Report Number: WUCSE-2003-8

2003-02-14

Twinscan: A Software Package for Homology-Based Gene Prediction

Paul Flicek

A complete mapping from genome to proteome would constitute a foundation for genome-based biology and provide targets for pharmaceutical and therapeutic intervention. This is one reason gene structure prediction has been a major subfield of computational biology for over 20 years. Many of the widely used gene prediction systems were developed in the 1990s and are unable to take advantage of the revolution in comparative genomics brought on by the sequencing of the entire genomes of an increasing numbers of vertebrates. Twinscan is a new system for high-throughput gene-structure prediction that exploits the patterns of conservation observed in alignments... **Read complete abstract on page 2.**

Twinscan: A Software Package for Homology-Based Gene Prediction

Paul Flicek

Complete Abstract:

A complete mapping from genome to proteome would constitute a foundation for genome-based biology and provide targets for pharmaceutical and therapeutic intervention. This is one reason gene structure prediction has been a major subfield of computational biology for over 20 years. Many of the widely used gene prediction systems were developed in the 1990s and are unable to take advantage of the revolution in comparative genomics brought on by the sequencing of the entire genomes of an increasing numbers of vertebrates. Twinscan is a new system for high-throughput gene-structure prediction that exploits the patterns of conservation observed in alignments between a target genomic sequence and its homologous sequence in other organisms. The approach employs a symbolic conservation sequence that effectively combines many local alignments into a single global alignment. This has several important properties that make Twinscan particularly useful for high-throughput gene prediction. For mammals, Twinscan has been shown to be significantly more accurate and reliable by all measures than any non-comparative genomic method. Twinscan is based on, and includes as a component, the same hidden Markov model topology as Genscan, a popular non-homology based gene prediction program. Twinscan has an object-oriented design and is implemented in the C++ programming language. Twinscan's three major components consist of probabilistic models of both the DNA sequence and the conservation sequence as well as a dynamic programming framework. Both the models and the computational structure are complicated aggregate classes. In this report, the design and implementation of Twinscan is described at the source-code level for the first time.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-8

Twinscan: A Software Package for Homology-Based Gene Prediction

Authors: Flicek, Paul

February 14, 2003

Abstract: A complete mapping from genome to proteome would constitute a foundation for genome-based biology and provide targets for pharmaceutical and therapeutic intervention. This is one reason gene structure prediction has been a major subfield of computational biology for over 20 years. Many of the widely used gene prediction systems were developed in the 1990s and are unable to take advantage of the revolution in comparative genomics brought on by the sequencing of the entire genomes of an increasing numbers of vertebrates. Twinscan is a new system for high-throughput gene-structure prediction that exploits the patterns of conservation observed in alignments between a target genomic sequence and its homologous sequence in other organisms. The approach employs a symbolic conservation sequence that effectively combines many local alignments into a single global alignment. This has several important properties that make Twinscan particularly useful for high-throughput gene prediction. For mammals, Twinscan has been shown to be significantly more accurate and reliable by all measures than any non-comparative genomic method.

Twinscan is based on, and includes as a component, the same hidden Markov model topology as Genscan, a popular non-homology based gene prediction program. Twinscan has an object-oriented design and is implemented in the C++ programming language. Twinscan™s three major components consist of probabilistic models of both the DNA sequence and the conservation sequence as well as a dynamic programming

TWINSCAN: A SOFTWARE PACKAGE FOR HOMOLOGY-BASED GENE PREDICTION

Paul Flicek

Department of Computer Science and Engineering
Washington University
Saint Louis, Missouri 63130

Technical Report WUCSE-2003-8
Prepared under the direction of Professor Michael R. Brent
Presented in partial fulfillment of the requirements of the degree of
MASTER OF SCIENCE
December, 2002

Figures

Abstract

A complete mapping from genome to proteome would constitute a foundation for genome-based biology and provide targets for pharmaceutical and therapeutic intervention. This is one reason gene structure prediction has been a major subfield of computational biology for over 20 years. Many of the widely used gene prediction systems were developed in the 1990s and are unable to take advantage of the revolution in comparative genomics brought on by the sequencing of the entire genomes of an increasing numbers of vertebrates. Twinscan is a new system for high-throughput gene-structure prediction that exploits the patterns of conservation observed in alignments between a target genomic sequence and its homologous sequence in other organisms. The approach employs a symbolic conservation sequence that effectively combines many local alignments into a single global alignment. This has several important properties that make Twinscan particularly useful for high-throughput gene prediction. For mammals, Twinscan has been shown to be significantly more accurate and reliable by all measures than any non-comparative genomic method.

