

Merging of gene predictions

Sebastian R. Spiegler
Wellcome Trust Sanger Institute
Wellcome Trust Genome Campus
Hinxton, Cambridge CB10 1SA,
England.
Email: ss7@sanger.ac.uk

September 16, 2003

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 4 |
| 1.1 | Abstract | 4 |
| 1.2 | Acknowledgements | 4 |
| 1.3 | The Sanger Insitute | 4 |
| 2 | Computational gene prediction | 6 |
| 2.1 | Gene structure | 6 |
| 2.2 | Computational Gene Prediction | 6 |
| 2.2.1 | Similarity based Approach to Gene Prediction | 7 |
| 2.2.2 | Statistical Approach to Gene Prediction | 8 |
| 2.2.3 | Gene prediction concepts | 9 |
| 3 | Benchmarking gene predictions | 11 |
| 3.1 | Measuring the prediction accuracy | 11 |
| 3.2 | Using the prediction accuracy for merging of gene predictions | 12 |
| 4 | Software Development | 15 |
| 4.1 | Object-Oriented Software Development (OOSD) Metholodogy | 15 |
| 4.2 | Appliance of OOSD in the Circular Model for Software Development | 15 |
| 5 | Development of <i>GFMerge</i> | 17 |
| 5.1 | Project description | 17 |
| 5.2 | Software development process according to the Circular Model | 18 |
| 5.2.1 | Cycle 1 | 18 |
| 5.2.2 | Cycle 2 | 19 |
| 6 | Software architecture and design of <i>GFMerge</i> | 23 |
| 6.1 | Unified Modeling Language (UML) | 23 |
| 6.1.1 | Class diagrams | 23 |
| 6.1.2 | Sequence diagrams | 24 |
| 6.2 | Data structures in <i>GFMerge</i> | 24 |
| 6.2.1 | class GenePredicter | 24 |
| 6.2.2 | class Prediction | 24 |
| 6.2.3 | abstract class <i>GFMergeFeature</i> | 24 |

| | | |
|----------|--|-----------|
| 6.2.4 | GFMergeRegion | 26 |
| 6.3 | Program flow of GFMerge | 26 |
| 6.3.1 | Main class GFMerge | 26 |
| 6.3.2 | class GFMerge_Analysis - preprocessing | 27 |
| 6.3.3 | class GFMerge_Analysis - computing | 27 |
| 7 | Selected algorithms of <i>GFMerge</i> | 28 |
| 7.1 | Clustering the sequence | 28 |
| 7.2 | High Scoring Path | 30 |
| 7.2.1 | Description | 30 |
| 7.2.2 | Implementation and optimization | 30 |
| 8 | Documentation of <i>GFMerge</i> | 33 |
| 8.1 | reference manual | 33 |
| 8.2 | Merging process | 34 |
| 8.2.1 | Introduction | 34 |
| 8.2.2 | cDNA splice site analysis | 34 |
| 8.2.3 | cDNA overlap analysis | 35 |
| 8.2.4 | Blast overlap analysis | 35 |
| 8.2.5 | Total exon length analysis | 36 |
| 8.2.6 | Gene length analysis | 36 |
| 8.2.7 | Average conditional probability analysis | 37 |
| A | GFMerge | 38 |
| A.1 | Input/Output format | 38 |

List of Figures

| | | |
|-----|---|----|
| 2.1 | DNA - RNA - Protein, Rockefeller Uni | 7 |
| 3.1 | Average conditional probability - distribution table | 13 |
| 3.2 | Average conditional probability - formula | 13 |
| 4.1 | Circular Model [1] | 16 |
| 5.1 | Merging rules | 21 |
| 5.2 | BLASTX special case | 21 |
| 5.3 | Logging scheme of <i>GFMerge</i> | 22 |
| 6.1 | Class diagramm GFMerge - data objects | 25 |
| 7.1 | Pseudo code for clustering a sequence | 29 |
| 7.2 | Pseudo code for high scoring path recursion | 31 |
| 7.3 | High scoring path in two dimensional space | 31 |
| 8.1 | Command line arguments | 34 |
| 8.2 | cDNA splice site analysis - illustration | 35 |
| 8.3 | cDNA overlap analysis - illustration | 36 |
| 8.4 | Blast overlap analysis - illustration | 36 |
| 8.5 | Total exon and genemodel length analysis - illustration | 37 |
| A.1 | EMBL file format | 39 |

Chapter 1

Introduction

1.1 Abstract

During the recent years the sequencing has turned into an efficient high-through-put process which accelerated ventures like the Human Genome Project. Producing enormous amounts of raw sequence data the transformation into useable knowledge needs more and more automation to keep up with the accumulated data.

A new path has been striken, *in silicio* analysis, but it will never completely replace *in vitro* lab confirmation. Mathematical and statistical methods like the usage of sequence signals and content, similarity to known genomes are used by so called *gene structure prediction programs* to perform the *ab initio* task of spotting genes on a plain sequence. Although those programs continously further developed and widely used they are unable to provide the desired correctnes in automatic gene discovery.

Therefore it can be usefull to combine predictions from various gene prediction programs. On average each of those programs has an accuracy of its prediction which is approximately 80of my project was the re-analysis and combination of predictions from different programs in order to gain a higher accuracy. To perform this task additional external data like similarities to known sequences was used ([2]).

The result is the Java application *GFMerge* which generates a condensed prediction for a sequence out of various gene structure predictions.

1.2 Acknowledgements

Many thanks ...

1.3 The Sanger Insitute

The Sanger Insitute, which was founded in April 1993, is one of the world's leading sites of genome sequencing and analysis. It started with only 15 staff members in temporary laboratories but increased its workforce to more than 600 to date. Named after the double Nobel Laureate Dr Fred

Sanger, who pioneered the gene-sequencing technique still used at the Genome Campus today, it based in Hinxton, Cambridge, UK. The institute's largest programme, predominantly founded by the Wellcome Trust, was the contribution to the Human Genome Project. One third of the whole project was completed at Hinxton. Furthermore research is done on genomes of organisms causing some of our deadliest diseases, such as tuberculosis and malaria ([3], [4]).

Chapter 2

Computational gene prediction

This chapter briefly introduces biological terms which are used in the ensuing introduction into computational gene prediction.

2.1 Gene structure

A gene is a discrete unit of hereditary information consisting of a specific nucleotide sequence in DNA (or RNA in some viruses). It is involved in producing a polypeptide chain ([5]). Its main characteristic is the organization of its structure into exons and introns. An exon can be described as any segment of an interrupted gene which is represented in the mature RNA product. Whereas an intron is a component of DNA which is transcribed, but removed from within the transcript by splicing together the sequences (exons) on either side of it ([6]).

Generally speaking, exons can be divided into four classes:

- 5' end exons
- internal exons
- 3' end exons
- intronless genes / single exon genes

2.2 Computational Gene Prediction

In 1982, Fickett developed first programs to predict coding regions in genomic DNA sequences. In terms of computational efficiency and accuracy of predictions, substantial progress was made during the nineties. Later during the nineties, analysis went from sequences shorter than a few kilobases to chromosome size sequences. But accurate identification of every gene in a genome by computational methods still is distant goal. [7]

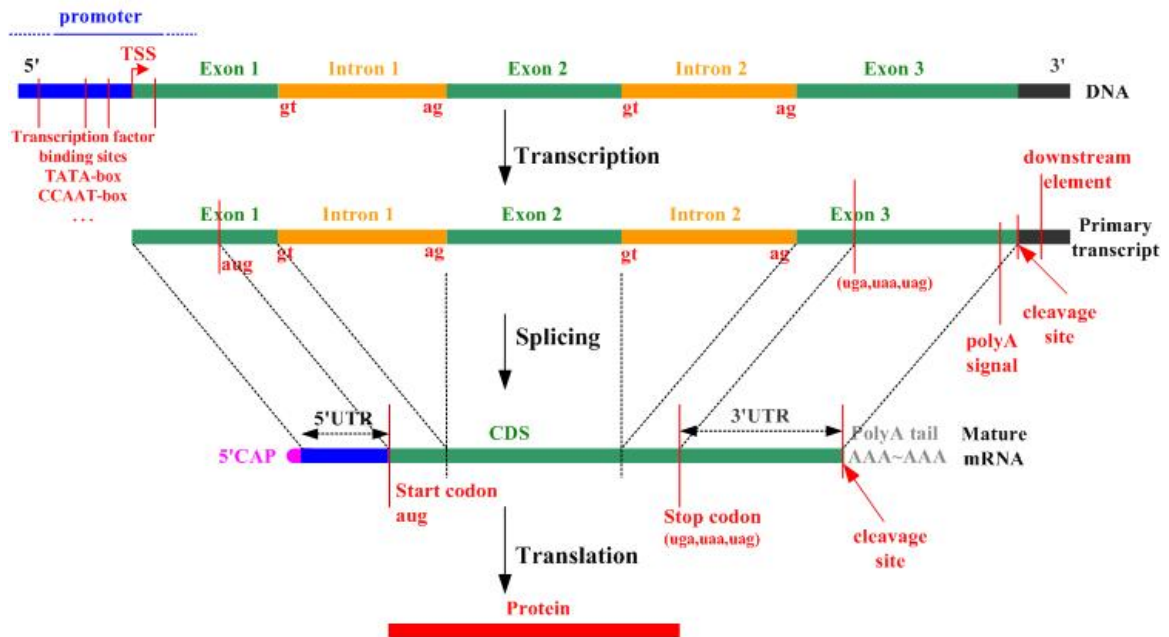


Figure 2.1: DNA - RNA - Protein, Rockefeller Uni

2.2.1 Similarity based Approach to Gene Prediction

Unlike *ab initio* methods similarity based methods seek to extract information from genomic sequence data by comparing the sequence with other sequences which are known to be coding or not. The aim is to find segments which are conserved over evolutionary time. Two main classes of similarity-based gene detection approaches can be distinguished. On one hand, the comparison of a DNA query sequence with a protein or cDNA sequence or database of such sequences, and on the other hand, the comparison of two or more genomic sequences. When comparing sequences from different species one can detect conserved fragments which can be distinguished from non-conserved segments. [7]

