAtoms, Atom[] productAtoms, AtomContainer
      substrateSubstructure, AtomContainer productSubstructure) {
1262    boolean correspondenceFound = true;
1263
1264    Bond[] substrateConnectedBonds0;
1265    Bond[] substrateConnectedBonds1;
1266    Bond[] productConnectedBonds0;
1267    Bond[] productConnectedBonds1;
1268    // solution #1 first do obvious check when symbols are different
1269    if (substrateAtoms[0].getSymbol() != substrateAtoms[1].getSymbol()) {
1270        correspondenceFound = true;
1271        if (substrateAtoms[0].getSymbol() == productAtoms[0].getSymbol() &&
        substrateAtoms[1].getSymbol() == productAtoms[1].getSymbol()) { // if
        atom1 in substrate is same as atom1 in product
1272      return false;
1273        } else {
1274      return true;
1275        }
1276    } else { // find adjacent bonds in substructures (substrate and product
    coordinates)
1277        // save atoms of substructure
1278        Atom productAtom0 = getAtomInMolecule(productAtoms[0],
        productSubstructure);
1279        Atom productAtom1 = getAtomInMolecule(productAtoms[1],
        productSubstructure);
```

```
1280        Atom substrateAtom0 = getAtomInMolecule(substrateAtoms[0],
            substrateSubstructure);
1281        Atom substrateAtom1 = getAtomInMolecule(substrateAtoms[1],
            substrateSubstructure);
1282
1283        if (productAtom1 == null) {
1284      System.out.println("productatom0 null");
1285      System.out.println(productAtoms.length);
1286        }
1287        productConnectedBonds0 =
            getBondsConnectedToAtomNotIncludingCurrentBond(productSubstructure,
            productAtom0, productBond);
1288        productConnectedBonds1 =
            getBondsConnectedToAtomNotIncludingCurrentBond(productSubstructure,
            productAtom1, productBond);
1289        substrateConnectedBonds0 =
            getBondsConnectedToAtomNotIncludingCurrentBond(substrateSubstructure,
            substrateAtom0, substrateBond);
1290        substrateConnectedBonds1 =
            getBondsConnectedToAtomNotIncludingCurrentBond(substrateSubstructure,
            substrateAtom1, substrateBond);
1291
1292        // solution #2 if number of connected bonds on each atom is different
1293        if (substrateConnectedBonds0.length !=
            substrateConnectedBonds1.length) {
1294      correspondenceFound = true;
1295      if ( (substrateConnectedBonds0.length == productConnectedBonds0.length)
          && (substrateConnectedBonds1.length == productConnectedBonds1.length))
          {
1296          return false;
1297      } else {
1298          return true;
1299      }
1300        } else {
1301      // check to see if bond identities are the same
1302      String substrateS0 = getBondArrayString(substrateConnectedBonds0);
1303      String substrateS1 = getBondArrayString(substrateConnectedBonds1);
1304      String productS0 = getBondArrayString(productConnectedBonds0);
1305      String productS1 = getBondArrayString(productConnectedBonds1);
1306
1307      if (substrateS0.compareTo(substrateS1) != 0) {
1308          correspondenceFound = true;
1309          if ( (substrateS0.compareTo(productS0) == 0) &&
              (substrateS1.compareTo(productS1) == 0) ) {
1310        return false;
1311          } else {
1312        return true;
1313          }
1314      } else {
1315          correspondenceFound = false;
1316          return false;
1317      }
1318        }
```

```
1319        // if they are not switched, # connected bonds should be the same,
            type of bonds should be the same
1320    }
1321      }
1322
1323
1324      /** -----------------------------------------------------------------
1325       *          MCS methods
1326       * ------------------------------------------------------------------
1327       */
1328
1329
1330      /**       ---- getMCSFromPair -----
1331       *
1332       * param AtomContainer mol1, first molecule
1333       * param AtomContainer mol2, second molecule
1334       *
1335       * this is the new recursive implementation that will solve lots of
         problems 2/1/07
1336       */
1337
1338      private static void getMCSFromPair(AtomContainer mol1, AtomContainer
          mol2, AtomContainer wholeMolecule, AtomContainer prodMolecule) throws
          CDKException, IOException {
1339    ArrayList acList = new ArrayList(uit.getOverlaps((AtomContainer)mol1,
        (AtomContainer)mol2));
1340    ArrayList rmapList = new
        ArrayList(uit.getOverlapMaps((AtomContainer)mol1, (AtomContainer)mol2));
1341
1342    acList = sortACList(acList, rmapList);
1343    rmapList = sortRMapList(rmapList);
1344
1345    if (acList.size() > 0) {
1346        // remove first subgraph from substrate and product
1347        AtomContainer firstSubgraph = (AtomContainer)acList.get(0);
1348        AtomContainer newMol1 = getNonOverlappingSubstructure(mol1,
            firstSubgraph);
1349        AtomContainer newMol2 = getNonOverlappingSubstructureDiffCoord(mol2,
            firstSubgraph);
1350
1351        // call function again, add it to first subgraph and return
1352        if (firstSubgraph.getBondCount() == 0) {
1353      return;
1354        }
1355        else if ((newMol1.getBondCount() > 0) && (newMol2.getBondCount() >
            0)) {  // if there are still bonds left to compare
1356       getMCSFromPair(newMol1, newMol2, wholeMolecule, prodMolecule);
1357        }
1358         acStack.add(0, firstSubgraph);
1359         // renumber rmaps based on wholeMolecule coordinates
1360        ArrayList rmapFirstSubgraph = (ArrayList)rmapList.get(0);
1361        // loop through bonds
1362        for (int r = 0; r < rmapFirstSubgraph.size(); r++) {
```

```
1363      RMap rmap = (RMap)rmapFirstSubgraph.get(r);
1364
1365      // take care of substrate
1366      Bond subgraphBond = mol1.getBondAt(rmap.getId1());
1367      // get the number of that bond in the whole molecule
1368      int wholeMolId =
          wholeMolecule.getBondNumber(getBondInMolecule(subgraphBond,
          wholeMolecule));
1369      // reassign RMap ids
1370      if (wholeMolId > -1) {
1371          rmap.setId1(wholeMolId);
1372      } else {
1373          // didn't find bond in molecule
1374      }
1375
1376      // take care of product
1377      subgraphBond = mol2.getBondAt(rmap.getId2());
1378      int prodMolId =
          prodMolecule.getBondNumber(getBondInMolecule(subgraphBond,
          prodMolecule));
1379      if (prodMolId > -1) {
1380          rmap.setId2(prodMolId);
1381      } else { // didn't find bond in molecule
1382      }
1383
1384        }
1385        rmapStack.add(0, rmapFirstSubgraph);
1386        return;
1387    } else {
1388        return;
1389    }
1390      }
1391
1392      /**       --- switchRMapIds ---
1393       * sets id1 to be id2 and id2 to be id1
1394       */
1395      private static ArrayList switchRMapIds(ArrayList rmapList) {
1396    for (int i = 0; i < rmapList.size(); i++) {
1397        RMap rmap = (RMap)rmapList.get(i);
1398        int tempId1 = rmap.getId1();
1399        rmap.setId1(rmap.getId2());
1400        rmap.setId2(tempId1);
1401    }
1402    return rmapList;
1403      }
1404
1405      /**       ---- getMCSFromPairSecondCoords -----
1406       *
1407       * param AtomContainer mol1, first molecule
1408       * param AtomContainer mol2, second molecule
1409       *
1410       * this is the new recursive implementation that will solve lots of
          problems 2/1/07
```

```
1411         * uses uit getOverlaps to make sure overlap is the same when things
             are reversed
1412         */
1413
1414        private static void getMCSFromPairSecondCoords(AtomContainer mol1,
            AtomContainer mol2, AtomContainer wholeMolecule, AtomContainer
            prodMolecule) throws CDKException, IOException {
1415      ArrayList acList = new
          ArrayList(uit.getOverlapsSecondCoords((AtomContainer)mol1,
          (AtomContainer)mol2));
1416      ArrayList rmapList = new
          ArrayList(uit.getOverlapMapsSecondCoords((AtomContainer)mol1,
          (AtomContainer)mol2));
1417      acList = sortACList(acList, rmapList);
1418      rmapList = sortRMapList(rmapList);
1419
1420      if (acList.size() > 0) {
1421          // remove first subgraph from substrate and product
1422          AtomContainer firstSubgraph = (AtomContainer)acList.get(0);
1423          AtomContainer newMol2 = getNonOverlappingSubstructure(mol2,
              firstSubgraph);
1424          AtomContainer newMol1 = getNonOverlappingSubstructureDiffCoord(mol1,
              firstSubgraph);
1425
1426          // call function again, add it to first subgraph and return
1427          if (firstSubgraph.getBondCount() == 0) {
1428        return;
1429          }
1430          else if ((newMol1.getBondCount() > 0) && (newMol2.getBondCount() >
              0)) {  // if there are still atoms left to compare
1431        getMCSFromPairSecondCoords(newMol1, newMol2, wholeMolecule,
            prodMolecule);
1432          }
1433          acStack.add(0, firstSubgraph);
1434          // renumber rmaps based on wholeMolecule coordinates
1435          ArrayList rmapFirstSubgraph =
              switchRMapIds((ArrayList)rmapList.get(0));
1436          // loop through bonds
1437          for (int r = 0; r < rmapFirstSubgraph.size(); r++) {
1438        RMap rmap = (RMap)rmapFirstSubgraph.get(r);
1439
1440        // take care of product
1441        Bond subgraphBond = mol2.getBondAt(rmap.getId1());
1442        // get the number of that bond in the whole molecule
1443        int prodMolId =
            prodMolecule.getBondNumber(getBondInMolecule(subgraphBond,
            prodMolecule));
1444        // reassign RMap ids
1445        if (prodMolId > -1) {
1446            rmap.setId1(prodMolId);
1447        } else {
1448            // didn't find bond in molecule
1449        }
```

```
1450
1451    // take care of substrate
1452    subgraphBond = mol1.getBondAt(rmap.getId2());
1453    int subMolId =
        wholeMolecule.getBondNumber(getBondInMolecule(subgraphBond,
        wholeMolecule));
1454    if (subMolId > -1) {
1455        rmap.setId2(subMolId);
1456    } else {
1457        // didn't find bond in molecule
1458    }
1459      }
1460      rmapStack.add(0, rmapFirstSubgraph);
1461      return;
1462  } else {
1463      return;
1464  }
1465    }
1466
1467    /**
1468     *           ---- getMCS ----
1469     *
1470     * param ArrayList molList - list of molecules
1471     * param int[] referenceIndex - index of molecule which provides
         coordinates for substructure
1472     * return ArrayList - List of MCS found
1473     *
1474     * Gets Maxmum Common Subgraph from a list of molecules
1475     * right now starts MCS looping with reference molecule, but I might
         need to change that so that I do all things in the same order
1476     * may need to add referenceIndex parameter
1477     *
1478     */
1479    private static AtomContainer getMCS(ArrayList molList) throws
        CDKException, IOException{
1480
1481  ArrayList subgraphs = null;
1482  if (molList.size() == 1) {
1483      return (AtomContainer)molList.get(0);
1484  } else if (molList.size() == 0) {
1485      return new AtomContainer();
1486  } else {
1487      // pop first two items from stack and get pairwise MCS (all
          possibilities)
1488      ArrayList listToUse = new ArrayList(molList);
1489      AtomContainer mol1 = (AtomContainer)listToUse.remove(0);
1490      AtomContainer mol2 = (AtomContainer)listToUse.remove(0);
1491
1492      // order of uit.getOverlaps doesn't matter because I find isomorphs
          with each substrate later
1493      if (mol1.getBondCount() >= mol2.getBondCount()) {
1494    subgraphs = new ArrayList(uit.getOverlaps((AtomContainer)mol1,
        (AtomContainer)mol2));
```

```
1495        } else {
1496      subgraphs = new ArrayList(uit.getOverlaps((AtomContainer)mol2,
          (AtomContainer)mol1));
1497        }
1498
1499        ArrayList savedArrayList = new ArrayList(listToUse);
1500        ArrayList conservedSubstructureList = new ArrayList();
1501        // for each subgraph
1502        for (int i = 0; i < subgraphs.size(); i++) {
1503      // push subgraph into new stack and call MCS again
1504      ArrayList currentArrayList = new ArrayList(savedArrayList);
1505      currentArrayList.add(0, (AtomContainer)subgraphs.get(i));
1506      // add each answer to a list
1507      conservedSubstructureList.add(getMCS(currentArrayList));
1508        }
1509
1510        // pick the largest answer in the list to return
1511        int largestAC = 0;
1512        int largestACIndex = -1;
1513        for (int i = 0; i < conservedSubstructureList.size(); i++) {
1514      AtomContainer currentCSS =
          (AtomContainer)conservedSubstructureList.get(i);
1515      if (currentCSS.getBondCount() >= largestAC) {
1516          largestAC = currentCSS.getBondCount();
1517          largestACIndex = i;
1518      }
1519        }
1520        return (AtomContainer)conservedSubstructureList.get(largestACIndex);
1521    }
1522      }
1523
1524      /**           --- getAllIsomorphs ---
1525       * returns a list of all isomorphic subgraphs present in mol
1526       */
1527      private static ArrayList getAllIsomorph(AtomContainer mol,
          AtomContainer subgraph) throws IOException, CDKException{
1528    //find one, removing that, keep looking, until you can't find any more or
        there are no more atoms left
1529    boolean foundNoMore = false;
1530    AtomContainer molToUse = mol;
1531    ArrayList isomorphList = new ArrayList();
1532    while (!foundNoMore && molToUse.getAtomCount() > 0) {
1533        // try to find one
1534        ArrayList foundList = new ArrayList(uit.getOverlaps(molToUse,
            subgraph));
1535        foundList = sortACList(foundList);
1536        if (foundList.size() > 0) { // if found
1537      // add to list
1538      AtomContainer foundAC = (AtomContainer)foundList.get(0);
1539      if (foundAC.getBondCount() > 0 && foundAC.getBondCount() ==
          subgraph.getBondCount()) {
1540          isomorphList.add(foundAC);
1541          // remove that and loop through again
```