Twinscan is based on, and includes as a component, the same hidden Markov model topology as Genscan, a popular non-homology based gene prediction program. Twinscan has an object-oriented design and is implemented in the C++ programming language. Twinscan's three major components consist of probabilistic models of both the DNA sequence and the conservation sequence as well as a dynamic programming framework. Both the models and the computational structure are complicated aggregate classes. In this report, the design and implementation of Twinscan is described at the source-code level for the first time.

Chapter 1
Introduction

In the past five years biology has embraced the genomic era. The first multicellular organism to have its genome sequence published was *Ceanorhabditis elegans* (The *C. elegans* Sequencing Consortium 1998). In the following years, the genome sequences of increasing complex animals have been published, including *Drosophila melanogaster* (Adams et al. 2000), *Tetraodon nigroviridis* (Crollius et al. 2000), *Mus musculus* (Mouse Genome Sequencing Consortium 2002), and *Homo sapiens* (International Human Genome Sequencing Consortium 2001; Venter et al. 2001). Today the complete genomes of nearly 700 organisms have been published or are being sequenced (Bernal et al. 2001; Kyrpides 1999). But there would be little value in all this biological sequence data without high-speed computational methods to analyze it.

One of the most common forms of computational analysis is the search for protein coding genes on long sequences of deoxyribose nucleic acid (DNA), the linear polymer that makes up the genome. In its native state, DNA is a double helix of two anti-parallel strands of nucleotides (the forward and reverse strands) held together by hydrogen bonds between complementary base pairs (bp) (Lodish et al. 2000). DNA nucleotides normally exhibit Watson-Crick base pairing: adenine (A) with thymine (T) and cytosine (C) with guanine (G).

DNA was known to be the molecule of heredity before its structure was determined (Watson and Crick 1953). Later the degenerate genetic code was solved; one *codon* (three bases of DNA) translates to one of 20 amino acids. The central dogma of molecular biology states that genetic information is transcribed from DNA to RNA and translated from RNA into a protein. The messenger RNA (mRNA) is extensively processed between transcription and translation. The DNA for any organism contains the

instructions for synthesizing all of its proteins, but only a small fraction of the total DNA actually codes proteins (in humans about 1-2% of the genome is coding).

The *shotgun* method of DNA sequencing was introduced by Sanger and co-workers (Sanger and Coulson 1978; Sanger et al. 1977). The procedure involves randomly breaking the DNA molecules into 400-800 bp pieces that are individually sequenced. These fragments of the total sequence are called *reads*. By matching overlapping sections, the reads are assembled into the final sequence. To ensure an adequate assembly, the reads must include approximately 5-10 times the number of bases as the original DNA molecule. Any gaps remaining in the assembly are closed through a process called *finishing*.

Computational gene finding is possible because genomic DNA includes coded signals to the cell's protein production machinery. Especially important for accurate gene prediction are the signals related to pre-mRNA splicing. The availability of genomic DNA sequence allows for computational searching for these patterns. For example, in a typical mammalian gene as shown in Figure 1, the protein coding sequence is broken into separate *exons* by non-coding *introns*, while intergenic regions (not shown) separate complete genes. On both ends of the introns are strong splice site consensus sequences, while specific codons represent both translation initiation and termination: ATG for initiation and TAG, TGA, or TAA for termination. In the coding region, the sequence of codons running from an ATG to a stop codon is called the reading frame of the sequence. Flanking the coding region are additional signals regulating transcription, including promoter sequences that may bind regulatory proteins to enhance or diminish the rate of protein production. These signals lead to several classes of probabilistic models to be implemented by any gene prediction program.