As an example the approach of Gelfand et al., 1996 [8] will be described. Given a genomic sequence, they first find a set of *candidate blocks* that contains all *true* exons. This can be done by selecting all blocks between potential *acceptor* and *donor* sites. (i.e., between AG and GT dinucleotides) with further *filtering* of this set (in a way that does not lose the actual exons). The resulting set of blocks can contain many false exons, of course, and currently it is impossible to distinguish all actual exons from this set by a statistical procedure. Instead of trying to find the actual exons, Gelfand et al. select a related target protein in GenBank and explore all possible block assemblies with the goal of finding an assembly with the block highest similarity score to the *target* protein. The number of different block assemblies is huge, but the *spliced alignment* algorithm, which is the key ingredient of the method, scans all of them in polynomial time.

Genomic Query Against Protein or cDNA Target An essential part of similarity based gene prediction programs is constituted by methods which rely on comparison of query sequences with

protein or cDNA sequences. A popular database search program is the *Basic local alignment tool* (*BLAST*), a computer program for comparing DNA and protein sequences. The *BLASTX* version translates a genomic query into a set of amino acid sequences in all six reading frames. Then it compares the set of amino acid sequences against a database of known proteins. In doing so, *BLASTX* identifies segments in the genomic query which are similar to database proteins. Proteins are likely to correspond to coding exons. *BLASTN*, *FASTA* are tools for comparison of a genomic query against a database of cDNA sequences such as ESTs. [2],[7]

These database search programs are no dedicated gene prediction tools. They report exclusively matching sequences and are not capable to automatically identify start and stop codons or splice sites. After database search and identification of potential targets, additional tools are required to define exonic structures.

Despite a considerably higher accuracy of similarity based gene prediction tools, in the practice of large scale sequence analysis, pure *ab initio* gene prediction programs appear to be preferred over their extended versions which incorporate sequence similarity searches. Part of the reason may be difficulty of usage and substantially longer execution time. Another reason may be inconvenient training and incomplete reference database which are temporally dynamic, exponential in growth and contain accumulation of annotation errors.[7]

Genomic Query Against Genomic Target The above described genome-genome comparison uses the rationale that a random mutation in a functional, not necessarily protein-coding, region is usually deleterious to the organism, and hence unlikely to become fixed in a species, whereas mutations in non-functional regions are not, or at least much less, surveyed by natural selection. According to this, they are more likely to be fixed and thus generate the sequence divergence between species that we observe today. This fact is exploited in order to localize genes, determine gene structures and regulatory regions and to infer gene function. The potential to infer complete gene structures from their similarities with genes in related species decreases with divergence time, leading to increased number of false-negative predictions (missing genes). [7]

2.2.2 Statistical Approach to Gene Prediction

Open Reading Frames The most simple way for detection of potential coding regions is the search for *Open Reading frames (ORFs)*¹ On average the distance between a Start and Stop codon² amounts to 21 bases. In this case a long ORF is a potential gene. Unfortunately, long ORFs fail to detect short genes or short exons. [9]

Codon usage Another statistical measure is the *codon usage*. It can be used to recognize diffuse regularities in protein coding regions. A 64-mer vector gives the frequencies of each of 64 possible codons in a window. Codon usage vectors differ between coding and non-coding windows. Therefore they can be used as a measure for gene predictor. [9]

¹Sequence of codons in DNA that starts with a Start codon, ends with a Stop codon and has no other Stop codon inside.[9]

²A codon is a three-nucleotide sequence of DNA or mRNA that specifies a particular amino acid or termination signal; basic unit of the genetic code ([5])

Sequence features In order to recognize exons in 'sea' of intronic DNA specific *sequence features* are used which appear frequently in genes and infrequently elsewhere. Those features can be divided into two types, *signals* which correspond to boundary sites and *content* which corresponds to extended functional regions. [2]

Scoring A scoring function can be deployed to evaluate each sequence feature. Genes are predicted by finding the gene structure with the highest score, given the sequence. The best scoring function is the *conditional probability* $P(a/s)$ where a given *sequence* s contains a *feature* a . The score is the likelihood P of s containing a . [2]

Splicing signals Certain signals can be deployed to detect intron-exon junctions. These so called *splicing signals* are conserved sequence segments of eight nucleotides at an exon-intron boundary (5' end or donor splice site) and a sequence of four nucleotides at an intron-exon boundary (3' end or acceptor splice site). A signal can be represented as a consensus pattern of most frequent nucleotides at each position of an alignment. The frequency information is captured by *Position Weight Matrices* which assign frequency based scores to each possible nucleotide at each position of the signal.

2.2.3 Gene prediction concepts

The trend in gene prediction goes from statistics-based to similarity-based and EST-based algorithms. The combinatorial approach includes protein similarities to derive exon-intron structure. Later those predictions are used for experimental verification by biologists.

Linear Discriminant Analysis and Quadratic Discriminant Analysis Two classical, statistical pattern-recognition methods that are used to categorize samples into two classes. Once samples have been represented as points in space, linear discriminant analysis (LDA) finds an optimal plane surface that best separates points which belong to two classes. Quadratic discriminant analysis (QDA) finds an optimal curved (quadratic) surface instead. Both methods seek to minimize some form of classification error. For example, if there are ten true exons and ten *pseudoexons*³, and two feature variables - 5' splice-site (ss) score and 3'-ss score - these samples could be represented by 20 points in a two-dimensional space (the 5'-ss score on the x axis and the 3'-ss score on the y axis). LDA (or QDA) would compute a straight (or curved) line through the space that can best separate the two classes of exons (with the minimal classification error). [2]

Perceptron Method A machine learning algorithm for pattern recognition or classification. Unlike LDA-based approaches, which calculate theoretically the final best-discriminant plane, a perceptron method is based on a simple neural network which begins with an arbitrary initial plane and then iteratively moves the plane in a way that tries to reduce the classification error at each step. [2]

³A pre-mRNA sequence that resembles an exon, both in its size and in the presence of flanking splice-site sequences, but that is never recognized as an exon by the splicing machinery.

Hidden Markov Models Probability models that were first developed in the speech-recognition field and later applied to protein- and DNA-sequence pattern recognition. Hidden Markov Models (HMMs) represent a system as a set of discrete states and as transitions between those states, each of the possible transitions having an associated probability. Markov models are 'hidden' when one or more of the states cannot be observed directly. HMMs are valuable in bioinformatics because they allow a search or alignment algorithm to be built on firm probability bases, and it is straightforward to train the parameters (transition probabilities) with known data. [2]

Hexamer-Coding Measures Some methods interpret sequences as successions of *words* (so-called because nucleotides are not independent of each other, but tend to occur together as if in a word) of length k (k -tuples); 6-tuples are called hexamer. In-frame hexamer frequencies in a region of DNA have traditionally been used as a powerful way of discriminating coding regions from non-coding regions, as some *words* are more likely to be present in either type of DNA. A score s for a hexamer w , such as CAGCAG, can be defined as $s(w) = \log(freq(w))$. Because the frequency of CAGCAG is relatively high in exons, its score in exons will be higher than that of, for example, TAATAA. [2]

Weight Matrix Method and Weight Array Method Used for scoring a signal motif site. In the weight matrix method (WMM), a score $s(x,b)$ is assigned to each position x for each base pair b , such that the total score of a motif site can be calculated as the sum of scores at all positions in the site. In the weight array method (WAM), a score $s(x,w)$ is assigned to each position x for each word w of length k (when $k=1$, the two methods are the same). [2]

Maximal-Dependence Decomposition (MDD) Donor Matrices A set of donor splice-site weight matrices that are generated using the WMM, each of which is built for a different class of splicing donor sites in such a way that the dependence between nucleotide positions is minimized. [2]

Decision Tree A classification scheme, which can be used, for example, to split a sample into two subsamples according to some rule (feature variable threshold). Each subsample can be further split, and so on. [2]

Artificial Neural Networks A collection of mathematical models that emulate some of the observed properties of biological nervous systems and draw on the analogies of adaptive biological learning. The key element of the artificial neural network (ANN) model is the novel structure of the information processing system. It is composed of many highly interconnected processing elements that are analogous to neurons and are tied together with weighted connections that are analogous to synapses. Once it is trained on known exon or intron sample sequences, it will be able to predict exons or introns in a query sequence automatically. [2]

Chapter 3

Benchmarking gene predictions

3.1 Measuring the prediction accuracy

The first comprehensive comparative analysis of gene prediction programs was published by Burset and Guigo, 1996 [10]. A number of gene prediction programs were run on a large set of vertebrate genomic sequences coding for single genes and performance metrics were introduced to evaluate accuracy of predictions.