```
1542          molToUse = getNonOverlappingSubstructure(molToUse, foundAC);
1543      } else {
1544          foundNoMore = true;
1545      }
1546        } else {  // if not found
1547      // flip foundNoMore
1548      foundNoMore = true;
1549        }
1550
1551    }
1552    return isomorphList;
1553      }
1554
1555
1556      /* -----------------------------------------------------------------
1557       *       main superfamily and substrate/product MCS methods
1558       * -----------------------------------------------------------------
1559       */
1560
1561      /**
1562       *             ---- loopThroughMCS ----
1563       *
1564       * param ArrayList molList - list of Reactions
1565       * return ArrayList molList - list of Reactions with conserved
       substructure added
1566       *
1567       * may change this to doConservedSubstructure b/c I'm not looping
       anymore
1568       */
1569      private static void loopThroughMCS(ArrayList molList) throws
      CDKException, IOException {
1570
1571    // find conservedSubstructure that has coordinates of first molecule in
    list
1572    ArrayList acList = getAtomContainerList(molList);
1573    AtomContainer conservedSubstructure = getMCS(acList);
1574    System.out.println("conserved \t\t\t\t" +
    conservedSubstructure.getAtomCount() + "\t" +
    conservedSubstructure.getBondCount());
1575    printSMILES(conservedSubstructure);
1576    System.out.println("#conserved substructure occurrences");
1577    // loop through other molecules and find all occurrences of substructure
    in substrates
1578    for (int i = 0; i < acList.size(); i++) {
1579        ArrayList isomorphList = getAllIsomorph((AtomContainer)acList.get(i),
        conservedSubstructure);
1580        Reaction reaction = (Reaction)molList.get(i);
1581        RCAtomContainer substrate = reaction.getSubstrateAtomContainer();
1582        RCAtomContainer product = reaction.getProductAtomContainer();
1583        System.out.print(substrate.getMoleculeName().trim());
1584        if (product != null) {
1585      System.out.print(" - ");
1586      try {
```

```
1587        System.out.print(product.getMoleculeName().trim() + " " +
            reaction.getReactionIndex() + "\n");
1588    } catch (NullPointerException e) {
1589        System.out.println();
1590        continue;
1591    }
1592      } else {
1593    System.out.println(reaction.getReactionIndex());
1594      }
1595      System.out.print("substrate \t\t\t\t" +
      substrate.getLigand().getAtomCount() + "\t" +
      substrate.getLigand().getBondCount() + "\n");
1596      if (product != null) {
1597    System.out.print("product \t\t\t\t" +
    product.getLigand().getAtomCount() + "\t" +
    product.getLigand().getBondCount() + "\n");
1598      }
1599
1600      System.out.println("occurrences \t\t\t" + isomorphList.size());
1601      ((Reaction)molList.get(i)).getSubstrateAtomContainer().
      setConservedSubstructureList(isomorphList);
1602  }
1603  System.out.println("#substructure info");
1604    }
1605
1606
1607
1608    /**       ---- combineACStack ----
1609     *
1610     * starts from beginning of stack and combines atom containers if there
     are no overlaps
1611     * if there are no overlaps, it also puts rMaps into new list
1612     * checks to see that none of atoms are in otherAC (by mapping
     combinedAC to otherAC)
1613     *
1614     * param ArrayList acStack
1615     * param ArrayList rmapStack
1616     * returns Overlap (which has atomContainer and rmap ArrayList)
1617     */
1618    private static Overlaps combineACStack(ArrayList acList, ArrayList
    rmapList, AtomContainer otherAC) throws IOException, CDKException{
1619
1620  AtomContainer combinedAC = new AtomContainer();
1621  AtomContainer combinedOtherAC = new AtomContainer();
1622  ArrayList newRMapList = new ArrayList();
1623  acList = sortACList(acList, rmapList);
1624
1625  // loop through atom containers in stack
1626  for (int ac = 0; ac < acList.size(); ac++) {
1627
1628      Atom[] newAtomsArray = ((AtomContainer)acList.get(ac)).getAtoms();
1629      boolean addCurrentAC = true;
1630
```

```
1631        // loop through atoms in current atom container and make sure they're
            not in already combinedAC
1632        for (int i = 0; i < newAtomsArray.length; i++) {
1633      Atom atom = getAtomInMolecule(newAtomsArray[i], combinedAC);
1634      if (atom != null) {
1635          overlapCount++;
1636          addCurrentAC = false;
1637      }
1638        }
1639        // loop through atoms of in corresponding atoms of current ac and
            make sure they're not in otherAC
1640        // will have to loop through rmaps to get the correspondence
1641        // I think 2 is the id of the otherAC in the rmaps
1642        // don't have to do this if addCurrentAC is already false
1643        AtomContainer otherACToAdd = new AtomContainer();
1644        if (addCurrentAC) {
1645
1646      ArrayList rml = (ArrayList)rmapList.get(ac);
1647      for (int i = 0; i < rml.size(); i++) {
1648          RMap rmap = (RMap)rml.get(i);
1649          Bond otherBond = otherAC.getBondAt(rmap.getId2());
1650          Atom[] bondAtoms = otherBond.getAtoms();
1651          otherACToAdd.addAtom(bondAtoms[0]);
1652          otherACToAdd.addAtom(bondAtoms[1]);
1653          otherACToAdd.addBond(otherBond);
1654
1655            for (int a = 0; a < bondAtoms.length; a++) {
1656        Atom atom = getAtomInMolecule(bondAtoms[a], combinedOtherAC);
1657        if (atom != null) {
1658            addCurrentAC = false;
1659        }
1660            }
1661      }
1662        }
1663
1664        if (addCurrentAC) {
1665      combinedAC.add((AtomContainer)acList.get(ac));
1666      combinedOtherAC.add(otherACToAdd);
1667      newRMapList.add((ArrayList)rmapList.get(ac));
1668        }
1669    }
1670
1671    return new Overlaps(combinedAC, newRMapList);
1672
1673      }
1674
1675
1676      /**
1677       *           ---- doSubstrateProductMCS ----
1678       *
1679       * for all the Reactions that have substrate and product
1680       * find non reacting substructure and set that in the Substrate of the
         reaction
```

```
1681      */
1682     private static void doSubstrateProductMCS(ArrayList reactionList)
         throws IOException, CDKException{
1683   Reaction reaction;
1684   RCAtomContainer substrate;
1685   RCAtomContainer product;
1686   Overlaps commonSubstructure;
1687   Overlaps productSubstructure;
1688   for (int i = 0; i < reactionList.size(); i++) {
1689       overlapCount = 0;
1690       printSMILES(blankMolecule);
1691       reaction = (Reaction)reactionList.get(i);
1692       product = reaction.getProductAtomContainer();
1693       substrate = reaction.getSubstrateAtomContainer();
1694       printSMILES(substrate.getLigand());
1695
1696       System.out.print(substrate.getMoleculeName().trim());
1697
1698       if (product != null) {
1699
1700     System.out.print(" - ");
1701     try {
1702         System.out.print(product.getMoleculeName().trim() + " " +
         reaction.getReactionIndex() + "\n");
1703     } catch (NullPointerException e) {
1704         System.out.println();
1705         continue;
1706     }
1707
1708     // do MCS between substrate and product
1709     printSMILES(product.getLigand());
1710
1711     // initialize new acStack
1712     // do getMCSFromPair...
1713     // add the atomcontainers from acStack into one atomcontainer, without
     adding overlapping atoms
1714       // also take care of rmaps
1715
1716     ArrayList productACList = reaction.getProductACList();
1717     productACList = sortACList(productACList);
1718
1719     acStack = new ArrayList();
1720     rmapStack = new ArrayList();
1721     getMCSFromPair(substrate.getLigand(), product.getLigand(),
     substrate.getLigand(), product.getLigand());
1722     Overlaps nrSubstrateOverlap = combineACStack(acStack, rmapStack,
     product.getLigand());
1723
1724     acStack = new ArrayList();
1725     rmapStack = new ArrayList();
1726     getMCSFromPairSecondCoords(substrate.getLigand(), product.getLigand(),
     substrate.getLigand(), product.getLigand());
```

```
1727    Overlaps nrProductOverlap = combineACStack(acStack, rmapStack,
        substrate.getLigand());
1728
1729    int nonreversedOverlapCount = overlapCount;
1730    overlapCount = 0;
1731
1732    reverseProduct(reaction);
1733    reverseSubstrate(reaction);
1734
1735    acStack = new ArrayList();
1736    rmapStack = new ArrayList();
1737    getMCSFromPair(substrate.getLigand(), product.getLigand(),
        substrate.getLigand(), product.getLigand());
1738    Overlaps rSubstrateOverlap = combineACStack(acStack, rmapStack,
        product.getLigand());
1739
1740    acStack = new ArrayList();
1741    rmapStack = new ArrayList();
1742    getMCSFromPairSecondCoords(substrate.getLigand(), product.getLigand(),
        substrate.getLigand(), product.getLigand());
1743    Overlaps rProductOverlap = combineACStack(acStack, rmapStack,
        substrate.getLigand());
1744
1745    Overlaps substrateOverlap;
1746    Overlaps productOverlap;
1747
1748    if (overlapCount < nonreversedOverlapCount) {
1749        // use reversed
1750        substrateOverlap = rSubstrateOverlap;
1751        productOverlap = rProductOverlap;
1752    } else if (overlapCount > nonreversedOverlapCount) {
1753        // use nonreversed and rever substrate and product back
1754        reverseProduct(reaction);
1755        reverseSubstrate(reaction);
1756        substrateOverlap = nrSubstrateOverlap;
1757        productOverlap = nrProductOverlap;
1758
1759    } else {
1760        // use the one where the rmap length equals the bond count
1761
1762        int nrBondCount = nrSubstrateOverlap.getOverlap().getBondCount();
1763        int rBondCount = rSubstrateOverlap.getOverlap().getBondCount();
1764        int nrMapCount = 0;
1765        for (int r = 0; r < nrSubstrateOverlap.getRMaps().size(); r++) {
1766      ArrayList substructureRMap =
          (ArrayList)nrSubstrateOverlap.getRMaps().get(r);
1767      nrMapCount += substructureRMap.size();
1768        }
1769
1770        int rMapCount = 0;
1771        for (int r = 0; r < rSubstrateOverlap.getRMaps().size(); r++) {
1772      ArrayList substructureRMap =
          (ArrayList)rSubstrateOverlap.getRMaps().get(r);
```

```
1773        rMapCount += substructureRMap.size();
1774          }
1775          if (nrBondCount == nrMapCount) {
1776        substrateOverlap = nrSubstrateOverlap;
1777        productOverlap = nrProductOverlap;
1778        reverseProduct(reaction);
1779        reverseSubstrate(reaction);
1780          } else if (rBondCount == rMapCount) {
1781        substrateOverlap = rSubstrateOverlap;
1782        productOverlap = rProductOverlap;
1783          } else {
1784        if (Math.abs(rBondCount-rMapCount) <=
        Math.abs(nrBondCount-nrMapCount)) {
1785            substrateOverlap = rSubstrateOverlap;
1786            productOverlap = rProductOverlap;
1787        } else {
1788            substrateOverlap = nrSubstrateOverlap;
1789            productOverlap = nrProductOverlap;
1790            reverseProduct(reaction);
1791            reverseSubstrate(reaction);
1792        }
1793          }
1794      }
1795
1796      // remove stereoparity changes from common substructure
1797      if (reaction.isRacemase() ||
      (substrateOverlap.getOverlap().getAtomCount() ==
      substrate.getLigand().getAtomCount())) {
1798          reaction.subtractStereoparityChanges(substrateOverlap.getOverlap(),
          substrateOverlap.getRMaps(), productOverlap.getOverlap());
1799      }
1800
1801      // add commonsubstructure to substrate object
1802      substrate.setNonreactingSubstructure(substrateOverlap.getOverlap());
1803      // to reacting substructure, add atoms which have more bonds in product
1804      reaction.addAtomsOfChangedBonds(substrateOverlap.getRMaps());
1805
1806      // calculate overlaps
1807      substrate.calculateAllOverlaps();
1808
1809      // print results
1810      printSMILES(substrate.getReactingSubstructure());
1811      printSMILES(substrate.getNonreactingSubstructure());
1812
1813      printSMILES(substrate.getConservedSubstructure());
1814      printSMILES(substrate.getUnconservedSubstructure());
1815
1816      printSMILES(substrate.getReactingConservedOverlap());
1817      printSMILES(substrate.getReactingUnconservedOverlap());
1818      printSMILES(substrate.getNonreactingConservedOverlap());
1819      printSMILES(substrate.getNonreactingUnconservedOverlap());
1820
```