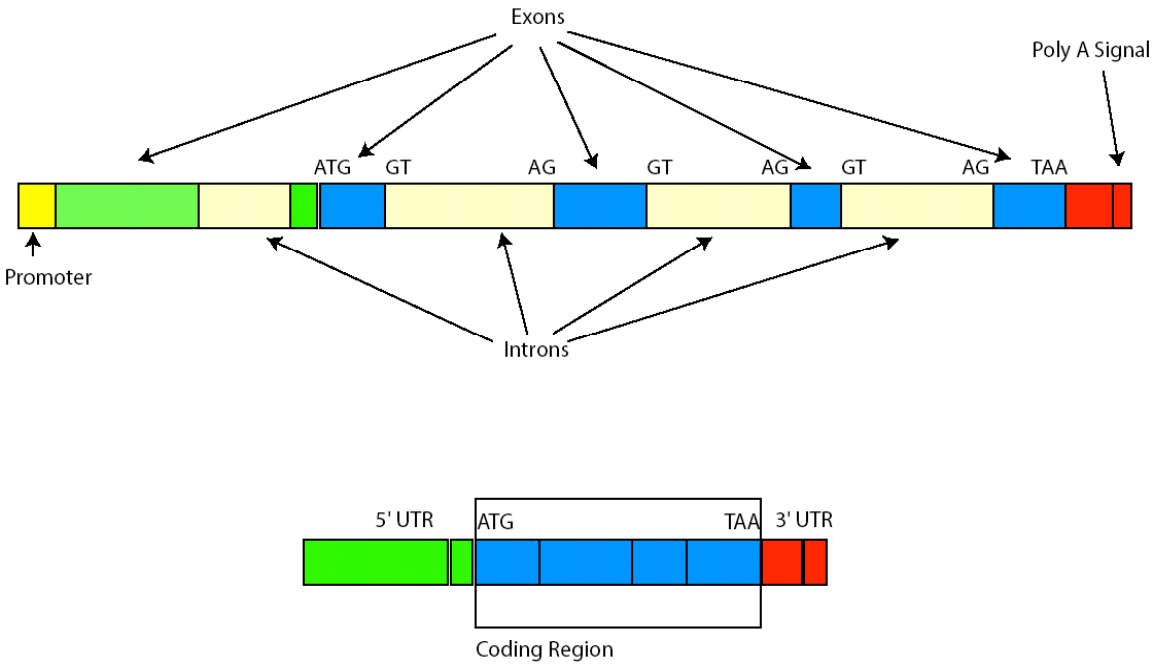

Figure 1. RNA Splicing

Because of its importance and the need to annotate newly sequenced DNA quickly and accurately, computational gene-structure prediction is active area of research. A number of approaches to this problem have been developed with generally increasing amounts of success. However, the fundamental algorithms have changed little since 1997 (Zhang 2002) when the Genscan (Burge and Karlin 1997) program, featuring an explicit-state duration hidden Markov model (HMM) was introduced. Genscan's performance far exceeded other available gene prediction programs, and it dispensed with many of the restrictive and unrealistic assumptions (e.g. one complete gene per sequence) of other programs. Although there are limits to its accuracy, Genscan continues to be widely used and it has become synonymous with gene prediction for many people.

Genscan's most serious limitation is that the model does not account for evolutionary conservation. This is a critical source of information because there is strong selective pressure for exons to be conserved while introns accumulate random mutations. When homologous sections of two genomes (e.g. human and mouse) are aligned, the alignment in coding regions differs from the alignment in non-coding regions. This difference contains information about the evolutionary history of the organisms. The availability of

complete genomes from related organisms allows computational gene prediction guided by evolutionary conservation for the first time. This idea is closely related to that of comparative anatomy, which has long been used to describe the similarities and differences among the physical structures of organisms. It is assumed that organisms closely related in their evolutionary history will exhibit close correspondence of these physical structures and that similar structures are likely used in similar ways. Using this idea to compare fully sequenced genomes of various organisms provides a much more powerful tool for determining gene structure.