In order to evaluate the accuracy of a gene prediction program on a test sequence, the by the program predicted gene structure has to be compared with the actual gene structure of the sequence experimentally validated by, for instance, mRNA. The accuracy can be evaluated at different levels of resolution:

- nucleotide level
- exon level
- gene level

These three levels offer complementary views of program's accuracy. At each level two basic measures can be introduced, *sensitivity* and *specificity*. They are essential measures for prediction errors of the first and second kind. *Sensitivity* (Sn) is the proportion of real elements (coding nucleotides, exons, genes) that have been correctly predicted. In contrast, *Specificity* (Sp) is the proportion of predicted elements that are correct. Both measures values lie inbetween 0 and 1. In a perfect prediction both measures are equal to 1.

Neither Sn nor Sp alone constitute good measures for global accuracy since one can have sensitivity with little specificity and vice versa. A single scalar value desirable to summarize both measures. In gene finding literature, the preferred measure on the nucleotide level is the *correlation coefficient*. At the exon level, an exon is considered correctly predicted (TE) only if predicted exon is identical to the true one, in particular both 5' and 3' end boundaries have to be correct. A predicted exon is considered wrong (WE), if it has no overlap with any real exon. A summary measure on the exon level is simply the average of *sensitivity* and *specificity*. At the gene level, a gene is correctly predicted if all of the coding exons are identified, every intron-exon boundary is correct, and all exons are included in the proper gene. Missed genes (MG) are the fraction of true genes for which

none of its exons is overlapped by any predicted gene. Wrong genes (WG) are the fraction of those predicted genes for which none of the exons is overlapped by any real gene.

3.2 Using the prediction accuracy for merging of gene predictions

When merging different gene predictions provided by diverse gene structure prediction computer programs one will soon realize that there are no standardized performance and quality metrics which enable consistent comparison among the programs in terms of the prediction quality.

If one has to choose among overlapping genemodels of different predictions one cannot use the scores and probabilities provided by the gene prediction programs due to the fact that this information is calculated in complex mathematical algorithms which differ from each program and sometimes are not publicly available. Introducing a standardized scoring system seems to be essential for further comparison.

For the purpose of calculating a accuracy measure a test data set has to be spotted. When choosing this test data set one has to take into account that gene prediction programs were trained on certain sequences. To minimize the overlap of test and training data in order to obtain a non biased measure one should consider only sequences which were entered in databases after the release of the prediction programs although some programmes are continuously updated and further developed.

Bearing in mind that most gene predictors have problems with finding the exact boundaries of exons, a prediction program which recognizes the correct region of a gene, but mispredicts the boundaries of the gene, would achieve a similar score as a program which predicts non-existing genes. For this reason the nucleotide level was chosen for determining the prediction accuracy.

One can represent the coding value of nucleotides along the sequence as a joint distribution of two binary variables which form a 2x2 contingency table (Figure 3.1). This table contains four cells. The number of nucleotides which are correctly predicted (*true positive*) are placed in the upper left cell while the number of nucleotides which are predicted as coding but are non-coding in reality (*false positive*) are placed in the upper right cell. The lower left cell contains the quantity of nucleotides which are wrongly predicted as non-coding (*false negative*) while the lower right cell contains nucleotides where prediction and reality agree in non-coding (*true negative*).

With the information of the table, one is able to calculate measures of accuracy. Burset and Guigo already mentioned that neither sensitivity nor specificity alone can be used as global measures for the prediction accuracy. There were several approaches to combine sensitivity and specificity. Most of these combined measurements were in special situations undefined or the components (Sn and Sp) were not equally weighted.

One appropriate measure for global prediction accuracy is the 'Average Conditional Probability' (*acp*) which was chosen for comparative analysis in GFMerge (Figure 3.2).

The *acp* value embodies two advantages, firstly, it summarizes the information of the 2x2 contingency table and secondly, it can be computed in any circumstances and is in this way an appropriate measure for global prediction accuracy.

The *class PredAccuracy* implements the computation of the *acp* calculation. It has to be noted that due to the double helix structure of the DNA genes can be situated on both strands. The

| | | Reality | | |
|------------|------------|---------|------------|---------|
| | | coding | non-coding | |
| Prediction | coding | TP | FP | TP + FP |
| | non-coding | FN | TN | FN + TN |
| | | TP + FN | FP + TN | |

Figure 3.1: Average conditional probability - distribution table

$$acp = \frac{1}{4} \left[\frac{TP}{TP+FN} + \frac{TP}{TP+FP} + \frac{TN}{TN+FP} + \frac{TN}{TN+FN} \right]$$

Figure 3.2: Average conditional probability - formula

first part of the calculation is the transformation of the nucleotides, which represents one of the four bases, into a binary variable which has either the state coding or non-coding. In two steps the forward and reverse strand of the prediction are compared with the annotation. For each of the four cases (true positive, true negative, false positive, false negative) an integer variable is incremented. According to the formula the *acp* value is calculated.

Chapter 4

Software Development

4.1 Object-Oriented Software Development (OOSD) Methodology

The main features of object-oriented programming are *abstraction*, *encapsulation*, *inheritance* and *reuse*. An entity in the problem domain is abstracted as an *object*. The object encapsulates related data (in form of instance variables) and its operations (called *methods*). An object can be *inherited* by another object to provide reuse. The *reuse* is "flexible" in that the "child" object can add new operations (and data) and/or modify existing ones.

Object-Oriented Software Development focuses on objects during its phases, *analysis*, *design*, *implementation* and *testing*. During object-oriented *analysis* the problem is described. Functional and non-functional requirements are determined. Entity-Relationship diagrams are drawn. An entity-dictionary which is a repository of entity names and descriptions is written. Entities are validated as black boxes. The next step is object-oriented *design* when object diagrams are developed and data structures are determined. For each object specifications are identified. Pseudo-code is written for each operation. Test strategies are developed for class integration. After design follows *implementation*. An adequate object-oriented programming is chosen to produce source code which will be debugged and then compiled. In the OOSD methodology *unit testing* proceeds as follows. For each class, test its member functions. Test all constructors/destructors and overloaded operators. *Integration test* all methods within the class. It has to be sure that class specifications are met. A *test log* entry should specify who, what, when, inputs, outputs both expected and actual, and conclusions. [11]

4.2 Appliance of OOSD in the Circular Model for Software Development

During the last decades the computer industry matured and hardware became smaller, faster and cheaper. Computer systems have grown larger and more complex. As a result, the methodology used to produce software has correspondingly grown in complexity. In an increasing number of systems, it is necessary to build and operate a portion of the system before the requirements for the entire system can be thoroughly understood. The *circular model* proposes a software life cycle model

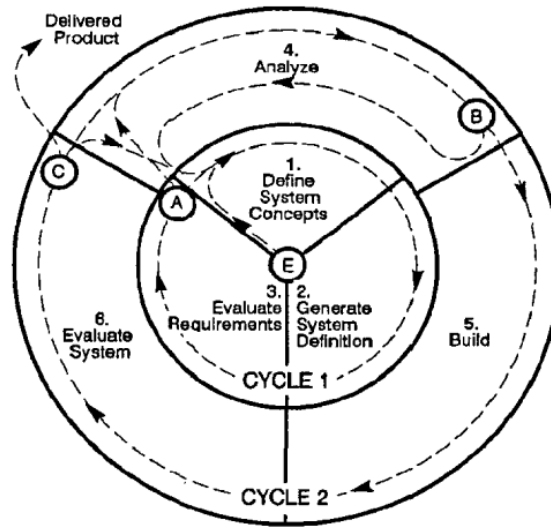


Figure 4.1: Circular Model [1]

that not only addresses resolving requirements, prototyping and incremental development, but also provides a method to handle the total software life cycle in a continuum without discontinuities that are inherent in other models.

The circular model (Figure 4.1) was developed out of the need for a process that could be used not only for current methodologies, but also for methodologies that are emerging in *Object Oriented Development (OOD)*. The entry point into the circular model is in the center at point E. *Cycle 1* defines the system and plans the overall development. Requirements are analyzed and planning is done to lay out the development activities such that the system is built in the most efficient manner. Reviews of the product of the current stage involve the customer to ensure that the system is meeting the specified requirements. *Cycle 1* consists of the stages *Define Concepts*, *Generate Definitions* and *Evaluate Requirements*. *Cycle 2* is concerned with implementing the system defined in cycle 1 and consists of the stages *Analyze*, *Build (Implementation)* and *Evaluate System*. There are three major decision points in the circular model. Decision point A controls the transition of the process into cycle 2. If the results of the requirements evaluation (stage 3) indicates that the proposed solution generated during stage 2 does not satisfy the customer's requirements then cycle 1 is repeated until the requirements are adequately defined and clarified. Decision point B provides the mechanism to resolve anomalies between requirements and design before coding commences. Decision point C, crossing the boundary of cycle 2 to denote completion of the system, brings an orderly end to the life cycle model. The boundary point crossing occurs when the criterion for satisfactory performance of the system is met. [1]

Chapter 5

Development of *GFMerge*

5.1 Project description

Large scale sequencing of genomes, which is done at the Wellcome Trust Sanger Institute, produces a continuing data flow which has to be analyzed automatically. The goal is to extract as much biological knowledge from a sequence as possible. The core element of the analysis is finding genes. For this a number of different gene prediction programs are used on the same sequence. Results are parallelly inspected by a biologist in combination with evidence from other sources such as comparisons to known sequences from the same or another organism. Frequently genemodels proposed by various gene prediction tools for a particular region do not agree completely. Differences can be in level of confidence as given by the gene prediction program, structure, size or location. A typical situation, for instance, is that one program predicts a single gene in a region whereas another predicts two separate ones in that area. Overall the aim is to derive the most plausible genemodel taking into account all available evidence. This means that the best possible genemodel could represent a composite of the structures proposed by the individual tools. When the maximum amount of evidence is available one criterion can take precedence over another: e.g. a genemodel of the largest possible size may not be judged to be the authoritative one if evidence from similarity to other sequences indicates there is reason for it to be split into two separate genes.