```
1821    System.out.print("substrate \t\t\t\t" +
        substrate.getLigand().getAtomCount() + "\t" +
        substrate.getLigand().getBondCount() + "\n");
1822    System.out.print("product \t\t\t\t" +
        product.getLigand().getAtomCount() + "\t" +
        product.getLigand().getBondCount() + "\n");
1823
1824    System.out.print("reacting \t\t\t\t" +
        substrate.getReactingSubstructure().getAtomCount() + "\t" +
        substrate.getReactingSubstructure().getBondCount() + "\n");
1825    System.out.print("nonreacting \t\t\t" +
        substrate.getNonreactingSubstructure().getAtomCount() + "\t" +
        substrate.getNonreactingSubstructure().getBondCount() + "\n");
1826    System.out.print("conserved \t\t\t\t" +
        substrate.getConservedSubstructure().getAtomCount() + "\t" +
        substrate.getConservedSubstructure().getBondCount() + "\n");
1827    System.out.print("unconserved \t\t\t" +
        substrate.getUnconservedSubstructure().getAtomCount() + "\t" +
        substrate.getUnconservedSubstructure().getBondCount() + "\n");
1828
1829    System.out.print("reacting+conserved \t\t" +
        substrate.getReactingConservedOverlap().getAtomCount() + "\t" +
        substrate.getReactingConservedOverlap().getBondCount() + "\n");
1830    System.out.print("reacting+unconserved \t\t" +
        substrate.getReactingUnconservedOverlap().getAtomCount() + "\t" +
        substrate.getReactingUnconservedOverlap().getBondCount() + "\n");
1831    System.out.print("nonreacting+conserved \t\t" +
        substrate.getNonreactingConservedOverlap().getAtomCount() + "\t" +
        substrate.getNonreactingConservedOverlap().getBondCount() + "\n");
1832    System.out.print("nonreacting+unconserved \t" +
        substrate.getNonreactingUnconservedOverlap().getAtomCount() + "\t" +
        substrate.getNonreactingUnconservedOverlap().getBondCount() + "\n");
1833
1834
1835        }
1836  }
1837
1838  // figure out variation in which part of conserved substructure is
      reacting
1839  RCAtomContainer substratei, substratej;
1840  // for each pair of substrates
1841  System.out.println("#all_pairwise_overlap_of_reacting+conserved #atoms
      #bonds #atoms/r+c #atoms/r #bonds/r+c #bonds/r");
1842  for (int i = 0; i < reactionList.size()-1; i++) {
1843      for (int j = i+1; j < reactionList.size(); j++) {
1844    substratei =
        ((Reaction)reactionList.get(i)).getSubstrateAtomContainer();
1845    substratej =
        ((Reaction)reactionList.get(j)).getSubstrateAtomContainer();
1846    if ((substratei.getReactingSubstructure().getAtomCount() != 0) &&
        (substratej.getReactingSubstructure().getAtomCount() != 0)) {
1847        // get correspondence between conserved substructure in one and
            conserved substructure in another
```

```
1848        ArrayList rmapList = new
            ArrayList(uit.getOverlapMaps(substratei.getConservedSubstructure(),
            substratej.getConservedSubstructure()));
1849        rmapList = (ArrayList)rmapList.get(0);
1850
1851        // then see how many atoms and bonds overlap in the two overlaps
1852        // for each bond in rc overlap, get bond identifier in conserved
            substructure
1853        // also add atoms to atomlist
1854        // first substrate
1855        ArrayList bondListi = new ArrayList();
1856        ArrayList atomListi = new ArrayList();
1857        for (int b = 0; b <
            substratei.getReactingConservedOverlap().getBondCount(); b++) {
1858      Bond bond =
          getBondInMolecule(substratei.getReactingConservedOverlap().getBondAt(b),
          substratei.getConservedSubstructure());
1859      bondListi.add(substratei.getConservedSubstructure().getBondNumber(bond));
1860      int atomNum =
          substratei.getConservedSubstructure().getAtomNumber(bond.getAtomAt(0));
1861      if (!atomListi.contains(atomNum)) {
1862          atomListi.add(atomNum);
1863      }
1864      atomNum =
          substratei.getConservedSubstructure().getAtomNumber(bond.getAtomAt(1));
1865      if (!atomListi.contains(atomNum)) {
1866          atomListi.add(atomNum);
1867      }
1868        }
1869        for (int a = 0; a <
            substratei.getReactingConservedOverlap().getAtomCount(); a++) {
1870      Atom atom =
          getAtomInMolecule(substratei.getReactingConservedOverlap().getAtomAt(a),
          substratei.getConservedSubstructure());
1871      int atomNum =
          substratei.getConservedSubstructure().getAtomNumber(atom);
1872      if (!atomListi.contains(atomNum)) {
1873          atomListi.add(atomNum);
1874      }
1875        }
1876
1877        // second substrate
1878        ArrayList bondListj = new ArrayList();
1879        ArrayList atomListj = new ArrayList();
1880        for (int b = 0; b <
            substratej.getReactingConservedOverlap().getBondCount(); b++) {
1881      Bond bond =
          getBondInMolecule(substratej.getReactingConservedOverlap().getBondAt(b),
          substratej.getConservedSubstructure());
1882      bondListj.add(substratej.getConservedSubstructure().getBondNumber(bond));
1883      int atomNum =
          substratej.getConservedSubstructure().getAtomNumber(bond.getAtomAt(0));
1884      if (!atomListj.contains(atomNum)) {
```

```
1885            atomListj.add(atomNum);
1886        }
1887        atomNum =
            substratej.getConservedSubstructure().getAtomNumber(bond.getAtomAt(1));
1888        if (!atomListj.contains(atomNum)) {
1889            atomListj.add(atomNum);
1890        }
1891          }
1892          for (int a = 0; a <
              substratej.getReactingConservedOverlap().getAtomCount(); a++) {
1893        Atom atom =
            getAtomInMolecule(substratej.getReactingConservedOverlap().getAtomAt(a),
            substratej.getConservedSubstructure());
1894        int atomNum =
            substratej.getConservedSubstructure().getAtomNumber(atom);
1895        if (!atomListj.contains(atomNum)) {
1896            atomListj.add(atomNum);
1897        }
1898          }
1899
1900          // for each bond in rmaplist, see if that pair is in both bondlists
1901          float overlapBondCount = 0;
1902          AtomContainer overlapOfRCOverlap = new AtomContainer();
1903          for (int r = 0; r < rmapList.size(); r++) {
1904        if (bondListi.contains(((RMap)rmapList.get(r)).getId1()) &&
            bondListj.contains(((RMap)rmapList.get(r)).getId2())) {
1905            overlapBondCount++;
1906            Bond bondToAdd =
                substratei.getConservedSubstructure().getBondAt(((RMap)rmapList.get
1907            overlapOfRCOverlap.addBond(bondToAdd);
1908            for (int a = 0; a < 2; a++) {
1909          if (getAtomInMolecule(bondToAdd.getAtomAt(a), overlapOfRCOverlap)
              == null) {
1910              overlapOfRCOverlap.addAtom(bondToAdd.getAtomAt(a));
1911          }
1912            }
1913        } else {
1914        }
1915          }
1916
1917          boolean firstIsSmallerAtoms;
1918          boolean firstIsSmallerBonds;
1919          int minAtoms;
1920          int minBonds;
1921          if (substratei.getReactingSubstructure().getAtomCount() <
              substratej.getReactingSubstructure().getAtomCount()) {
1922        firstIsSmallerAtoms = true;
1923        minAtoms = substratei.getReactingConservedOverlap().getAtomCount();
1924          } else {
1925        firstIsSmallerAtoms = false;
1926        minAtoms = substratej.getReactingConservedOverlap().getAtomCount();
1927          }
```

```
1928        if (substratei.getReactingSubstructure().getBondCount() <
            substratej.getReactingSubstructure().getBondCount()) {
1929      firstIsSmallerBonds = true;
1930      minBonds = substratei.getReactingConservedOverlap().getBondCount();
1931        } else {
1932      firstIsSmallerBonds = false;
1933      minBonds = substratej.getReactingConservedOverlap().getBondCount();
1934        }
1935
1936        // loop through bonds in rmaplist again
1937        // for each atom pair in rmaplist
1938        // see if the atom is present in substrateconservedi and
            substrateconservedj
1939        float overlapAtomCount = 0;
1940        for (int r = 0; r < rmapList.size(); r++) {
1941      Bond bondi =
          substratei.getConservedSubstructure().getBondAt(((RMap)rmapList.get(r)).
1942      Bond bondj =
          substratej.getConservedSubstructure().getBondAt(((RMap)rmapList.get(r)).
1943      if (areAtomsForBondSwitched(bondi, bondj, bondi.getAtoms(),
          bondj.getAtoms(), substratei.getConservedSubstructure(),
          substratej.getConservedSubstructure())) {
1944          // handle first atom pair of bond
1945          Atom ai = bondi.getAtomAt(0);
1946          Atom aj = bondj.getAtomAt(1);
1947          int atomNumi =
              substratei.getConservedSubstructure().getAtomNumber(ai);
1948          int atomNumj =
              substratej.getConservedSubstructure().getAtomNumber(aj);
1949          if ((atomListi.contains(atomNumi)) &&
              (atomListj.contains(atomNumj)) && (getAtomInMolecule(ai,
              substratei.getReactingConservedOverlap()) != null) &&
              (getAtomInMolecule(aj, substratej.getReactingConservedOverlap())
              != null)) {
1950        if (getAtomInMolecule(ai, overlapOfRCOverlap) == null) {
1951            // add atom if it's not in the overlapOfRCOverlap
1952            if (overlapOfRCOverlap.getAtomCount() < minAtoms) {
1953          overlapOfRCOverlap.addAtom(ai);
1954            }
1955        }
1956          }
1957          // handle second atom pair of bond
1958          ai = bondi.getAtomAt(1);
1959          aj = bondj.getAtomAt(0);
1960          atomNumi =
              substratei.getConservedSubstructure().getAtomNumber(ai);
1961          atomNumj =
              substratej.getConservedSubstructure().getAtomNumber(aj);
1962          if ((atomListi.contains(atomNumi)) &&
              (atomListj.contains(atomNumj)) && (getAtomInMolecule(ai,
              substratei.getReactingConservedOverlap()) != null) &&
              (getAtomInMolecule(aj, substratej.getReactingConservedOverlap())
              != null)) {
```

```
1963          if (getAtomInMolecule(ai, overlapOfRCOverlap) == null) {
1964              // add atom if it's not in the overlapOfRCOverlap and if you
                  don't have more atoms than you're supposed to
1965              if (overlapOfRCOverlap.getAtomCount() < minAtoms) {
1966            overlapOfRCOverlap.addAtom(ai);
1967              }
1968          }
1969            }
1970
1971        } else {
1972            // handle first atom pair of bond
1973            Atom ai = bondi.getAtomAt(0);
1974            Atom aj = bondj.getAtomAt(0);
1975            int atomNumi =
                substratei.getConservedSubstructure().getAtomNumber(ai);
1976            int atomNumj =
                substratej.getConservedSubstructure().getAtomNumber(aj);
1977            if ((atomListi.contains(atomNumi)) &&
                (atomListj.contains(atomNumj)) && (getAtomInMolecule(ai,
                substratei.getReactingConservedOverlap()) != null) &&
                (getAtomInMolecule(aj, substratej.getReactingConservedOverlap())
                != null)) {
1978          if (getAtomInMolecule(ai, overlapOfRCOverlap) == null) {
1979              // add atom if it's not in the overlapOfRCOverlap
1980              if (overlapOfRCOverlap.getAtomCount() < minAtoms) {
1981            overlapOfRCOverlap.addAtom(ai);
1982              }
1983          }
1984            }
1985            // handle second atom pair of bond
1986            ai = bondi.getAtomAt(1);
1987            aj = bondj.getAtomAt(1);
1988            atomNumi =
                substratei.getConservedSubstructure().getAtomNumber(ai);
1989            atomNumj =
                substratej.getConservedSubstructure().getAtomNumber(aj);
1990            if ((atomListi.contains(atomNumi)) &&
                (atomListj.contains(atomNumj)) && (getAtomInMolecule(ai,
                substratei.getReactingConservedOverlap()) != null) &&
                (getAtomInMolecule(aj, substratej.getReactingConservedOverlap())
                != null)) {
1991          if (getAtomInMolecule(ai, overlapOfRCOverlap) == null) {
1992              // add atom if it's not in the overlapOfRCOverlap
1993              if (overlapOfRCOverlap.getAtomCount() < minAtoms) {
1994            overlapOfRCOverlap.addAtom(ai);
1995              }
1996          }
1997            }
1998        }
1999
2000          }
2001
2002          // #atoms #bonds #atoms/r+c #atoms/r #bonds/r+c #bonds/r
```