Twinscan (Korf et al. 2001) is a new system for high-throughput gene-structure prediction that extends the probability model of Genscan and exploits patterns of evolutionary conservation to improve prediction of protein-coding genes. Its key advance is the use of a symbolic conservation sequence that effectively combines many local alignments into a single global alignment. As an example of the importance of a local alignment strategy, consider the mouse and human sequence alignment in Figure 2. A single portion of the upper mouse sequence, participates in five local alignments with the lower human sequence. Any global alignment method would require that the mouse sequence align to only one place in human. Twinscan's algorithm is able to use all five local alignments to improve predictive accuracy.

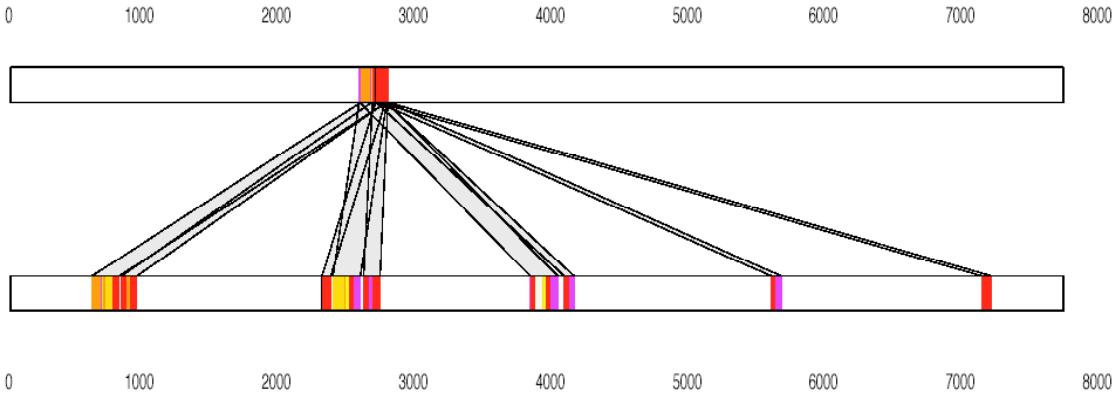

Figure 2. Alignment of 8000 bp of human and mouse DNA.

For mammals, Twinscan has been shown to be significantly more accurate and reliable by all measures than any gene prediction program that does not use comparative

information (Flicek et al. 2003). Additionally, it offers several advantages over previously published comparative genomics algorithms that employ a global alignment between a pair of presumed homologous sequences. These approaches generally carry with them two assumptions: that there is exactly one gene on the sequence of interest, and that *synteny* (i.e. gene order) is maintained between the sequences. Twinscan can handle the more realistic situation of multiple, incomplete or no genes on the target sequence. It also allows for inversions, duplications and changes in exon-intron structure between the target sequence and its homologs. Furthermore, homologs may come from multiple sources, and may be of high or low quality, which allows Twinscan to take advantage of the earliest products of a sequencing project well before sequence assembly or finishing (Flicek et al. 2003). Finally, the system can easily be extended to take into account protein and transcript homologies. These features make Twinscan particularly useful for high-throughput gene prediction.

However, several complications for effective prediction remain. Memory limitations prevent chromosome-sized sequences from being analyzed without fragmentation, a process that splits real genes. Highly expressed genes are more likely to have the compositional biases used by gene finders, likely making unusual genes or those with tissue-specific expression patterns more difficult to find. Indeed, the state of computational gene prediction in absolute terms remains relatively poor, reflecting the difficulty of the problem.

Chapter 2

How Twinscan Works

[*Twinscan is directly based on Genscan and any description of the Twinscan algorithm must include a description of Genscan. Some of the following is summarized from Burge (1997)*[1] *and describes the methods for modeling protein-coding genes.]*

2.0 Hidden Markov Models

Hidden Markov models were first introduced to biological sequence analysis by David Haussler and his colleagues in 1992 from the field of speech processing where they had been used for twenty years (Durbin et al. 1998). An HMM can be described as a model for generating sequence. Each state of the model has ability to both generate bases A, C, G, or T and transition to another state in the model. Both the probability of generating specific bases and which transitions are allowed may be specified for each state. At the end of the process, a sequence of bases is produced and visible, but the underlying sequence of states remains hidden (Eddy 1998). If the states of the HMM are defined to include intergenic regions, introns, exons, and other parts of genes, then the HMM can be used to model gene structure. Gene prediction involves the analysis of a given DNA sequence to determine the most likely sequence of states from the HMM.