The aim of the project is to develop a tool to automate parts of the above evaluation process. It would apply the criteria of confidence score, size and similarity to other known sequences to condense all possibilities into the most likely genemodel which could then be inspected by a biologist. The task would have to perform compromise: normalisation of the score, clustering genemodels which are exactly or approximately in the same region of the sequence and integrating the high quality segments of each prediction and sequence similarity search into a single valid consensus genemodel. The tool should be expandable so that other evidence for gene structures could be added as it becomes available. The development of the tool would include putting together a functional specification, making a design of the tool, writing and testing the tool itself. It is expected that Java classes may be used for carrying out mathematical functions. It should be noted that it is desirable for this tool to be written in Perl using BioPerl objects. This would allow the tool to read data in from simple EMBL files or interface with other BioPerl compatible software. Furthermore it would also allow the output to be easily read into databases and other software packages which are currently used in the

software and database environment of the Pathogene Sequencing Unit.

5.2 Software development process according to the Circular Model

5.2.1 Cycle 1

Define Concepts

Cycle 1 is concerned with definition of the system. During the *Define Concepts* state the objectives of the system are elaborated through the analysis of the customer supplied system specification. A document that describes the operation of the system is written detailing the system's objectives and how the objectives are envisioned to be accomplished. Stage 1 has a high degree of customer interaction to ensure that the customer's needs are fully understood. [1]

At the beginning, a rough description of the project was provided (see section 5.1). After consulting relevant literature (especially Burset et al., 1996 [10]), the project domain could be narrowed down and first ideas were formulated. Later, during several user interviews the project description was iteratively improved and gained in detail. *Feasibility studies* and *Requirements analysis* were subsequently performed.

First insights:

- The idea of integrating high quality segments of different gene predictions from a same region into a new gene prediction was discarded due to the high complexity of rules and exceptions, problem domains like frameshifting¹ and ORFs² can be mentioned.
- It was decided to implement a set of simple rules (Figure 5.1). These rules are the base for the merging process when one gene model will be picked over other overlapping ones in a region.

Generate Definition

Stage 2, *Generate System Definition*, is concerned with top level specifications for the software. A top level system design is formulated by decomposing the total software system into software subsystems. [1]

The system was basically divided into two subsystems, *data object definition* and *analysis classes*. Data objects represent *gene prediction programs* and their *gene predictions*. A gene prediction holds predicted *gene models* which consist of *exons* and *introns*. *Similarity* data, like BLAST or cDNA evidence, contains exons and introns as well. Analysis classes embody *sequence similarity analysis*, *protein similarity analysis*, *gene model length analysis* and *accuracy of the gene predictor*. A thorough description of the data object definition and analysis classes follows in section 6.3 and 6.2.

¹occurring when the number of nucleotides inserted or deleted is not a multiple of 3, thus resulting in improper grouping into codons and consequently changing the frame in which triplets are translated into a protein

²check for amino acids with termination codons becomes necessary

5.2.2 Cycle 2

Cycle 2 is entered when the customer and contractor make the decision that the system is adequately understood and specified. Adequately means that the abstraction of the system that is being transitioned into cycle 2 can be implemented in its entirety. [1]

Analysis

The *Analyze* stage consists of designing the software to a detail where coding can easily take place. This is accomplished by performing the design and validating the design against the specifications detailed during cycle 1. The *Analyze* stage ends when the software design satisfactorily passes the design review. [1]

For *data object definition* a class diagram was produced (see Figure 6.1) which clearly states dependencies among data objects and their inheritance. Major algorithms of the subsystem *analysis classes* were formulated into *pseudo code* (see Figures 7.2, 7.1). A analysis hierarchy which states the chronology of analysis steps was developed (see Chapter 8.2).

Implementation

The decision to enter the *Built* stage is based upon a complete design that contains no open items, i.e., no infamous TBDs (To Be Determined). [1]

Like in section 4.2 described, it might be necessary to build some parts of the system before most requirements for the whole system can be understood. There will be always requirements which have been overlooked during analysis and design, but are necessary to implement. In *bioinformatics* the incremental software development process often reminds of a *build-and-fix approach*[12]. It does not matter how well analyzed and designed the first program version is, it always has to be modified, to include biological exceptions although it often breaks a well-designed implementation.

A brief example shall illustrate the issue. In a test set, following situation (Figure 5.2) occurred. Three gene models overlap a BLASTX feature. Although the first gene model has a better total overlap, gene model 2 and 3 have a slightly higher accumulated overlap on the high-scoring path and are therefore picked. In biological terms, it is better to choose gene model 1, because all of its exons match a single protein. To make the BLAST analysis algorithm take gene model 1, intron overlap was introduced. This involved several changes in the analysis class hierarchy. Basic methods in the abstract class *FeatureOverlapAnalysisTools* had to be adjusted which entailed into alterations in the classes *CDnaOverlapAnalysisTools* and *BlastOverlapAnalysisTools*.

Although the project description stated differently, the object-oriented language Java was favored over the script language Perl for implementation. Apart from a better performance, *BioJava*³, a comfortable Java library for bioinformatics issues, was one of the decision criterions. The decision for Java did not negatively affect the usability, but it allowed a better implementation using a clean object-oriented paradigm. Even though the final program will be built in a Perl pipeline, it can be easily wrapped by a Perl script.

³The BioJava Project is an open-source project dedicated to providing Java tools for processing biological data. This will include objects for manipulating sequences, file parsers, CORBA interoperability, DAS, access to ACeDB, dynamic programming, and simple statistical routines to name just a few things. The BioJava library is useful for automating those daily and mundane bioinformatics tasks. As the library matures, the BioJava libraries will provide a foundation upon which both free software and commercial packages can be developed. For further information see www.biojava.org

Code and unit test In this stage the design is coded and unit tested. [1] Apart from the *data object definition* and *analysis classes*, several submodules were developed separately and thoroughly tested. The *high scoring path recursion* (see 7.2) as well as the calculation of the *average conditional probability* (see 3) can be mentioned in this context.

During the implementation process, modules were independently tested. For logging purposes in order to debug and test a new class was implemented, which enables the user as well as the developer to log certain data which is produced while calculation. The logging functionality is separated from the analysis code. When running *GFMerge* the verbosity can be adjusted. One can either log a single *modul*, a *level* of modules, which corresponds to an analysis level, or the whole *program* processing data. The debugging mode can be passed as a command line argument.

Subsystem and System integration After testing had been accomplished and the modules had been proved working correct, they were integrated to form the final program *GFMerge*.

Evaluate System

The *Evaluate System* stage consists of integrating the product with the rest of the system, *independent testing* and formal *acceptance testing* to *demonstrate* the system satisfies the requirements specification. At the end of the this stage the software is delivered to the customer and the project is completed.

Independent and Acceptance test After the integration of all modules the program was put to the acid test by several biologists. Variations from the desired output, like shown in Figure 5.2, were eliminated.

Operational demonstration During the development process, an *API*⁴ was developed by using *literate programming*⁵ commands for *JavaDoc*. The program was introduced by a *presentation* explaining the "black-box" mechanisms and a written *documentation* was provided.

⁴ An Application Programming Interface (API) is a set of definitions of the ways in which one piece of computer software communicates with another. It is a method of achieving abstraction, usually (but not necessarily) between lower-level and higher-level software. One of the primary purposes of an API is to provide a set of commonly-used functions—for example, to draw windows or icons on the screen. Programmers can then take advantage of the API by making use of its functionality, saving them the task of programming everything from scratch. APIs themselves are abstract: software which provides a certain API is often called the *implementation* of that API.[13]

⁵Literate programming is a certain way of writing computer programs. It is seen as communications to human beings, as works of literature, hence the name "literate programming". Documentation and source code are written into one source file. The compilable source code and the formatted documentation can be extracted from this file with specific utilities (*JavaDoc for Java*). The information is presented in a reading order suitable for human consumption (*API in HTML*). The code is automatically rearranged for computer execution. [13]

If different gene models in one region overlap:

- ① Pick gene model with highest sequence similarity (cDNA). If several gene models have the same similarity, keep all of them.
- ② Pick gene model with highest protein similarity (BLAST). If several gene models have the same similarity, keep all of them.
- ③ Several identical predictions take precedence over a single non-identical gene model.
- ④ Pick rather long gene models than shorter gene models not to lose any coding information. Total exon length which is the length of the gene model without introns and genemodel length which includes all introns are used for comparison.
- ⑤ Pick gene model with the highest *average conditional probability* of its prediction program, if several overlapping gene models are either identical or if there is no other evidence.
- ⑥ Use percentage coverage of similarity data to score the overlap.
- ⑦ Accumulate all BLAST hits from the same protein to a gene model and use the sum for scoring.
- ⑧ If two or more predictions overlap, take the one with the highest score.
- ⑨ A new scoring system has to be introduced due to the fact that gene structure prediction programs use different scoring algorithms whose scores cannot be converted in an universal scoring system.