```
2003        System.out.print(overlapOfRCOverlap.getAtomCount() + " " +
            overlapOfRCOverlap.getBondCount() + " ");
2004
2005        // record that number / smallest reacting substructure of pair
2006        if (firstIsSmallerAtoms) {
2007      System.out.print((overlapOfRCOverlap.getAtomCount() + 0.0) /
          substratei.getReactingConservedOverlap().getAtomCount() + " ");
2008      System.out.print((overlapOfRCOverlap.getAtomCount() + 0.0) /
          substratei.getReactingSubstructure().getAtomCount() + " ");
2009        } else {
2010      System.out.print((overlapOfRCOverlap.getAtomCount() + 0.0) /
          substratej.getReactingConservedOverlap().getAtomCount() + " ");
2011      System.out.print((overlapOfRCOverlap.getAtomCount() + 0.0) /
          substratej.getReactingSubstructure().getAtomCount() + " ");
2012        }
2013        if (firstIsSmallerBonds) {
2014      System.out.print(overlapBondCount /
          substratei.getReactingConservedOverlap().getBondCount() + " ");
2015      System.out.println(overlapBondCount /
          substratei.getReactingSubstructure().getBondCount());
2016      // ?? divide by ReactingSubstructure?  ReactingCoservedOverlap?
2017        } else {
2018      System.out.print(overlapBondCount /
          substratej.getReactingConservedOverlap().getBondCount() + " ");
2019      System.out.println(overlapBondCount /
          substratej.getReactingSubstructure().getBondCount());
2020        }
2021
2022    }
2023
2024      }
2025  }
2026
2027    }
2028
2029    /**
2030     *        ---- filterReactionsBySize ----
2031     */
2032    private static void filterReactionsBySize(ArrayList molList) {
2033  RCAtomContainer substrateRcac;
2034  RCAtomContainer productRcac;
2035  ArrayList indicesToRemove = new ArrayList();
2036  int diff;
2037
2038  String HString = new String("H");
2039  for (int i = 0; i < molList.size(); i++) {
2040      productRcac = ((Reaction)molList.get(i)).getProductAtomContainer();
2041      if (productRcac != null) {
2042    substrateRcac = ((Reaction)molList.get(i)).getSubstrateAtomContainer();
2043    // rewriting this to not count hydrogens for hte comparison
2044    Atom[] productAtomsList = productRcac.getLigand().getAtoms();
2045    // count number of non-hydrogen atoms
2046    int numNonHProductAtoms = 0;
```

```
2047      for (int a =0; a < productAtomsList.length; a++) {
2048          if (!HString.equals(productAtomsList[a].getSymbol())) {
2049        numNonHProductAtoms++;
2050          }
2051      }
2052      Atom[] substrateAtomsList = substrateRcac.getLigand().getAtoms();
2053      // count number of non-hydrogen atoms
2054      int numNonHSubstrateAtoms = 0;
2055      for (int a =0; a < substrateAtomsList.length; a++) {
2056          if (!HString.equals(substrateAtomsList[a].getSymbol())) {
2057        numNonHSubstrateAtoms++;
2058          }
2059      }
2060
2061      diff = Math.abs(numNonHProductAtoms - numNonHSubstrateAtoms);
2062      if (diff > MINATOMCOUNT) {
2063          indicesToRemove.add(new Integer(i));
2064      }
2065        }
2066    }
2067    if (indicesToRemove.size() > 0) {
2068        for (int i = indicesToRemove.size()-1; i >= 0; i--) {
2069      molList.remove(((Integer)indicesToRemove.get(i)).intValue());
2070        }
2071    }
2072      }
2073
2074
2075      /**          --- getAtomContainerList ---
2076       * March 9, 2007
2077       */
2078      private static ArrayList getAtomContainerList(ArrayList reactionList) {
2079    ArrayList acArrayList = new ArrayList();
2080    for (int i = 0; i < reactionList.size(); i++) {
2081        acArrayList.add(((Reaction)reactionList.get(i)).getSubstrateAtomContainer
2082    }
2083    return acArrayList;
2084      }
2085
2086      /**
2087       *          ---- Main ----
2088       */
2089      public static void main(String[] args) throws IOException,
          CDKException{
2090
2091    int i;
2092    for (i =0; i < 15; i++) {
2093        blankMolecule.addAtom(new Atom("C"));
2094        if (i > 0) {
2095      blankMolecule.addBond(i, i-1, 1);
2096        }
2097    }
2098    // set up SMILESWriter to output file
```

```
2099   sWriter = new SMILESWriter(initFileWriter(args[1]));
2100   // read in file with list of file names
2101   ArrayList filenameList = getReactionsList(args[0]);
2102
2103   // read in file to get EC numbers
2104   String ECFile = args[0].substring(0,args[0].length() -
       7).concat("info.txt");
2105   ArrayList ECList = getECNumList(ECFile);
2106
2107   // make these files into CDK Molecules
2108   ArrayList molList = getMoleculesForList(filenameList);
2109
2110   // Set EC Numbers for the reactions
2111   for (i = 0; i < molList.size(); i++) {
2112       ((Reaction)molList.get(i)).setECNumber((String)ECList.get(i));
2113   }
2114
2115   // Read in stereochemistry
2116   readStereochemistryForList(molList);
2117
2118   // Filter out cases where substrate and product are really different
       sizes
2119   filterReactionsBySize(molList);
2120
2121   // check for Maximum Common Subgraphs
2122   loopThroughMCS(molList);
2123
2124   // take care of substrate-product MCS
2125   doSubstrateProductMCS(molList);
2126
2127     }
2128 }
2129
2130 //-----------------------------------------------------------------------
2131
2132
2133 /** RCAtomContainer.java
2134  * RCSubstructures.java <superfamily substrates file> <reaction dir>
      <output file>
2135  *
2136  * Ranyee Chiang
2137  * April 5, 2006
2138  *
2139  * stores atom info, reacting substructure, and nonreacting substructure
2140  */
2141
2142 class RCAtomContainer {
2143
2144     public AtomContainer wholeLigand;
2145     public AtomContainer reactingSubstructure;
2146     public AtomContainer nonreactingSubstructure;
2147     // before overlaps are calcalculated - keep substructures as list
2148     public ArrayList conservedSubstructureList;
```

```
2149    public ArrayList unconservedSubstructureList;
2150    // after overlaps are calculated, figure out one conserved substructure
        and keep it
2151    public AtomContainer conservedSubstructure;
2152    public AtomContainer unconservedSubstructure;
2153    public AtomContainer reactingConservedOverlap;
2154    public AtomContainer reactingUnconservedOverlap;
2155    public AtomContainer nonreactingConservedOverlap;
2156    public AtomContainer nonreactingUnconservedOverlap;
2157    public String moleculeName;
2158    private UniversalIsomorphismTester uit;
2159
2160    // Set all atoms' stereoparity to some non value
2161    // Use set and getStereoParity
2162    // Have main function call method with input as coordinates and
        stereoparity and set it
2163
2164    RCAtomContainer() {
2165  uit = new UniversalIsomorphismTester();
2166  this.wholeLigand = new AtomContainer();
2167  conservedSubstructureList = new ArrayList();
2168  unconservedSubstructureList = new ArrayList();
2169  reactingConservedOverlap = null;
2170  reactingUnconservedOverlap = null;
2171  nonreactingConservedOverlap = null;
2172  nonreactingUnconservedOverlap = null;
2173    }
2174
2175    RCAtomContainer(AtomContainer lig) {
2176  uit = new UniversalIsomorphismTester();
2177  this.wholeLigand = lig;
2178  resetStereoparity();
2179
2180  conservedSubstructureList = new ArrayList();
2181  unconservedSubstructureList = new ArrayList();
2182
2183  reactingConservedOverlap = null;
2184  reactingUnconservedOverlap = null;
2185  nonreactingConservedOverlap = null;
2186  nonreactingUnconservedOverlap = null;
2187    }
2188
2189    RCAtomContainer(AtomContainer lig, String name) {
2190  uit = new UniversalIsomorphismTester();
2191  this.wholeLigand = lig;
2192  resetStereoparity();
2193  if (name != null) {
2194      this.moleculeName = name.trim();
2195  } else {
2196      this.moleculeName = name;
2197  }
2198
2199  conservedSubstructureList = new ArrayList();
```

```
2200    unconservedSubstructureList = new ArrayList();
2201    reactingConservedOverlap = null;
2202    reactingUnconservedOverlap = null;
2203    nonreactingConservedOverlap = null;
2204    nonreactingUnconservedOverlap = null;
2205      }
2206
2207
2208
2209      /** resetStereoparity
2210       *
2211       * param non
2212       *
2213       * return non
2214       */
2215      private void resetStereoparity() {
2216    // loop through all atoms
2217    Atom[] atomList = this.wholeLigand.getAtoms();
2218    // set the stereoparity of all atoms to be 0
2219    for (int a = 0; a < atomList.length; a++) {
2220        if (atomList[a].getStereoParity() != 0) {
2221        }
2222        atomList[a].setStereoParity(0);
2223    }
2224      }
2225
2226      public void setStereoparity(int sp, float x, float y, float z) {
2227    // check coordinates of all atoms
2228    Atom[] atomList = this.wholeLigand.getAtoms();
2229    Atom atom;
2230    int count = 0;
2231    boolean found = false;
2232    while ((count < atomList.length) && (!found)) {
2233        // if x y and z match, set stereoparity of that atom
2234        atom = atomList[count];
2235        // if the atom has 3d and not 2d coordinates
2236        if (atom.getX2d() == 0 && atom.getY2d() == 0 && (atom.getX3d() != 0
        || atom.getY3d() != 0 || atom.getZ3d() != 0)) {
2237      if ( (Math.abs(atom.getX3d()-x) < 0.0001) && (Math.abs(atom.getY3d()-y)
      < 0.0001) && (Math.abs(atom.getZ3d()-z) < 0.0001) ) {
2238          atom.setStereoParity(sp);
2239          found = true;
2240      }
2241        } else {
2242      if ( (Math.abs(atom.getX2d()-x) < 0.0001) && (Math.abs(atom.getY2d()-y)
      < 0.0001) ) {
2243          atom.setStereoParity(sp);
2244          found = true;
2245      }
2246        }
2247        count++;
2248    }
2249      }
```

```
2250
2251    /**
2252     *         ---- getAtomInMolecule ----
2253     *
2254     * param Atom atom
2255     * param AtomContainer molecule
2256     *
2257     * return boolean - true if atom with that coordinates, atom type are
        present, false otherwise
2258     */
2259    private static Atom getAtomInMolecule(Atom atom, AtomContainer
        molecule) {
2260  Atom molAtom;
2261  Atom[] atomList = molecule.getAtoms();
2262  // loop through all atoms in molecule
2263  for (int i = 0; i < atomList.length; i++) {
2264      // check to see if this atom has same info as parameter atom
2265      molAtom = atomList[i];
2266      // check to see if it's using 3d or 2d coordinates
2267      if (atom.getX2d() - molAtom.getX2d() == 0 && atom.getY2d() -
        molAtom.getY2d() == 0 && atom.getX2d() == 0 && molAtom.getY2d() == 0)
        {
2268    if (atom.getX3d() == molAtom.getX3d() && atom.getY3d() ==
        molAtom.getY3d() && atom.getZ3d() == molAtom.getZ3d() &&
        atom.getSymbol().equals(molAtom.getSymbol())) {
2269        return molAtom;
2270    }
2271      } else {
2272    if (atom.getX2d() == molAtom.getX2d() && atom.getY2d() ==
        molAtom.getY2d() && atom.getSymbol().equals(molAtom.getSymbol())) {
2273        return molAtom;
2274    }
2275      }
2276
2277  }
2278  return null;
2279    }
2280
2281    /**
2282     * getBondInMolecule
2283     *
2284     * param Bond bond
2285     * param AtomContainer molecule
2286     *
2287     * return Bond in molecule that has same info as bond
2288     */
2289    private Bond getBondInMolecule(Bond bond, AtomContainer molecule) {
2290  Bond molBond;
2291  Bond[] bondList = molecule.getBonds();
2292  // loop through all bonds in molecule
2293  for (int i = 0; i < bondList.length; i++) {
2294      // check to see if this bond has the same info as parameter bond
2295      molBond = bondList[i];
```

```
2296        // molecule has 2d coordinates
2297        if ((bond.get2DCenter().distance(new Point2d(0.0, 0.0)) != 0) &&
            (bond.get3DCenter().distance(new Point3d(0.0, 0.0, 0.0)) == 0)) {
2298      if (bond.get2DCenter().distance(molBond.get2DCenter()) == 0) {
2299          return molBond;
2300      }
2301        } else {    // molecule has 3d coordinates
2302      if (bond.get3DCenter().distance(molBond.get3DCenter()) == 0) {
2303          return molBond;
2304      }
2305        }
2306    }
2307    return null;
2308      }
2309
2310
2311 // substructure is subgraph of molecule
2312      /**
2313       * getNonOverlappingSubstructure
2314       *
2315       * param AtomContainer molecule
2316       * param AtomContainer substructure
2317       *
2318       * returns AtomContainer part of  // copy ligand
2319    // from the copy, remove all atoms (check coordinates) that are present
        in substructure
2320   molecule that is not contained in substructure
2321       */
2322      private AtomContainer getNonOverlappingSubstructure(AtomContainer
          molecule, AtomContainer substructure) {
2323    Atom molAtom;
2324    Bond molBond;
2325    Vector atomVector;
2326    int i;
2327    // check that substructure is substructure of molecule
2328    // copy the molecule
2329    AtomContainer acCopy = (AtomContainer)molecule.clone();
2330
2331    Bond[] substructureBondList = substructure.getBonds();
2332    for (i = 0; i < substructureBondList.length; i++) {
2333        molBond = getBondInMolecule(substructureBondList[i], acCopy);
2334        if (molBond != null) {
2335      // get atoms connected to this bond
2336      atomVector = molBond.getAtomsVector();
2337      acCopy.removeBond((Atom)atomVector.get(0), (Atom)atomVector.get(1));
2338        }
2339    }
2340    // remove disconnected atoms
2341    Atom[] acCopyAtomList = acCopy.getAtoms();
2342    for (i = 0; i < acCopyAtomList.length; i++) {
2343        if (acCopy.getBondCount(acCopyAtomList[i]) == 0) {
2344      acCopy.removeAtom(acCopyAtomList[i]);
2345        }
```

```
2346    }
2347    return acCopy;
2348      }
2349
2350  // set ligand
2351      public void setLigand(AtomContainer lig) {
2352    wholeLigand = lig;
2353    resetStereoparity();
2354      }
2355
2356      // method for helping me debug
2357      // print out each bond in molecule and whether it's conserved/not
          reacting/not
2358      public void printInfoForAllBonds() {
2359
2360    Bond[] bondList = this.wholeLigand.getBonds();
2361    for (int b = 0; b < bondList.length; b++) {
2362        System.out.print(b + " ");
2363        // see if it's reacting or nonreacting
2364        Bond conserved = getBondInMolecule(bondList[b],
            this.conservedSubstructure);
2365        if (conserved == null) {
2366      System.out.print("unconserved ");
2367        } else {
2368      System.out.print("conserved ");
2369        }
2370        Bond unconserved = getBondInMolecule(bondList[b],
            this.unconservedSubstructure);
2371        if (unconserved == null) {
2372      System.out.println("conserved");
2373        } else {
2374      System.out.println("unconserved");
2375        }
2376    }
2377      }
2378
2379
2380  // set nonreacting substructure
2381      public void setNonreactingSubstructure(AtomContainer ss) {
2382
2383    this.nonreactingSubstructure = ss;
2384    this.reactingSubstructure =
        getNonOverlappingSubstructure(this.wholeLigand,
        this.nonreactingSubstructure);
2385      }
2386
2387  // set conserved substructure list
2388  //    when you set conserved substructure, figure out unconserved
      substructure
2389      public void setConservedSubstructureList(ArrayList
          conservedSubstructureList) {
2390    this.conservedSubstructureList = conservedSubstructureList;
2391    // loop through each of the possible conserved substructures
```

```
2392   // find unconservedSubstructure and add it to unconservedSubstructureList
2393   for (int i = 0; i < this.conservedSubstructureList.size(); i++) {
2394       this.unconservedSubstructureList.add(getNonOverlappingSubstructure(this
       (AtomContainer)this.conservedSubstructureList.get(i)));
2395   }
2396     }
2397
2398     // set conserved substructure
2399     public void setConservedSubstructure(int i) {
2400   this.conservedSubstructure =
       (AtomContainer)this.conservedSubstructureList.get(i);
2401   this.unconservedSubstructure =
       (AtomContainer)this.unconservedSubstructureList.get(i);
2402     }
2403
2404     public void setConservedSubstructure(AtomContainer css) {
2405   this.conservedSubstructure = css;
2406   this.unconservedSubstructure =
       getNonOverlappingSubstructure(this.wholeLigand,
       this.conservedSubstructure);
2407     }
2408
2409 // set molecule name
2410     public void setMoleculeName(String name) {
2411   this.moleculeName = name.trim();
2412     }
2413
2414 // get ligand
2415     public AtomContainer getLigand() {
2416   return wholeLigand;
2417     }
2418 // get reacting substructure
2419     public AtomContainer getReactingSubstructure() {
2420   return this.reactingSubstructure;
2421     }
2422 // get nonreacting substructure
2423     public AtomContainer getNonreactingSubstructure() {
2424   return this.nonreactingSubstructure;
2425     }
2426 // get conserved substructure list
2427     public ArrayList getConservedSubstructureList() {
2428   return this.conservedSubstructureList;
2429     }
2430 // get non-conserved substructure list
2431     public ArrayList getUnconservedSubstructureList() {
2432   return this.unconservedSubstructureList;
2433     }
2434
2435 // get conserved substructure
2436     public AtomContainer getConservedSubstructure() {
2437   return this.conservedSubstructure;
2438     }
2439 // get non-conserved substructure
```

```
2440     public AtomContainer getUnconservedSubstructure() {
2441   return this.unconservedSubstructure;
2442     }
2443
2444 // get molecule name
2445     public String getMoleculeName() {
2446   return this.moleculeName;
2447     }
2448
2449 // private get overlap between two substructures
2450
2451     private static void printCoordinates(AtomContainer ac) {
2452   Atom[] atoms = ac.getAtoms();
2453   for (int j = 0; j < atoms.length; j++) {
2454       Atom a = atoms[j];
2455       System.out.print(a.getPoint2d());
2456   }
2457   System.out.println();
2458     }
2459
2460     /**
2461      *         ---- reverseSubstrateAtomOrder ----
2462      *
2463      * takes the atom array of the substrate, reverses that
2464      * this is a hack around the problem of symmetric molecules
2465      * or molecules with multiple copies of same moiety
2466      */
2467     public void reverseSubstrateAtomOrder() {
2468   int count;
2469   if (getLigand() != null) {
2470       // reverse atom array
2471       Atom[] atomList = getLigand().getAtoms();
2472       Atom[] newAtomList = new Atom[atomList.length];
2473       // reverse atomList;
2474       count = 0;
2475       for (int i = atomList.length-1; i >= 0; i--) {
2476     newAtomList[count] = atomList[i];
2477     count = count + 1;
2478       }
2479       getLigand().setAtoms(newAtomList);
2480
2481       // reverse bond array
2482       Bond[] bondList = getLigand().getBonds();
2483       Bond[] newBondList = new Bond[bondList.length];
2484       count = 0;
2485       for (int i = bondList.length-1; i >= 0; i--) {
2486     newBondList[count] = bondList[i];
2487     count = count + 1;
2488       }
2489       getLigand().removeAllBonds();
2490       for (int i = 0; i < newBondList.length; i++) {
2491     getLigand().addBond(newBondList[i]);
2492       }
```

```
2493  }
2494    }
2495
2496    /**
2497     *            ---- calculateAllOverlaps ----
2498     *
2499     * param none
2500     * returns none
2501     *
2502     * calculates overlapping atom containers between reacting-conserved,
       reacting-unconserved, etc
2503     */
2504    public void calculateAllOverlaps() {
2505  calculateReactingConservedOverlap();
2506  calculateReactingUnconservedOverlap();
2507  calculateNonreactingConservedOverlap();
2508  calculateNonreactingUnconservedOverlap();
2509    }
2510
2511    /**
2512     *           ---- getReactingConservedOverlap ----
2513     *
2514     * param none
2515     * returns AtomContainer overlap between reacting and conserved
       substructures
2516     *
2517     * if reactingonservedoverlap is null, calculate it
2518     * otherwise return it
2519     */
2520    public AtomContainer getReactingConservedOverlap() {
2521  if (this.reactingConservedOverlap == null) {
2522      return calculateReactingConservedOverlap();
2523  } else {
2524      return this.reactingConservedOverlap;
2525  }
2526    }
2527
2528    /**
2529     *          ---- calculateReactingConservedOverlap ----
2530     *
2531     * param none
2532     * returns AtomContainer overlap between reacting and conserved
       substructures
2533     *
2534     * Does the actual calculations and returns the result
2535     */
2536    public AtomContainer calculateReactingConservedOverlap() {
2537
2538  // if conserved substructure list is zero
2539  if (getConservedSubstructureList().size() == 0) {
2540      // set conserved substructure to be nothing
2541      setConservedSubstructure(new AtomContainer());
2542      // set reacting conserved to be nothing
```

```
2543        this.reactingConservedOverlap = new AtomContainer();
2544        // return empty atom container
2545        return this.reactingConservedOverlap;
2546    }
2547
2548    // initialize list for possible overlaps(will wind up being the same
        length as conservedSubstructureList)
2549    ArrayList overlapList = new ArrayList();
2550    // for each possible conserved substructure
2551    for (int i = 0; i < getConservedSubstructureList().size(); i++) {
2552        AtomContainer currentConservedSubstructure =
            (AtomContainer)getConservedSubstructureList().get(i);
2553        // check for bond overlap
2554        AtomContainer overlapAC = new AtomContainer();
2555        Bond conservedBond;
2556        Bond[] reactingBondsList = getReactingSubstructure().getBonds();
2557        for (int b = 0; b < reactingBondsList.length; b++) {
2558      conservedBond = getBondInMolecule(reactingBondsList[b],
          currentConservedSubstructure);
2559      if (conservedBond != null) {
2560          Atom[] connectedAtoms = reactingBondsList[b].getAtoms();
2561          if (getAtomInMolecule(connectedAtoms[0], overlapAC) == null) {
2562        overlapAC.addAtom(connectedAtoms[0]);
2563          }
2564          if (getAtomInMolecule(connectedAtoms[1], overlapAC) == null) {
2565        overlapAC.addAtom(connectedAtoms[1]);
2566          }
2567          overlapAC.addBond(reactingBondsList[b]);
2568      }
2569        }
2570        // also check for atom overlap (to handle cases where reacting
            substructure is adjacent to conserved substructure
2571        Atom conservedAtom;
2572        Atom overlapAtom;
2573        Atom[] reactingAtomsList = getReactingSubstructure().getAtoms();
2574        for (int a = 0; a < reactingAtomsList.length; a++) {
2575      overlapAtom = getAtomInMolecule(reactingAtomsList[a], overlapAC);
2576      if (overlapAtom == null) {
2577          conservedAtom = getAtomInMolecule(reactingAtomsList[a],
              currentConservedSubstructure);
2578          if (conservedAtom != null) {
2579        overlapAC.addAtom(reactingAtomsList[a]);
2580          }
2581      }
2582        }
2583        // add the overlapAC to list
2584        overlapList.add(overlapAC);
2585    }
2586
2587    // return the substructure that is the largest
2588    int largestOverlap = 0;
2589    int largestOverlapIndex = 0;
2590    for (int i = 0; i < overlapList.size(); i++) {
```

```
2591          AtomContainer currentOverlap = (AtomContainer)overlapList.get(i);
2592          if (currentOverlap.getBondCount() > largestOverlap) {
2593        largestOverlap = currentOverlap.getBondCount();
2594        largestOverlapIndex = i;
2595          }
2596    }
2597    // if there are no bonds, see which one has most atoms
2598    if (largestOverlap == 0) {
2599          for (int i = 0; i < overlapList.size(); i++) {
2600        AtomContainer currentOverlap = (AtomContainer)overlapList.get(i);
2601        if (currentOverlap.getAtomCount() > largestOverlap) {
2602            largestOverlap = currentOverlap.getAtomCount();
2603            largestOverlapIndex = i;
2604        }
2605          }
2606    }
2607    // set the conserved substructure
2608    setConservedSubstructure(largestOverlapIndex);
2609    this.reactingConservedOverlap =
        (AtomContainer)overlapList.get(largestOverlapIndex);
2610    return (AtomContainer)overlapList.get(largestOverlapIndex);
2611      }
2612
2613      /**
2614       *           ---- getReactingUnonservedOverlap ----
2615       *
2616       * param none
2617       * returns AtomContainer overlap between reacting and conserved
         substructures
2618       *
2619       * if reactingUnconservedoverlap is null, calculate it
2620       * otherwise return it
2621       */
2622      public AtomContainer getReactingUnconservedOverlap() {
2623    if (this.reactingUnconservedOverlap == null) {
2624          return calculateReactingUnconservedOverlap();
2625    } else {
2626          return this.reactingUnconservedOverlap;
2627    }
2628      }
2629
2630      /**
2631       *          ---- calculateReactingUnconservedOverlap ----
2632       *
2633       * param none
2634       * returns AtomContainer overlap between reacting and unconserved
         substructures
2635       *
2636       * Does the actual calculations and returns the result
2637       */
2638      public AtomContainer calculateReactingUnconservedOverlap() {
2639
2640    if (this.unconservedSubstructure == null) {
```