Figure 5.1: Merging rules

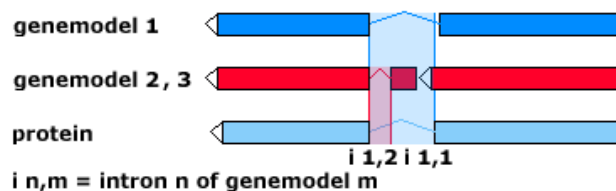


Figure 5.2: BLASTX special case

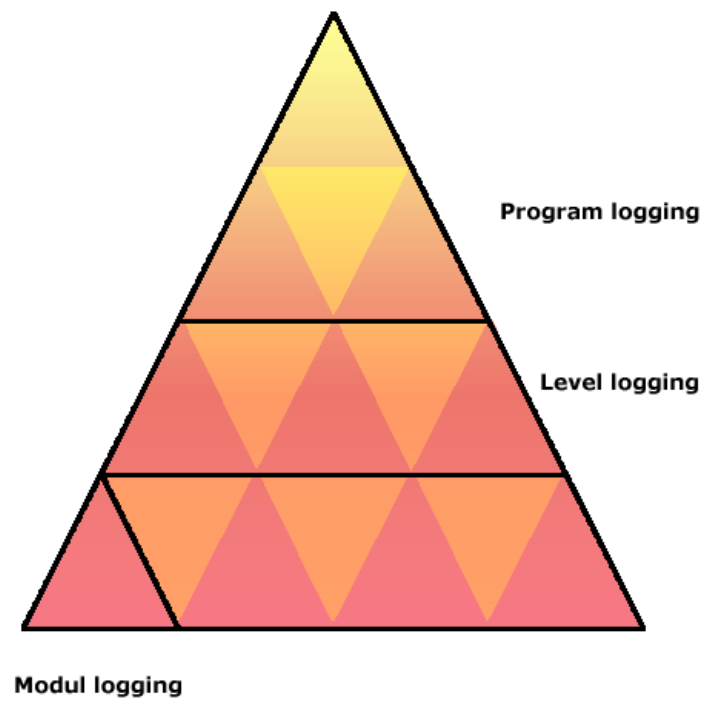


Figure 5.3: Logging scheme of *GMerge*

Chapter 6

Software architecture and design of *GFMerge*

6.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) follows the hype of object-oriented analysis and design methods. After a standardization process with the Object Management Group (OMG) it is now an OMG standard. Although it is called a modeling language UML is rather a method. It embodies both a modeling language as well as a process. The modeling language is mainly the graphical notation. The advice which steps to take while doing a design can be assigned to a process. The main goal of software development is the final code which is going to be executable by the user. In this sense UML does not primarily support the development process. The fundamental reason for the use of UML is a better communication. Certain concepts can be described more clearly. Natural language is too imprecise when it comes to more complex concepts. On the other hand the source code is precise but too detailed. UML can emphasise important details while blanking out information which are unnecessary for the overall view of a system.

Basically, class and sequence diagrams were used for the analysis of requirements and the design of GFMerge. Class diagrams map requirements into an object-oriented model. Classes show parts which need further work and are a good base for user-developer communication. Each class should be treated as a concept in the user's mind. Another good approach is the usage of sequence diagrams which illustrate parts of the program flow. A good understanding of the user's world is essential for developing good software. Especially in bioinformatics difficulties are reassigned when computer scientist meets biologist.

6.1.1 Class diagrams

Class diagrams are the backbone of most object-oriented methods. A class diagram describes the types of objects in the system and the relationships that exist among them. Operations and attributes are assigned to classes within the diagram. Constraints show relationships among objects.

As described in Cook's and Daniels' book "Designing Object Systems" [14] there are three perspectives which can be used when drawing a class diagram. The first perspective is the *Conceptual*

or *Essential* one. The diagram represents concepts of the domain under study. There does not have to be a direct mapping inbetween the conceptual model and the later implementation. It can be considered as language-independent. The second perspective is called *Specification*. It deals with interfaces of software, not with their implementation. The core of effective object-oriented programming is the programming to a class' interface rather than to its implementation. The *Implementation* shows the implementation of classes. This view is the most often used although in many cases it is better to stick to the specification perspective.

For the modelling of GFMerge I decided for the *implementation perspective*. The first diagram (Figure 6.1) combines dependencies between data structure objects. It includes interfaces which are used. The view follows the succession of instantiation. The first objects which are created are *GenePredictor* instances. In the next step *Prediction* objects are assigned to appropriate gene predictor. Each prediction contains a list of *GeneModel* instances which hold themselves *Exon* and *Intron* objects. *GeneModel* and *SimFeature* (protein or sequence similarities) objects can be pooled to *GFMergeRegion* objects.

6.1.2 Sequence diagrams

6.2 Data structures in GFMerge

6.2.1 class GenePredictor

The class *GenePredictor* represents a single gene structure prediction computer program. Objects of this class are instantiated for each prediction file in the training data set. While the constructor is called a *PredAccuracy* object does the calculation of the 'average conditional probability' and returns a value which is stored in the attribute *acp*.

The second attribute *name* contains the program's label found in the training file in the *method tag* of every feature. All tags have to have the same value otherwise an exception is thrown.

The name is used to establish references among gene predictor objects and prediction objects. The label used in the prediction files has to be the same like the ones used in the training set alternatively predictions cannot be assigned to the evaluated prediction program and an exception would be thrown.

6.2.2 class Prediction

After the instantiation of *GenePredictor* *Prediction* objects are created. They wrap a BioJava *Sequence* and contain additional information. Referencing to a gene predictor an absolute value about the quality of each prediction can be obtain. Furthermore a prediction contains an *ArrayList* of *GeneModel* objects.

The attribute *genePredictorName* is gained from the prediction file and used for producing a reference to the gene predictor.

6.2.3 abstract class GFMergeFeature

The abstract class *GFMergeFeature* is the basic structural design of common attributes and methods of class *GeneModel* and *SimFeature*. A predicted gene, a cDNA and a Blast feature have the same

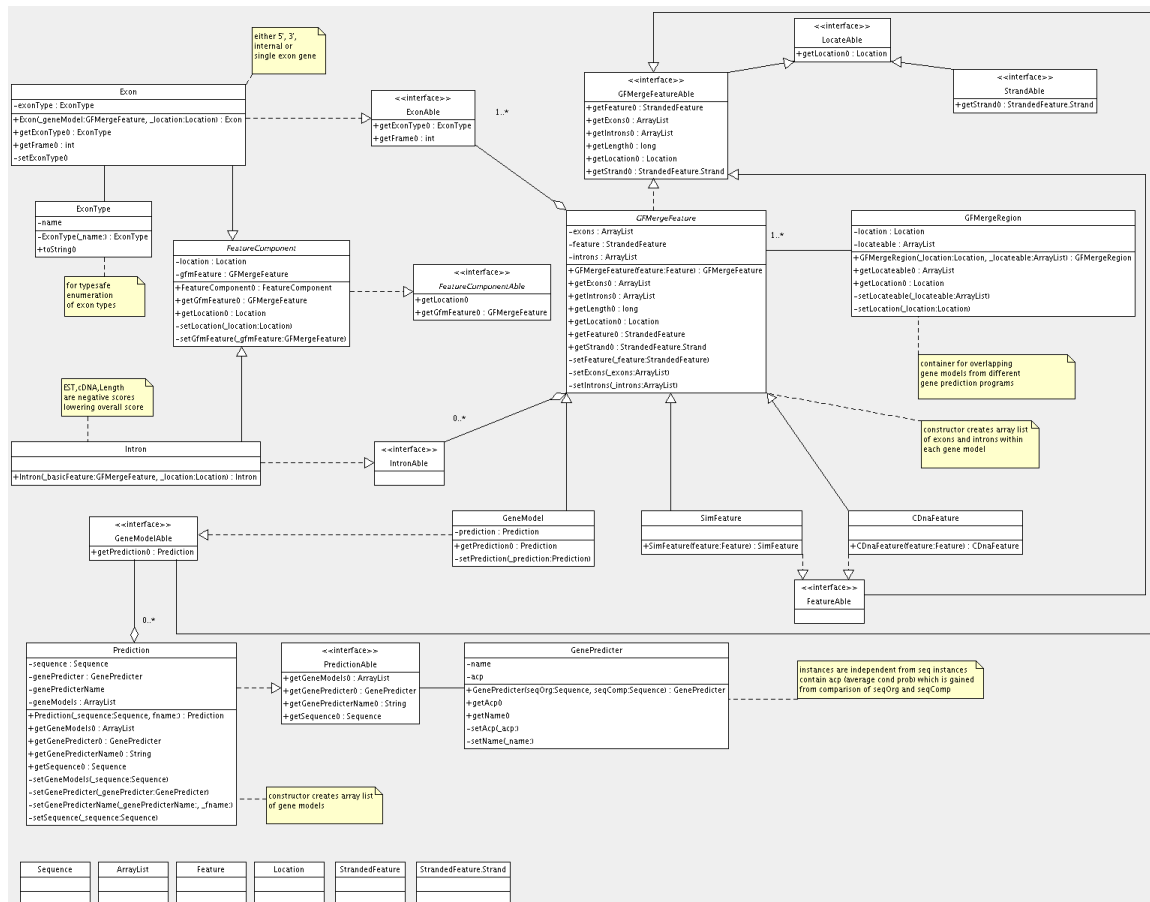


Figure 6.1: Class diagram GFMerge - data objects

structure. They consist of *Exons* and *Introns* and have a location on the sequence, are stranded and have a length. *GFMergeFeature* wraps the BioJava *Feature* class.