```
2641        calculateReactingConservedOverlap();
2642    }
2643    // initialize new AtomContainer
2644    AtomContainer overlapAC = new AtomContainer();
2645
2646    Bond unconservedBond;
2647    Bond[] reactingBondsList = getReactingSubstructure().getBonds();
2648    for (int b = 0; b < reactingBondsList.length; b++) {
2649        unconservedBond = getBondInMolecule(reactingBondsList[b],
            getUnconservedSubstructure());
2650        if (unconservedBond != null) {
2651      Atom[] connectedAtoms = reactingBondsList[b].getAtoms();
2652      if (getAtomInMolecule(connectedAtoms[0], overlapAC) == null) {
2653          overlapAC.addAtom(connectedAtoms[0]);
2654      }
2655      if (getAtomInMolecule(connectedAtoms[1], overlapAC) == null) {
2656          overlapAC.addAtom(connectedAtoms[1]);
2657      }
2658      overlapAC.addBond(reactingBondsList[b]);
2659        }
2660    }
2661    // also check for atom overlap (to handle cases where reacting
    substructure is adjacent to conserved substructure
2662    Atom unconservedAtom;
2663    Atom overlapAtom;
2664    Atom[] reactingAtomsList = getReactingSubstructure().getAtoms();
2665    for (int a = 0; a < reactingAtomsList.length; a++) {
2666        overlapAtom = getAtomInMolecule(reactingAtomsList[a], overlapAC);
2667        if (overlapAtom == null) {
2668      unconservedAtom = getAtomInMolecule(reactingAtomsList[a],
      getUnconservedSubstructure());
2669      if (unconservedAtom != null) {
2670          overlapAC.addAtom(reactingAtomsList[a]);
2671      }
2672        }
2673    }
2674    this.reactingUnconservedOverlap = overlapAC;
2675    return overlapAC;
2676      }
2677
2678
2679      /**
2680       *          ---- getNonreactingConservedOverlap ----
2681       *
2682       * param none
2683       * returns AtomContainer overlap between nonreacting and conserved
       substructures
2684       *
2685       * if nonreactingonservedoverlap is null, calculate it
2686       * otherwise return it
2687       */
2688      public AtomContainer getNonreactingConservedOverlap() {
2689    if (this.nonreactingConservedOverlap == null) {
```

```
2690        return calculateNonreactingConservedOverlap();
2691    } else {
2692        return this.nonreactingConservedOverlap;
2693    }
2694      }
2695
2696      /**
2697       *          ---- calculateNoneactingConservedOverlap ----
2698       *
2699       * param none
2700       * returns AtomContainer overlap between nonreacting and conserved
           substructures
2701       *
2702       * Does the actual calculations and returns the result
2703       */
2704      public AtomContainer calculateNonreactingConservedOverlap() {
2705
2706    if (this.conservedSubstructure == null) {
2707        calculateReactingConservedOverlap();
2708    }
2709    // initialize new AtomContainer
2710    AtomContainer overlapAC = new AtomContainer();
2711
2712    Bond conservedBond;
2713    Bond[] nonreactingBondsList  = getNonreactingSubstructure().getBonds();
2714    for (int b = 0; b < nonreactingBondsList.length; b++) {
2715        conservedBond = getBondInMolecule(nonreactingBondsList[b],
            getConservedSubstructure());
2716        if (conservedBond != null) {
2717      Atom[] connectedAtoms = nonreactingBondsList[b].getAtoms();
2718      if (getAtomInMolecule(connectedAtoms[0], overlapAC) == null) {
2719          overlapAC.addAtom(connectedAtoms[0]);
2720      }
2721      if (getAtomInMolecule(connectedAtoms[1], overlapAC) == null) {
2722          overlapAC.addAtom(connectedAtoms[1]);
2723      }
2724      overlapAC.addBond(nonreactingBondsList[b]);
2725        }
2726    }
2727    // also check for atom overlap (to handle cases where reacting
        substructure is adjacent to conserved substructure
2728    Atom conservedAtom;
2729    Atom overlapAtom;
2730    Atom[] nonreactingAtomsList  = getNonreactingSubstructure().getAtoms();
2731    for (int a = 0; a < nonreactingAtomsList.length; a++) {
2732        overlapAtom = getAtomInMolecule(nonreactingAtomsList[a], overlapAC);
2733        if (overlapAtom == null) {
2734      conservedAtom = getAtomInMolecule(nonreactingAtomsList[a],
          getConservedSubstructure());
2735      if (conservedAtom != null) {
2736          overlapAC.addAtom(nonreactingAtomsList[a]);
2737      }
2738        }
```

```
2739    }
2740    this.nonreactingConservedOverlap = overlapAC;
2741    return overlapAC;
2742      }
2743
2744      /**
2745       *          ---- getNonreactingUnconservedOverlap ----
2746       *
2747       * param none
2748       * returns AtomContainer overlap between nonreacting and unconserved
         substructures
2749       *
2750       * if nonreactingUnconservedoverlap is null, calculate it
2751       * otherwise return it
2752       */
2753      public AtomContainer getNonreactingUnconservedOverlap() {
2754    if (this.nonreactingUnconservedOverlap == null) {
2755        return calculateNonreactingUnconservedOverlap();
2756    } else {
2757        return this.nonreactingUnconservedOverlap;
2758    }
2759      }
2760
2761      /**
2762       *         ---- calculateNonreactingUnconservedOverlap ----
2763       *
2764       * param none
2765       * returns AtomContainer overlap between nonreacting and unconserved
         substructures
2766       *
2767       * Does the actual calculations and returns the result
2768       */
2769      public AtomContainer calculateNonreactingUnconservedOverlap() {
2770
2771    if (this.unconservedSubstructure == null) {
2772        calculateReactingConservedOverlap();
2773    }
2774
2775    // initialize new AtomContainer
2776    AtomContainer overlapAC = new AtomContainer();
2777
2778    Bond unconservedBond;
2779    Bond[] nonreactingBondsList = getNonreactingSubstructure().getBonds();
2780    for (int b = 0; b < nonreactingBondsList.length; b++) {
2781        unconservedBond = getBondInMolecule(nonreactingBondsList[b],
        getUnconservedSubstructure());
2782        if (unconservedBond != null) {
2783      Atom[] connectedAtoms = nonreactingBondsList[b].getAtoms();
2784      if (getAtomInMolecule(connectedAtoms[0], overlapAC) == null) {
2785          overlapAC.addAtom(connectedAtoms[0]);
2786      }
2787      if (getAtomInMolecule(connectedAtoms[1], overlapAC) == null) {
2788          overlapAC.addAtom(connectedAtoms[1]);
```

```
2789      }
2790      overlapAC.addBond(nonreactingBondsList[b]);
2791        }
2792    }
2793    // also check for atom overlap (to handle cases where reacting
        substructure is adjacent to conserved substructure
2794    Atom unconservedAtom;
2795    Atom overlapAtom;
2796    Atom[] nonreactingAtomsList = getNonreactingSubstructure().getAtoms();
2797    for (int a = 0; a < nonreactingAtomsList.length; a++) {
2798        overlapAtom = getAtomInMolecule(nonreactingAtomsList[a], overlapAC);
2799        if (overlapAtom == null) {
2800      unconservedAtom = getAtomInMolecule(nonreactingAtomsList[a],
          getUnconservedSubstructure());
2801      if (unconservedAtom != null) {
2802          overlapAC.addAtom(nonreactingAtomsList[a]);
2803      }
2804        }
2805    }
2806    this.nonreactingUnconservedOverlap = overlapAC;
2807    return overlapAC;
2808      }
2809
2810  }
2811
2812
2813  //---------------------------------------------------------------------------
2814
2815  /**
2816   * class Reaction
2817   *
2818   * stores filenames for substrate and product
2819   */
2820  class Reaction {
2821
2822      private String substrateFile;
2823      private ArrayList productFileList;
2824      private RCAtomContainer substrateAtomContainer;
2825      private RCAtomContainer productAtomContainer;
2826      private ArrayList productACList;
2827      private int rxnIndex;
2828      private ArrayList substrateProductCorrespondence = new ArrayList();   //
          ArrayList of ArrayLists (pairs of substrate atom/product atom)
2829      private boolean correspondenceFound = false;
2830      private String ECNumber;
2831
2832      Reaction(int i) {
2833    this.substrateFile = null;
2834    this.productFileList = new ArrayList();
2835    this.substrateAtomContainer = null;
2836    this.productAtomContainer = null;
2837    this.productACList = new ArrayList();
2838    this.ECNumber = null;
```

```
2839 rxnIndex = i;
2840   }
2841
2842   Reaction(int i, String sFile) {
2843 this.substrateFile = sFile;
2844 this.productFileList = new ArrayList();
2845 this.substrateAtomContainer = null;
2846 this.productAtomContainer = null;
2847 this.productACList = new ArrayList();
2848 this.ECNumber = null;
2849 rxnIndex = i;
2850   }
2851
2852   Reaction(int i, String sFile, String pFile) {
2853 this.substrateFile = sFile;
2854 this.productFileList = new ArrayList();
2855 this.productFileList.add(pFile);
2856 this.substrateAtomContainer = null;
2857 this.productAtomContainer = null;
2858 this.productACList = new ArrayList();
2859 this.ECNumber = null;
2860 rxnIndex = i;
2861   }
2862
2863   public int getReactionIndex() {
2864 return this.rxnIndex;
2865   }
2866
2867   public String getSubstrateFile() {
2868 return this.substrateFile;
2869   }
2870
2871   public ArrayList getProductFileList() {
2872 return this.productFileList;
2873   }
2874
2875   public RCAtomContainer getProductAtomContainer() {
2876 return this.productAtomContainer;
2877   }
2878
2879   public ArrayList getProductACList () {
2880 return this.productACList;
2881   }
2882
2883   public RCAtomContainer getSubstrateAtomContainer() {
2884 return this.substrateAtomContainer;
2885   }
2886
2887   public String getECNumber() {
2888 return this.ECNumber;
2889   }
2890
2891   public boolean isRacemase() {
```

```
2892    return (this.ECNumber.startsWith("5.1.") ||
        this.ECNumber.startsWith("5.5.") || this.ECNumber.startsWith("5.99."));
2893      }
2894
2895      public void setECNumber(String ec) {
2896    this.ECNumber = ec;
2897      }
2898
2899      public void addProductFile(String pFile) {
2900    this.productFileList.add(pFile);
2901      }
2902
2903      public void setSubstrateAtomContainer(RCAtomContainer ac) {
2904    this.substrateAtomContainer = ac;
2905      }
2906
2907      public void addProductAtomContainer(RCAtomContainer rac) {
2908    this.productACList.add(new AtomContainer(rac.getLigand()));
2909    if (this.productAtomContainer == null) {
2910        this.productAtomContainer = rac;
2911    } else {
2912        // this is code that just adds the product without checking for
            overlaps - if they're listed as separate molecules, there should be
            any overlaps
2913        // try out:
2914        this.productAtomContainer.getLigand().add(rac.getLigand());
2915        // change molecule name of product
2916        this.productAtomContainer.setMoleculeName(this.productAtomContainer.
            getMoleculeName()
            + " + " + rac.getMoleculeName());
2917        // this may not add the bonds so test it out by printing out number
            of atoms and bonds before and after
2918        // if it still doesn't work, try:
2919        // looping through bonds and adding bonds
2920        // if that doesn't work, loop through bonds, check if bond's atoms
            are already there and add if not
2921
2922    }
2923      }
2924      /**
2925       *          ---- getAtomInMolecule ----
2926       *
2927       * param Atom atom
2928       * param AtomContainer molecule
2929       *
2930       * return boolean - true if atom with that coordinates, atom type are
           present, false otherwise
2931       */
2932      private static Atom getAtomInMolecule(Atom atom, AtomContainer
          molecule) {
2933    Atom molAtom;
2934    Atom[] atomList = molecule.getAtoms();
2935    // loop through all atoms in molecule
2936    for (int i = 0; i < atomList.length; i++) {
```

```
2937        // check to see if this atom has same info as parameter atom
2938        molAtom = atomList[i];
2939        // check to see if it's using 3d or 2d coordinates
2940        if (atom.getX2d() - molAtom.getX2d() == 0 && atom.getY2d() -
            molAtom.getY2d() == 0 && atom.getX2d() == 0 && molAtom.getY2d() == 0)
            {
2941      if (atom.getX3d() == molAtom.getX3d() && atom.getY3d() ==
          molAtom.getY3d() && atom.getZ3d() == molAtom.getZ3d() &&
          atom.getSymbol().equals(molAtom.getSymbol())) {
2942          return molAtom;
2943      }
2944        } else {
2945      if (atom.getX2d() == molAtom.getX2d() && atom.getY2d() ==
          molAtom.getY2d() && atom.getSymbol().equals(molAtom.getSymbol())) {
2946          return molAtom;
2947      }
2948        }
2949
2950    }
2951    return null;
2952      }
2953
2954      /**
2955       *          ---- getBondInMolecule ----
2956       *
2957       * param Bond bond
2958       * param AtomContainer molecule
2959       *
2960       * return Bond in molecule that has same info as bond
2961       */
2962      private static Bond getBondInMolecule(Bond bond, AtomContainer
          molecule) {
2963    Bond molBond;
2964    Bond[] bondList = molecule.getBonds();
2965    Point2d zeropoint = new Point2d(0.0, 0.0);
2966    // loop through all bonds in molecule
2967    for (int i = 0; i < bondList.length; i++) {
2968        // check to see if this bond has the same info as parameter bond
2969        molBond = bondList[i];
2970        if ( (bond.get2DCenter().distance(zeropoint) == 0) &&
            (molBond.get2DCenter().distance(zeropoint) == 0) ) {
2971      if (bond.get3DCenter().distance(molBond.get3DCenter()) == 0) {
2972          return molBond;
2973      }
2974        }
2975        else {
2976      if (bond.get2DCenter().distance(molBond.get2DCenter()) == 0) {
2977          return molBond;
2978      }
2979        }
2980    }
2981    return null;
2982      }
```

```
2983
2984
2985      private boolean areBondsEquivalent(Bond b1, Bond b2) {
2986    Point2d zeropoint = new Point2d(0.0, 0.0);
2987    if ( (b1.get2DCenter().distance(zeropoint) == 0) &&
        (b2.get2DCenter().distance(zeropoint) == 0) ) {
2988        if (b1.get3DCenter().distance(b2.get3DCenter()) == 0) {
2989      return true;
2990        }
2991    } else {
2992        if (b1.get2DCenter().distance(b2.get2DCenter()) == 0) {
2993      return true;
2994        }
2995    }
2996    return false;
2997      }
2998
2999      // given a list of atoms, checks to see if atom is in list
3000      // checks coordinates and stereoparity
3001      private boolean atomInList(Atom atom, ArrayList aList) {
3002    Atom tempAtom;
3003    for (int a = 0; a < aList.size(); a++) {
3004        tempAtom = (Atom)aList.get(a);
3005        if ( (tempAtom.getX2d() == atom.getX2d()) && (tempAtom.getY2d() ==
            atom.getY2d()) && (tempAtom.getStereoParity() ==
            atom.getStereoParity()) ) {
3006      return true;
3007        }
3008    }
3009    return false;
3010      }
3011
3012      /**       --- getBondString ---
3013       * param Bond
3014       * returns String - alphebetized string with atoms on either side of
           bond with bond order in front
3015       */
3016      private String getBondString(Bond b) {
3017    String bondString = new String();
3018    //add in bond order
3019    bondString = bondString.concat(String.valueOf(b.getOrder()));
3020    // get atoms, sort them, add them to bondString
3021    Atom[] atoms = b.getAtoms();
3022    String a0 = atoms[0].getSymbol();
3023    String a1 = atoms[1].getSymbol();
3024    if (a0.compareTo(a1) > 0) {
3025        bondString = bondString.concat(a1);
3026        bondString = bondString.concat(a0);
3027    } else {
3028        bondString = bondString.concat(a0);
3029        bondString = bondString.concat(a1);
3030    }
3031    return bondString;
```

```
3032     }
3033
3034     /**           --- sortStringList ---
3035      * param ArrayList - of Strings
3036      * returns ArrayList - alphabetized list of strings
3037      */
3038     private String[] sortStringArray(String[] stringArray) {
3039   String str1;
3040   String str2;
3041   String tmpStr;
3042
3043   for (int i = 0; i < stringArray.length-1; i++) {
3044       for (int j = i+1; j < stringArray.length; j++) {
3045     str1 = stringArray[i];
3046     str2 = stringArray[j];
3047
3048     if (str1.compareTo(str2) > 0) {
3049         // swap
3050         stringArray[i] = str2;
3051         stringArray[j] = str1;
3052     }
3053       }
3054   }
3055   return stringArray;
3056     }
3057
3058     /**          --- getBondArrayString ---
3059      * param Bond[] - of Bonds
3060      * returns String - alphabetized String of all bonds concatenated
3061      */
3062     private String getBondArrayString(Bond[] bondList) {
3063
3064   Bond bond;
3065   String bondString;
3066
3067   String[] stringArray = new String[bondList.length];
3068   String wholeString = new String();
3069   for (int b = 0; b < bondList.length; b++) {
3070       bond = bondList[0];
3071       bondString = getBondString(bond);
3072       stringArray[b] = bondString;
3073   }
3074   stringArray = sortStringArray(stringArray);
3075   for (int i = 0; i < stringArray.length; i++) {
3076       wholeString = wholeString.concat(stringArray[i]);
3077   }
3078   return wholeString;
3079     }
3080
3081     private Bond[]
         getBondsConnectedToAtomNotIncludingCurrentBond(AtomContainer ac, Atom
         a, Bond b) {
3082   Bond[] allConnectedBonds = ac.getConnectedBonds(a);
```

```
3083    if (allConnectedBonds.length > 1) {
3084        Bond[] newConnectedBonds = new Bond[allConnectedBonds.length-1];
3085        int bondListLength = 0;
3086        if (newConnectedBonds.length == 0) {
3087      return new Bond[0];
3088        }
3089        for (int i = 0; i < allConnectedBonds.length; i++) {
3090      if (!areBondsEquivalent(allConnectedBonds[i], b)) {
3091          if (bondListLength < newConnectedBonds.length) {
3092        newConnectedBonds[bondListLength] = allConnectedBonds[i];
3093        bondListLength++;
3094          }
3095      }
3096        }
3097        return newConnectedBonds;
3098    } else {
3099        return new Bond[0];
3100    }
3101
3102      }
3103
3104      /**       --- areAtomsForBondSwitched ---
3105       * param Bond first bond
3106       * param Bond second bond
3107       * param AtomContainer substrateSubstructure - common substructure,
       uses substrate's coordinates
3108       * param AtomContainer productSubstructure - common substructure, uses
       product's coordinates
3109       * returns boolean false if a0-b0 and a1-b1, true if a0-b1 and a1-b0
3110       *
3111       * determines whether atom order needs to be switched to get
3112       * proper correspondence
3113       */
3114      private boolean areAtomsForBondSwitched(Bond substrateBond, Bond
      productBond, Atom[] substrateAtoms, Atom[] productAtoms, AtomContainer
      substrateSubstructure, AtomContainer productSubstructure) {
3115
3116    Bond[] substrateConnectedBonds0;
3117    Bond[] substrateConnectedBonds1;
3118    Bond[] productConnectedBonds0;
3119    Bond[] productConnectedBonds1;
3120    // solution #1 first do obvious check when symbols are different
3121    if (substrateAtoms[0].getSymbol() != substrateAtoms[1].getSymbol()) {
3122        correspondenceFound = true;
3123        if (substrateAtoms[0].getSymbol() == productAtoms[0].getSymbol() &&
        substrateAtoms[1].getSymbol() == productAtoms[1].getSymbol()) { // if
        atom1 in substrate is same as atom1 in product
3124      return false;
3125        } else {
3126      return true;
3127        }
3128    } else { // find adjacent bonds in substructures (substrate and product
    coordinates)
```

```
3129        // save atoms of substructure
3130        Atom productAtom0 = getAtomInMolecule(productAtoms[0],
            productSubstructure);
3131        Atom productAtom1 = getAtomInMolecule(productAtoms[1],
            productSubstructure);
3132        Atom substrateAtom0 = getAtomInMolecule(substrateAtoms[0],
            substrateSubstructure);
3133        Atom substrateAtom1 = getAtomInMolecule(substrateAtoms[1],
            substrateSubstructure);
3134
3135        if (productAtom1 == null) {
3136      System.out.println("productatom0 null");
3137      System.out.println(productAtoms.length);
3138        }
3139        productConnectedBonds0 =
            getBondsConnectedToAtomNotIncludingCurrentBond(productSubstructure,
            productAtom0, productBond);
3140        productConnectedBonds1 =
            getBondsConnectedToAtomNotIncludingCurrentBond(productSubstructure,
            productAtom1, productBond);
3141        substrateConnectedBonds0 =
            getBondsConnectedToAtomNotIncludingCurrentBond(substrateSubstructure,
            substrateAtom0, substrateBond);
3142        substrateConnectedBonds1 =
            getBondsConnectedToAtomNotIncludingCurrentBond(substrateSubstructure,
            substrateAtom1, substrateBond);
3143
3144        // solution #2 if number of connected bonds on each atom is different
3145        if (substrateConnectedBonds0.length !=
            substrateConnectedBonds1.length) {
3146      correspondenceFound = true;
3147      if ( (substrateConnectedBonds0.length == productConnectedBonds0.length)
          && (substrateConnectedBonds1.length == productConnectedBonds1.length))
          {
3148          return false;
3149      } else {
3150          return true;
3151      }
3152        } else {
3153      // check to see if bond identities are the same
3154      String substrateS0 = getBondArrayString(substrateConnectedBonds0);
3155      String substrateS1 = getBondArrayString(substrateConnectedBonds1);
3156      String productS0 = getBondArrayString(productConnectedBonds0);
3157      String productS1 = getBondArrayString(productConnectedBonds1);
3158
3159      if (substrateS0.compareTo(substrateS1) != 0) {
3160          correspondenceFound = true;
3161          if ( (substrateS0.compareTo(productS0) == 0) &&
              (substrateS1.compareTo(productS1) == 0) ) {
3162        return false;
3163          } else {
3164        return true;
3165          }
```

```
3166      } else {
3167          correspondenceFound = false;
3168          return false;
3169      }
3170        }
3171        // if they are not switched, # connected bonds should be the same,
            type of bonds should be the same
3172        // may have to extend out beyond first bond (recurse this function?)
3173    }
3174      }
3175
3176      /**          --- atomInCorrespondenceList ---
3177       *
3178       * determines whether a substrate atom is already in correspondence
           list
3179       *
3180       */
3181      private boolean atomInCorrespondenceList(Atom a) {
3182    //return false;
3183    // substrateProductCorrespondence is an ArrayList of ArrayLists (pairs)
3184    // loop through ArrayLists
3185    for (int i = 0; i < substrateProductCorrespondence.size(); i++) {
3186        // get first Atom in each ArrayList
3187        ArrayList correspondence =
            (ArrayList)substrateProductCorrespondence.get(i);
3188        // if that Atom a is present, return true
3189        Atom currentA = (Atom)correspondence.get(0);
3190        if (a == currentA) {
3191      return true;
3192        }
3193    }
3194    // if you make it out of the loop, return false
3195    return false;
3196      }
3197
3198      /**          ---- subtractStereoparityChanges ----
3199       * param AtomContainer substructure - the substructure found as being
           in common
3200       * param List mapList - maps atoms in substrate to atoms in product
3201       * param AtomContainer productSubstructure - the substructure found as
           being in common but with the product coordinates
3202       *
3203       * takes out parts of substructure that change steroparity
3204       * also suen4 bian4 (conveniently at the same time) stores
           substrate-product atom correspondences for commonsubstructure
3205       * this will have a few extra atoms that were later removed b/c of
           stereoparity changes
3206       * but that won't matter because I will be looking up substrate
           nonreacting substructure atoms to
3207       * get the corresponding product atom
3208       */
3209      public void subtractStereoparityChanges(AtomContainer substructure,
          ArrayList mapList, AtomContainer productSubstructure) {
```

```
3210  ArrayList fragment;
3211  RMap rMap;
3212  ArrayList atomsChangedList = new ArrayList();
3213  Bond substrateBond;
3214  Bond productBond;
3215  Atom[] substrateAtoms;
3216  Atom[] productAtoms;
3217  AtomContainer substrate = getSubstrateAtomContainer().getLigand();
3218  AtomContainer product = getProductAtomContainer().getLigand();
3219
3220  // loop through fragments
3221  for (int i = 0; i < mapList.size(); i++) {
3222      fragment = (ArrayList)mapList.get(i);
3223      // loop through RMaps for each atom in fragment
3224      for (int j = 0; j < fragment.size(); j++) {
3225    rMap = (RMap)fragment.get(j);
3226    try {
3227        substrateBond = substrate.getBondAt(rMap.getId1());
3228        productBond = product.getBondAt(rMap.getId2());
3229    } catch (ArrayIndexOutOfBoundsException e) {
3230        System.out.println("arrayindexoutofboundseception");
3231        continue;
3232    }
3233    // will have to get atoms of bonds (loop through them) - I don't know
      which atom matches to which
3234    substrateAtoms = substrateBond.getAtoms();
3235    productAtoms = productBond.getAtoms();
3236
3237    if (!areAtomsForBondSwitched(substrateBond, productBond,
      substrateAtoms, productAtoms, substructure, productSubstructure)) {
3238        // check to see if stereoparity changes
3239        if ((substrateAtoms[0].getSymbol() == productAtoms[0].getSymbol())
          && (substrateAtoms[1].getSymbol() == productAtoms[1].getSymbol()))
          {
3240      if (!(substrateAtoms[0].getStereoParity() == 0
        ||(substrateAtoms[0].getStereoParity() == 3) ||
        (productAtoms[0].getStereoParity() == 0) ||
        (productAtoms[0].getStereoParity() == 3))) {
3241          if (substrateAtoms[0].getStereoParity() !=
            productAtoms[0].getStereoParity()) {
3242        if (!atomInList(substrateAtoms[0], atomsChangedList)) {
3243            atomsChangedList.add(substrateAtoms[0]);
3244        }
3245          }
3246      }
3247
3248      if (!(substrateAtoms[1].getStereoParity() == 0 ||
        (substrateAtoms[1].getStereoParity() == 3) ||
        (productAtoms[1].getStereoParity() == 0) ||
        (productAtoms[1].getStereoParity() == 3))) {
3249          if (substrateAtoms[1].getStereoParity() !=
            productAtoms[1].getStereoParity()) {
3250        if (!atomInList(substrateAtoms[1], atomsChangedList)) {
```