abstract class FeatureComponent

The abstract class *FeatureComponent* represents class *Exon* and *Intron*. Common attributes are location and a reference to the *GFMergeFeature* the belong to.

class Exon

The class *Exon* is derived from the abstract class *FeatureComponent*. An additional attribute is the *ExonType*. The class *ExonType* was introduced for **typesafe enumeration in Java** and contains following values which an exon can be assigned to:

- *TYPE_5END* which means that the exon is the 5'-end of a feature

- *TYPE_3END* which means that the exon is the 3'-end of a feature
- *TYPE_INTERNAL* which means that the exon is an internal one
- *TYPE_SINGLE_EXON_GENE* which means that the feature consists of only a single exon

class Intron

The class *Intron* is derived from the abstract class *FeatureComponent*. In contrast to exons which are modelled as sublocations of a *Feature* object in BioJava introns are not implemented. A single intron corresponds to the gap inbetween two exons. It is calculated by the program.

class GeneModel

The class *GeneModel* is derived from the abstract class *GFMergeFeature*. Each genemodel contains a list of exons and introns, a reference to its prediction and it wraps a BioJava *StrandedFeature*.

A genemodel represents a location on the sequence which a gene prediction program conjectures a real gene.

class SimFeature

The class *SimFeature* is derived from the abstract class *GFMergeFeature*. It embodies both, cDNA and Blast features which are used as evidence to proof the correctness of a predicted gene. Apart from the reference to a prediction a *SimFeature* has the same properties as a *GeneModel*.

6.2.4 GFMergeRegion

The class *GFMergeRegion* is instantiated to gain non-overlapping clusters of *GFMergeFeature* objects. The list of regions across the sequence is calculated by a recursive algorithm. Each region contains overlapping *GFMergeFeature* which are either genemodels, cDNA or Blast features. Clustering the sequence in independent regions enables a faster calculation because redundant combinations of genemodels do not have to be considered. Attributes of *GFMergeRegion* are a location and an ArrayList of *LocateAble* which is an interface for *GFMergeFeature*.

6.3 Program flow of GFMerge

6.3.1 Main class GFMerge

GFMerge is considered as the main class of the application. It contains the stub of the logic. Firstly, an object of the class *SimpleArg* is created. All arguments provided when started the program are wrapped in a convenient way. The information can be retrieved easily and independently by using accessor methods. For further processing the reference of this *SimpleArg* object will be passed to methods.

As a second step a *GFMerge_analysis* object is created. To clearly separate the main program flow from the data processing the class *GFMerge_analysis* contains all analysis steps as invisible objects. The method *getMergedFeatureTable()* returns the result of the calculation, an ArrayList of genemodel regions of *GFMergeRegion* where no genemodel should overlap another one anymore.

This ArrayList of genemodel regions is saved to a file which represents the final output of GFMerge. After comparing the merged feature table to the original gene predictions provided as input removed genemodels are spotted and can be saved to an additional file.

6.3.2 class GFMerge_Analysis - preprocessing

As mentioned above a *GFMerge_Analysis* object is created in main class *GFMerge*. When calling the constructor *GFMerge_Analysis()* creates an ArrayList of *GenePredictor* objects, one for each training data file. Using the predictor objects and all prediction files an ArrayList of *Prediction* objects is generated where each prediction is linked to its gene predictor. A prediction contains a set of *GeneModels* which themselves consist of *Exon* and *Intron* objects.

Similarly cDNA and Blast objects are created according to the structural design of class *Sim-Feature*. These objects embody exons and introns as well as genemodels but they are not linked to a prediction and gene predictor object.

6.3.3 class GFMerge_Analysis - computing

Having packed the gene predictions and evidence data, cDNA and Blast features, into handy objects the computing can start. When calling the method *getMergedFeatureTable()* the first time the analysis is carried out. An ArrayList of *BasicFeatureAnalysis* items, an abstract class which *CDnaSpliceSiteAnalysis*, *CDnaOverlapAnalysis*, *BlastOverlapAnalysis*, *TotalExonLengthAnalysis*, *GeneModelLengthAnalysis* and *AvgCondProbAnalysis* are derived from, contains the analysis objects in the order of processing. Processing is kicked off in the analysis objects by calling method *getHighScoringGmRegArr* which returns an ArrayList of regions containing high scoring genemodels.

Chapter 7

Selected algorithms of *GFMerge*

7.1 Clustering the sequence

Most of the time gene predictions are unevenly distributed across a sequence. Some areas are full of overlapping features, others seem to be non-coding. One could think of clustering the sequence in independent regions of overlapping features. This approach would enable an acceleration of the analysis process since analysis is done separately in each region which means a high reduction of possibilities to consider. Individual results are merged afterwards to a continuous conclusion. For this purpose the class *GFMergeRegion* was introduced. Its data objects contain a cluster of overlapping *GFMergeFeature* objects.

The algorithm which is used to determine the location of regions is based on a recursion (Figure 7.1). Beforehand a preprocessing is done where genemodels of all *Prediction* objects are put into a tree map sorted by their start location (key). Due to the fact that several genemodels can identical or similar locations the value pair contains an array of genemodels with the same start location.

Java provides the class *TreeMap* based on *red-black tree*. Keys are sorted in ascending order. All operations on the tree take a $O(\log N)$ time cost in the worst case. A red-black tree is a binary search tree and can be described as following ([15]):

- ① Every node is colored either black or red.
- ② The root is black.
- ③ Every red node that is not a leaf has only black children.
- ④ Every path from the root to a leaf contains the same number of black nodes.

The beneficial effects of the coloring rules are a height of the red-black tree which is not exceeding the height of $2\log(N+1)$ and guarantees logarithmic search operations.

Due to this fact this binary tree enables efficient storage and retrieval of ordered items as long as the tree is balanced. A new item is inserted by placing it as a leaf. If the its parent is black, the color of the item should be red. On the other hand, if the parent's color is red, the has to be adjusted. Following operations are possible, changing the color of the tree's nodes and rotating the tree.

```

AoRegs      := array of regions;
ToGms       := tree of genemodels;
AoGmsInReg  := array of genemodels in single region;
AoOvGms     := array of overlapping genemodels;
AoOvGmsTemp:= array of overlapping genemodels;
HoGms       := hash of processed gms;

AoRegs = search(ToGms)
{
  HoGms      = new Hash();
  foreach gm in ToGms
  {
    AoGmsInReg = find(gm, HoGms, ToGms);
    AoRegs.add(AoGmsInReg);
  }
}

AoGmsInReg = find(gm, HoGms, ToGms);
{
  if (HoGms.contains(gm))
  {
    return null;
  }

  else
  {
    HoGms.put(gm);
    AoOvGms = findOverlaps(gm, ToGms);
    foreach gm in AoOvGms (= gmo)
    {
      AoOvGmsTemp = find(gmo, HoGms, ToGms);
      AoGmsInReg.add(AoOvGmsTemp);
    }
  }
  return AoGmsInReg;
}

AoOvGms = findOverlaps(gm, ToGms)
{
  foreach gm in ToGms (= gmComp)
  {
    if (gmComp.overlap(gm))
    {
      AoOvGms.add(gmComp);
    }
  }
  return AoOvGms;
}

```

Figure 7.1: Pseudo code for clustering a sequence

7.2 High Scoring Path

7.2.1 Description

The high scoring path is the path within a cluster or region of overlapping gene models which contains the maximum score as accumulation of single scores of non-overlapping gene models on the way.

In different stages of the merging process a score is assigned to a single gene model. In order to find the mentioned high scoring path a recursive algorithm was implemented in GFMerge.

The illustration shows four lanes which represent four gene predictions of different programs. One will realize that the boundaries differ as well as the allocated scores printed above each gene model. (The length of a gene model is not proportional to its score.) Different test modules were developed with the task of finding the highest score of a sequence of gene models within a cluster.

The first approach was a search algorithm which operated in the way as following described. The set of gene models would be sorted in a list. Starting with the highest item the list would be scanned in a descending order picking the next lower gene model which does not overlap the already selected ones. It attracted our attention that we would get a high score path but the accumulated score would not be the maximum score which could be achieved. In certain situations a group of non overlapping lower scoring gene models would have a higher score.