```
3251                atomsChangedList.add(substrateAtoms[1]);
3252            }
3253              }
3254          }
3255          // store substrate atom correspondence
3256          ArrayList correspondence = new ArrayList();
3257          if (!atomInCorrespondenceList(substrateAtoms[0])) {
3258              correspondence.add(substrateAtoms[0]);
                  correspondence.add(productAtoms[0]);
                  substrateProductCorrespondence.add(correspondence);
3259          } else {
3260              // if this one found the correspondence, use this one
3261              if (correspondenceFound &&
                  (!atomsHaveIdenticalCoordinates(productAtoms[0],
                  getCorrespondingProductAtom(substrateAtoms[0])))) {
3262            replaceCorrespondingProductAtom(substrateAtoms[0],
                productAtoms[0]);
3263            // else if it didn't find the correspondence, don't replace
3264              }
3265          }
3266          correspondence = new ArrayList();
3267          if (!atomInCorrespondenceList(substrateAtoms[1])) {
3268              correspondence.add(substrateAtoms[1]);
                  correspondence.add(productAtoms[1]);
                  substrateProductCorrespondence.add(correspondence);
3269          } else {
3270              // if this one found the correspondence, use this one
3271              if (correspondenceFound &&
                  (!atomsHaveIdenticalCoordinates(productAtoms[1],
                  getCorrespondingProductAtom(substrateAtoms[1])))) {
3272            replaceCorrespondingProductAtom(substrateAtoms[1],
                productAtoms[1]);
3273            // else if it didn't find the correspondence, don't replace
3274              }
3275          }
3276            } else {
3277            }
3278
3279      } else {  // when atom1 in substrate is same as atom2 in product
3280            if ((substrateAtoms[0].getSymbol() == productAtoms[1].getSymbol())
                && (substrateAtoms[1].getSymbol() == productAtoms[0].getSymbol()))
                {
3281          if (!(substrateAtoms[0].getStereoParity() == 0 ||
              (substrateAtoms[0].getStereoParity() == 3) ||
              (productAtoms[1].getStereoParity() == 0) ||
              (productAtoms[1].getStereoParity() == 3))) {
3282              if (substrateAtoms[0].getStereoParity() !=
                  productAtoms[1].getStereoParity()) {
3283            if (!atomInList(substrateAtoms[0], atomsChangedList)) {
3284                atomsChangedList.add(substrateAtoms[0]);
3285            }
3286              }
3287          }
```

```
3288        if (!(substrateAtoms[1].getStereoParity() == 0 ||
            (substrateAtoms[1].getStereoParity() == 3) ||
            (productAtoms[0].getStereoParity() == 0) ||
            (productAtoms[0].getStereoParity() == 3))) {
3289            if (substrateAtoms[1].getStereoParity() !=
                productAtoms[0].getStereoParity()) {
3290          if (!atomInList(substrateAtoms[1], atomsChangedList)) {
3291              atomsChangedList.add(substrateAtoms[1]);
3292          }
3293            }
3294        }
3295        // store substrate atom correspondence
3296        ArrayList correspondence = new ArrayList();
3297        if (!atomInCorrespondenceList(substrateAtoms[0])) {
3298            correspondence.add(substrateAtoms[0]);
                correspondence.add(productAtoms[1]);
                substrateProductCorrespondence.add(correspondence);
3299        } else {
3300            // if this one found the correspondence, use this one
3301            if (correspondenceFound &&
                (!atomsHaveIdenticalCoordinates(productAtoms[1],
                getCorrespondingProductAtom(substrateAtoms[0])))) {
3302          replaceCorrespondingProductAtom(substrateAtoms[0],
              productAtoms[1]);
3303          // else if it didn't find the correspondence, don't replace
3304            }
3305        }
3306        correspondence = new ArrayList();
3307        if (!atomInCorrespondenceList(substrateAtoms[1])) {
3308            correspondence.add(substrateAtoms[1]);
                correspondence.add(productAtoms[0]);
                substrateProductCorrespondence.add(correspondence);
3309        } else {
3310            // if this one found the correspondence, use this one
3311            if (correspondenceFound &&
                atomsHaveIdenticalCoordinates(productAtoms[0],
                getCorrespondingProductAtom(substrateAtoms[1]))) {
3312          replaceCorrespondingProductAtom(substrateAtoms[1],
              productAtoms[0]);
3313          // else if it didn't find the correspondence, don't replace
3314            }
3315        }
3316          } else {
3317
3318          }
3319    }
3320        }
3321  }
3322
3323  Atom substructureAtom;
3324  Atom[] connectedAtoms;
3325  Atom tempAtom;
3326  for (int a=0; a < atomsChangedList.size(); a++) {
```

```
3327        tempAtom = (Atom)atomsChangedList.get(a);
3328        // check if atom with changed stereoparity is present in substructure
3329        substructureAtom = getAtomInMolecule(tempAtom, substructure);
3330        if (substructureAtom != null) {
3331      // if so, remove it and its connected bonds
3332      connectedAtoms = substructure.getConnectedAtoms(substructureAtom);
3333      for (int ca = 0; ca < connectedAtoms.length; ca++) {
3334          substructure.removeBond(substructureAtom, connectedAtoms[ca]);
3335      }
3336      substructure.removeAtom(substructureAtom);
3337        }
3338    }
3339
3340    // remove any unconnected atoms
3341    Atom[] remainingAtoms = substructure.getAtoms();
3342    for (int i = 0; i < remainingAtoms.length; i++) {
3343        if (substructure.getBondCount(remainingAtoms[i]) == 0) {
3344      substructure.removeAtom(remainingAtoms[i]);
3345        }
3346    }
3347
3348      }
3349
3350      private boolean atomsHaveIdenticalCoordinates(Atom a1, Atom a2) {
3351    if (a1.getX2d() - a2.getX2d() == 0 && a1.getY2d() - a2.getY2d() == 0 &&
        a1.getX2d() == 0 && a2.getY2d() == 0) {
3352        if (a1.getX3d() == a2.getX3d() && a1.getY3d() == a2.getY3d() &&
            a1.getZ3d() == a2.getZ3d() && a1.getSymbol().equals(a2.getSymbol()))
            {
3353      return true;
3354        }
3355    } else {
3356        if (a1.getX2d() == a2.getX2d() && a1.getY2d() == a2.getY2d() &&
            a1.getSymbol().equals(a2.getSymbol())) {
3357      return true;
3358        }
3359    }
3360    return false;
3361      }
3362
3363      /**       --- replaceCorrespondingProductAtom ----
3364       * replaces the product atom for substrate atom with the new one
3365       */
3366
3367      private void replaceCorrespondingProductAtom(Atom substrateAtom, Atom
          productAtom) {
3368    // loop through ArrayLists (pairs) in substrateProductCorrespondence
3369    for (int i = 0; i < substrateProductCorrespondence.size(); i++) {
3370        // if substrateAtom is equal to first element of ArrayList
3371        ArrayList spPair = (ArrayList)substrateProductCorrespondence.get(i);
3372        if ((Atom)spPair.get(0) == substrateAtom) {
3373      // replace the second element of the list
3374      spPair.set(1, productAtom);
```

```
3375          }
3376      }
3377        }
3378
3379        /**          ---- getCorrespondingProductAtom -----
3380         * param Atom substrateAtom
3381         * return Atom corresponding product atom, null if not found
3382         *
3383         * using substrateProductCorrespondence (ArrayList), finds product atom
         which
3384         * corresponds to the substrate's atom
3385         */
3386        private Atom getCorrespondingProductAtom(Atom substrateAtom) {
3387      // loop through ArrayLists (pairs) in substrateProductCorrespondence
3388      for (int i = 0; i < substrateProductCorrespondence.size(); i++) {
3389          ArrayList spPair = (ArrayList)substrateProductCorrespondence.get(i);
3390          // if substrateAtom is equal to first element of ArrayList
3391          if ((Atom)spPair.get(0) == substrateAtom) {
3392        // return the second element
3393        return (Atom)spPair.get(1);
3394          }
3395      }
3396      return null;
3397        }
3398
3399        /**
3400         *          ---- addAtomToAC ----
3401         *
3402         * param AtomContainer baseAC
3403         * param Atom a
3404         *
3405         * adds bonds and molecules of addAC to baseAC
3406         */
3407        private static void addAtomToAC(Atom a, AtomContainer baseAC) {
3408      // make sure atom is not already in ac, which means that there should be
      no bonds containing this atom
3409      if (getAtomInMolecule(a, baseAC) == null) {
3410          // add atom to atom container
3411          baseAC.addAtom(a);
3412      }
3413        }
3414
3415        /**          ------ addAtomsOfChangedBonds() ----
3416         * param ArrayList mapList - mapList for substrate substructure overlap
3417         * XXparam AtomContainer productCommonSubstructure - product
         substructure overlap
3418         * returns none
3419         *
3420         * for new bonds made in the product (also checks for bonds broken, but
         may be unnecessary),
3421         * adds adjacent atoms to reacting substructure
3422         */
```

```
3423     public void addAtomsOfChangedBonds(ArrayList mapList) throws
         IOException, CDKException{
3424   Atom substrateAtom;
3425   Atom productAtom;
3426   AtomContainer nonreactingSubstructure =
       getSubstrateAtomContainer().getNonreactingSubstructure();
3427   AtomContainer reactingSubstructure =
       getSubstrateAtomContainer().getReactingSubstructure();
3428   AtomContainer substrate = getSubstrateAtomContainer().getLigand();
3429   AtomContainer product = getProductAtomContainer().getLigand();
3430
3431   if (nonreactingSubstructure != null) {
3432       // loop through atoms of nonreacting substructure
3433       Atom[] nonreactingAtoms = nonreactingSubstructure.getAtoms();
3434       for (int i = 0; i < nonreactingAtoms.length; i++) {
3435     // get atom in substrate and atom in product
3436     substrateAtom = getAtomInMolecule(nonreactingAtoms[i], substrate);
3437     productAtom = getCorrespondingProductAtom(substrateAtom);
3438     if (productAtom != null) {
3439
3440         // if bondcount is different
3441         if (substrate.getBondCount(substrateAtom) !=
         product.getBondCount(productAtom)) {
3442       // if the atom is NOT in the reacting substructure
3443       if (getAtomInMolecule(substrateAtom, reactingSubstructure) == null) {
3444           // add the atom to reacting substructure (don't need to remove it
           from nonreacting substructure)
3445           addAtomToAC(substrateAtom, reactingSubstructure);
3446
3447       } else {
3448       }
3449         }
3450     } else {
3451     }
3452       }
3453
3454   } else {}
3455     }
3456
3457 }
3458
3459
3460
3461 //----------------------------------------------------------------------
3462
3463 /**
3464  * class Overlaps
3465  *
3466  * stores AtomContainer and RMap for pairwise overlaps calculation
3467  */
3468 class Overlaps {
3469
3470     private AtomContainer overlap;
```

```
3471     private ArrayList rMaps;
3472
3473     Overlaps() {
3474   this.overlap = null;
3475   this.rMaps = new ArrayList();
3476     }
3477
3478     Overlaps(AtomContainer ac, ArrayList al) {
3479   this.overlap = ac;
3480   this.rMaps = al;
3481     }
3482
3483     public AtomContainer getOverlap() {
3484   return this.overlap;
3485     }
3486
3487     public ArrayList getRMaps() {
3488   return this.rMaps;
3489     }
3490
3491     public void addRMap(ArrayList rm) {
3492   this.rMaps.add(rm);
3493     }
3494
3495     public void setRMaps(ArrayList ac) {
3496   this.rMaps = ac;
3497     }
3498
3499     public void setOverlap(AtomContainer o) {
3500   this.overlap = o;
3501     }
3502
3503 }
```

Publishing Agreement

It is the policy of the University to encourage the distribution of all theses and dissertations. Copies of all UCSF theses and dissertations will be routed to the library via the Graduate Division. The library will make all theses and dissertations accessible to the public and will preserve these to the best of their abilities, in perpetuity.

Please sign the following statement:

I hereby grant permission to the Graduate Division of the University of California, San Francisco to release copies of my thesis or dissertation to the Campus Library to provide access and preservation, in whole or in part, in perpetuity.

Author Signature

September 5, 2008

Date