Aware of this problem a recursive algorithm was developed which would find all possible combinations and would choose the gene models on the path with the highest total score.

7.2.2 Implementation and optimization

The algorithm was implemented according to the pseudo code (Figure 7.2). The correctness was proven by further testing using a test data set similar to the one of the illustration (Figure 7.3).

Later the algorithm was optimized due to the fact that one ends up with all possible combinations (faculty of the number of gene models on the path) which represent the same path but in a different order of gene models. The number of possible combination exponentially increases with the length of the path. A number of rules were introduced which would reduce redundancy.

The first optimization is that a path should always start at the 'left corner' of the cluster. The first gene model of the paths should be the furthest downstream element, there should be no elements in the path array after the first element with a location further downstream than the first element.

The test data cluster contained thirteen elements. Before the introduction of optimization 1 the number of all possible paths was 640 (Table 7.2.2). After the first reduction of redundancy the number of paths could be decreased to 158. All paths start at the left corner of the cluster although there were still some paths which are identical apart from the order of their elements.

In order to get rid of the described redundancy another optimization was used. Only gene models should be added to a path if they have no predecessor which should be further downstream. The result of this optimization was the reduction of 158 pathes to the number of 34 in the test data set.

```

HoGms      := hash of genemodels;
AoP        := array of paths;
AoSP       := array of single path (containing gms);
HoHScP     := hash of highscoring path;
AoNoGms    := array of non overlapping gms;

HighScoringPath = findHighScorePath(HoGms)
{
    AoP        = getAllPaths(new AoP, new AoSP, HoGms);
    maxScore   = getMaxPathScore(AoP);
    HoHScP     = getHighScoringPath(maxScore, AoP);
    return HoHScP;
}

AoP = getAllPaths(new AoP, new AoSP, HoGms)
{
    AoNoGms    := getNonOverlappingGms(AoSP, HoGms);
    if(AoNoGms.size = 0)
    {
        AoP.add(AoSP);
    }
    else
    {
        foreach gm in AoNoGms
        {
            tempArr = copy of AoSP;
            tempArr.add(gm);
            AllPaths(AoP, tempArr, HoGms);
        }
    }
    return AoP;
}

```

Figure 7.2: Pseudo code for high scoring path recursion

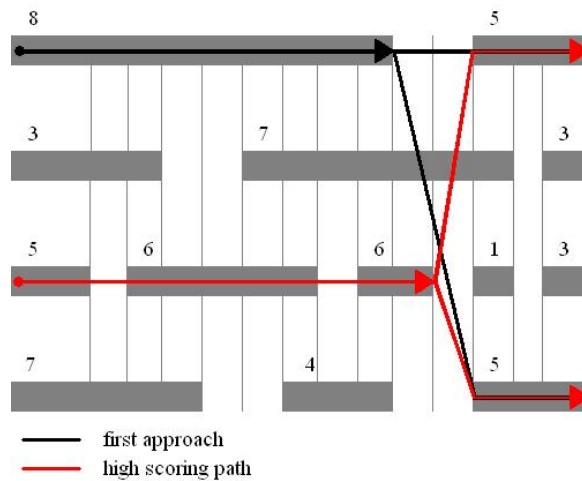


Figure 7.3: High scoring path in two dimensional space

| Optimization | No. of paths |
|--------------|--------------|
| before | 640 |
| opt. 1 | 158 |
| opt. 2 | 34 |

Table: Optimization of recursive algorithm

A third optimization in the algorithm concerned iteration over the pathes while looking for the high scoring path. Instead of using an array of pathes in which each path would be an array of gene models the paths were stored in a tree map sorted by their score. The sorting process would be automatically done when inserting the path element in the tree. The high scoring path could easily obtained by taking the last tree element. This optimization would lead to a saving of processing time which exponentially increases with the number of gene models within the cluster.

Chapter 8

Documentation of *GFMerge*

8.1 reference manual

Due to the fact that the main application area will be in a pipeline no graphical user interface is primarily needed. Being able to type all arguments used for processing on the command line *GFMerge* can be easily wrapped by a script (Perl script) to start computation within a pipeline. Furthermore it can be easily used from a command shell by using copy and paste to insert all necessary arguments.

If arguments are incorrect or missing a help message will be displayed. For later processing the accuracy of each genepredictor has to be measured in order to compare genepredictors to each other on the base of their correctness. To gain a non-biased measure an independent sequence with an annotation file as well as one geneprediction file for each genepredictor should be given. This so called training set should be a long sequence and if possible from the same organism in order to measure the accuracy. Command line arguments are [-a] for the annotation file and [-t "file1 file2 .. fileN"] for n genepredictions.

The other part of the command line arguments concerns the actual genepredictions which have to be collapsed into one. The argument [-p "file1 file2 .. fileN"] should contain the predictions from the same genepredictors like the ones of the training set. Being used as biological evidence a single cDNA file [-c] and Blast file [-b] are mandatory.

According to the quality of cDNAs strand consideration can be switched on by using the [-y] tag. When confirming exon of a genemodel internal cDNA boundaries are used. The allowed deviation of an exon boundary from the original cDNA boundary, described as fuzzy boundary, can be entered with the tag [-x]. The default value is the length of 15 bases upstream and downstream from the original location.

The tag [-f] is used to specify the name and path of the "path/filename.tab" file which represents the collapsed genepredictions. When the output of removed features is switched on [-r] a single "path/filename.removed.AnalysisLevel.tab" file is saved at each analysis level containing the set of genemodels which have been removed at this very level. At the end a "path/filename.removed.tab" file of all removed features is saved (Figure 8.1).

```

[-h]    help
[-a]    annotated training set
[-t "file1 file2 .. fileN"]  predictions on training set
[-p "file1 file2 .. fileN"]  predictions of gene predictors
[-b]    blast file
[-c]    cDNA file
[-f]    filename for output
[-r]    switch for output of removed features
[-x]    fuzzy cDNA boundaries
[-y]    consider cDNA strand

```

Figure 8.1: Command line arguments

8.2 Merging process

8.2.1 Introduction

The merging process is done in a hierarchy of different analysis stages. The order of those levels was carefully chosen in a way that the conclusiveness is declining from the first to the last level. Less accurate evidence should be used only if there is no other information available.

At each vertical stage overlapping gene predictions are compared to each other on the base of a horizontal scoring system. The sequence has been clustered in non-overlapping regions which contain themselves coinciding genemodels. If one calculates the high scoring path within all of those clusters the union is the high scoring path of the whole sequence since each cluster is independent.

Not having a direct correlation inbetween length of a genemodel and its score the high scoring path might not be the path of the high score non-coinciding gene predictions. It can happen that a number of short genes can have a higher accumulated score. For this reason every possible way through the two dimensional space of non-overlapping gene predictions has to be considered.

8.2.2 cDNA splice site analysis

The first level of analysis is the comparison of cDNA splice sites to exon boundaries of predicted genemodels. Only internal (exon) boundaries of cDNAs are considered for comparison (BIOLOGICAL REASON). In order to evaluate each single genemodel a splice site scoring system was introduced. If a genemodel exon boundary exactly overlaps a cDNA splice location, it obtains a score of "+1". A genemodel exon boundary scores "-1" in the case that it overlaps a cDNA location but not precisely a cDNA splice site. Whereas a genemodel boundary does not overlap a cDNA location at all, it receives a score of "0". If a cDNA splice site does not overlap a genemodel exon boundary but a genemodel location, a score of "-1" is added to the genemodel. The total score of every genemodel is calculated by summing up the score of all individual genemodel sublocations. Having scored each genemodel the program can find the highscoring path in the clustered sequence and removes the low scoring ones. Fuzzy boundaries of cDNAs are considered. They can be described as allowed deviation of a cDNA splice site and a genemodel sublocation boundary. The default is "15" bases but can be changed to an arbitrary value (Figure 8.2).

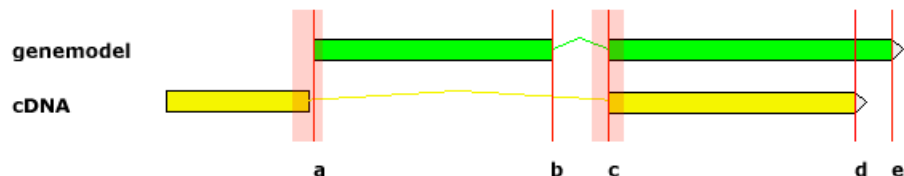


Figure 8.2: The illustration shows a gene prediction which is supported by a cDNA feature. The light red area around the cDNA splice sites are referred to as fuzzy boundaries. Within a certain deviation genemodel boundaries are proved right. Only internal intron/exon splice sites of the cDNA feature are considered.

In (a) the genemodel boundary overlaps an internal exon splice site. Because of the genemodel boundary being a start position and the cDNA splice site being an end position the cDNA boundary does not support the genemodel splice site. The boundary of the genemodel is scored with "-1" and for the non-overlapped cDNA splice site another "-1" is added to the genemodel score. The genemodel splice site in (b) overlaps the cDNA but matches no cDNA splice site. A score of "-1" is added to the total genemodel score. In (c) the genemodel and cDNA boundary agree, the total score is incremented by "+1". In (d) and (e) no value is added to the score. In the first case, external cDNA splice sites are not used for proving correct genemodel boundaries [BIOLOGICAL MEANING !!!] and in the second case the genemodel boundary is not considered because it does not overlap a cDNA. The total score is "-2".

8.2.3 cDNA overlap analysis

As a second level of analysis a scoring system of genemodels gained by their absolute overlap with cDNAs has been introduced. All sub features of cDNAs are compared to exons of predicted genemodels in order to calculate the overlapping number of bases. It does not matter if only one or more cDNAs overlap a genemodel or its exons. All overlap is collapsed into one compound location which the base overlap is determined from (Figure 8.3).

8.2.4 Blast overlap analysis

In the third step blast overlap is taken into account. For each genemodel overlapping blast hits are determined. Due to the fact that blast hits of different proteins can match a single genemodel one should distinguish the overlap of different proteins to a genemodel. On this account the blast overlap of each protein is calculated and the longest protein overlap is taken as score (Figure 8.4).

A percentage overlap analysis in the case of cDNA and blast overlap has been condemned because the algorithm would favour a shorter genemodel to a longer genemodel if both of them have the same or similar absolute overlap.

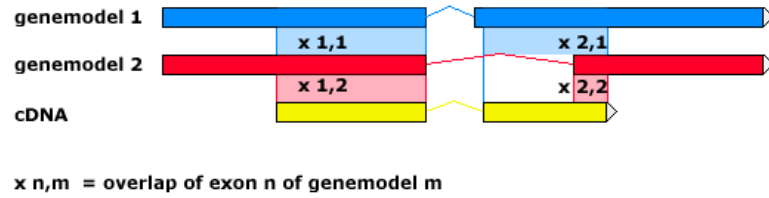


Figure 8.3: In this figure two genemodels overlap a cDNA feature. Although the first exons' overlap of both genemodel is equal in length ($x_{1,1} = x_{1,2}$). The second exon of genemodel 1 has a longer overlap than the one of genemodel 2 ($x_{2,1} > x_{2,2}$). Consequently genemodel 1 will gain a higher score than genemodel 2.

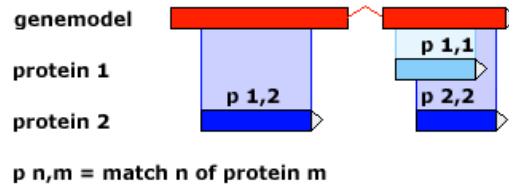


Figure 8.4: Three blast features of two different proteins matches a genemodel. As one can see is the overlap of protein 1 ($x_{1,1}$) shorter than the overlap of protein 2 which hits the genemodel in two different locations ($p_{1,2} + p_{2,2}$). The longer overlap is used for scoring the genemodel which is in this context protein 2.

8.2.5 Total exon length analysis

If there are still overlaps among genemodels after the previous analysis the total exon length is taken to score genemodels in order to keep as much coding information (exons, exon length) as possible. The algorithm rather picks genemodels with long exons than ones with shorter coding segments (Figure 8.5).

8.2.6 Gene length analysis

This step compares coinciding genemodels by using the total length of a genemodel including exon and intron length. After having picked genemodels with the longest possible accumulated length of all their exons the program now advantages genemodels with long introns for the purpose of allocating as much sequence space as possible (Figure 8.5).

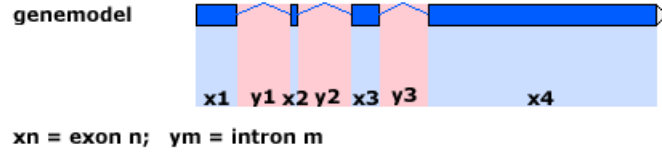


Figure 8.5:

The total exon length of a genemodel is the accumulated length of all exons.

$$t_{gm} = \sum_{i=1}^n x_i \text{ with } x_i \text{ as exon } i.$$

The genemodel length is the accumulated length including all introns.

$$l_{gm} = \sum_{i=1}^n x_i + \sum_{j=1}^m y_j \text{ with } y_j \text{ as intron } j.$$

$$n \geq 1, n = m - 1$$

8.2.7 Average conditional probability analysis

On the basis of a training data set including gene predictions of each gene prediction program and an annotation for the training sequence the average conditional probability has been calculated. It represents a probability value which embodies a measure of prediction accuracy at the base level. Due to the fact that a number of shorter genemodels with moderate acp value are rather taken than a single longer genemodel with a considerable high score the high scoring algorithm should be used only to choose among overlapping genemodels with similiar length and location on the sequence. The reason for this apparently paradox behaviour is the average conditional probability value should not be accumulated.

Appendix A

GFMerge

A.1 Input/Output format

The input and output data is going to be in a special file format which is called EMBL (Figure A.1). The EMBL format is a standard introduced by the European Bioinformatics Institute (EBI) which is part of the European Molecular Biology Laboratory (EMBL). It can be described as follows:

- no series of header lines are required
- first line in the file begins the first sequence entry of the file
- first line of each sequence entry contains a two letters ID in the first two spaces
- EMBL identifier follows in spaces 6 through 14
- second line of each sequence entry has the two letters AC in the first two spaces
- accession number follows in spaces 6 through 11
- third line of each sequence entry has the two letters DE in the first two spaces
- free form text definition follows in spaces 6 through 72
- fourth line in each sequence entry has the two letters SQ in the first two spaces
- length of the sequence beginning at or after space 13 follows
- blank space and the two letters BP after sequence length
- nucleotide sequence begins on the fifth line of the sequence entry
- each line of sequence begins with four blank spaces
- next 66 spaces hold the nucleotide sequence in six blocks of ten nucleotides
- each of the six blocks begins with a blank space followed by ten nucleotides
- first nucleotide is in space 6 of the line while the last is in space 70
- last line of each sequence entry in the file is a terminator line which has the two characters // in the first two spaces
- multiple sequences may appear in each file

```

ID  AA03518      standard; DNA; FUN; 237 BP.
XX
AC  U03518;
XX
DE  Aspergillus awamori internal transcribed spacer 1 (ITS1) and 18S
DE  rRNA and 5.8S rRNA genes, partial sequence.
XX
SQ  Sequence 237 BP; 41 A; 77 C; 67 G; 52 T; 0 other;
    aacctgcgga aggatcatta ccgagtgcgg gtcctttggg cccaacctcc catccgtgtc      60
    tattgtaccc tgttgcttcg gcgggcccg cgttgcgg ccgccggggg ggccgctctg      120
    ccccccgggc ccgtgcccgc cggagacccc aacacgaaca ctgtctgaaa gcgtgcagtc      180
    tgagttgatt gaatgcaatc agttaaaact ttcaacaatg gatctcttgg ttccggc      237
//

```

Figure A.1: EMBL file format

Bibliography

- [1] Marvin J. Carr. A circular model for software development. *Washington Ada Symposium*, pages 129–133, 1989. 3, 16, 18, 19, 20
- [2] Michael Q. Zhang. Computational prediction of eukaryotic protein-coding genes. *Nature Reviews Genetics*, 3:698–709, September 2002. 4, 8, 9, 10
- [3] The Wellcome Trust. The sanger institute, 2003. <http://www.wellcome.ac.uk/en/1/bioengensan.html>. 5
- [4] The Wellcome Trust. Sanger, old and new, 2003. <http://www.wellcome.ac.uk/en/1/awtpubnwswnoi29ana2.html>. 5
- [5] Lawrence G. Mitchell Neil A. Campbell, Jane B. Reece. *Biology*. Pearson Benjamin Cummings; 5th edition (January 1999), 1999. 6, 8
- [6] Paul Siliciano Benjamin Lewin. *Genes VI*. Oxford University Press; 6th edition (April 1997), 1997. 6
- [7] Thomas Wiehe Roderic Guigo. *Gene Prediction Accuracy in Large DNA Sequences*, pages 1–33. Caister Academic Press, Wymondham, UK, 2003. 6, 7, 8
- [8] Pavel A. Pevzner Mikhail S. Gelfand, Andrey A. Mironov. Gene recognition via spliced sequence alignment. *Proc Natl Acad Sci USA*, 93(17):9061–9066, August 1996. 7
- [9] Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. MIT Press; 1st edition (August 21, 2000), 2000. 8
- [10] Burset et al. Evaluation of gene structure prediction programs. *Genomics*, 34(3):353–367, June 1996. 11, 18
- [11] John H. Crenshaw Thomas J. Cheatham. Object-oriented vs waterfall software development. *ACM Annual Computer Science Conference*, pages 595–599, April 1999. 15
- [12] Daniel M. Berry. The inevitable pain of software development, including of extreme programming, caused by requirements volatility. *Agile Alliance*, September 2002. 19
- [13] Wikipedia the free encyclopedia. Computer terms, September 2003. <http://www.wikipedia.org/wiki/>. 20

- [14] John Daniels Steve Cook. *Designing Object Systems (Prentice Hall Object-oriented*. Prentice-Hall, Inc., 1994. 23
- [15] Florida International University Mark Allen Weiss. *Data Structures Algorithm Analysis in Java*. Addison Wesley Longman, Inc., 1999. 28