2.1 A Parse of the Sequence

The goal of the Twinscan algorithm is to determine the most likely *parse* of a given DNA sequence. A parse is the classification of every base in the input sequence, *S*, into an ordered set of functional states $\hat{q} = \{q_1, q_2, q_3, \cdots, q_n\}$ with an associated set of durations $\hat{d} = \{d_1, d_2, d_3, \cdots, d_n\}$ where the length of *S* is $L = \prod_{i=1}^{n} d_i$.

[1] Interested readers are encouraged to read Christopher Burge's brilliant and highly readable Ph.D. Thesis in its entirety.

Several steps are required to generate a parse: (1) q_1 is determined from an externally defined initial probability distribution on the states, π. As will be described below, there are 27 states in the model. (2) d_1 is generated conditional to q_1 from length distribution f_{q_1} (3) A sequence segment s_1 of length d_1 is generated conditional to both q_1 and d_1 according to the sequence generating model P_{q_1}. (4) The next state, q_2, is generated conditional to q_1 from the Markov state transition matrix, and d_2 and s_2 are generated as above. The process continues until $\prod_{i=1}^{n} d_i \geq L$.

2.2 HMM Topology

The parse is dependent on the topology of the HMM. Twinscan uses the Genscan model topology shown in Figure 3. Arrows indicate state transitions with non-zero probability for genes on the forward strand. States Exon 0, Exon 1, and Exon 2 represent internal exons, states I_0, I_1, and I_2 represent introns following exons with 0, 1, or 2 bases after the last complete codon. Init Exon and Term Exon represent, respectively, the first and last exons in a multi exon gene. The Exon Sngl state is used for genes consisting of a single exon. The upstream untranslated region (UTR) and the downstream UTR are represented by 5' and 3' states, respectively. The gene promoter region is represented by Prom and the poly-adenylation signal is represented by Poly A. An analogous model is used to represent genes on the negative strand. Following the terminology of Burge (1997), the states in the model designated with circles

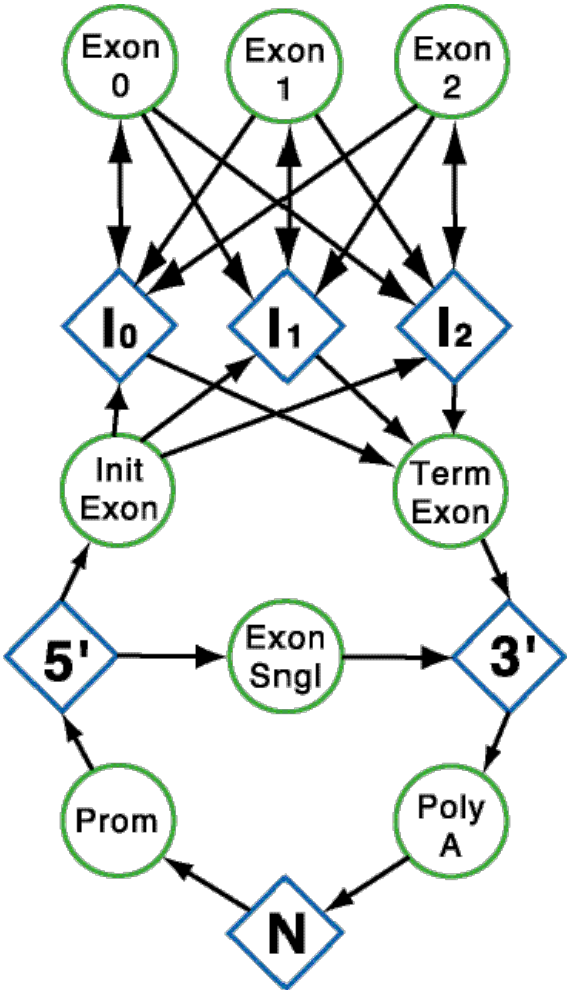

Figure 3. HMM Structure for Genscan and Twinscan

are referred to as *c-states*, while those designated by diamonds are *d-states*. Note that a parse must alternate between c-states and d-states.

Each c-state in the model is itself composed of a number of component models that will be described in detail below. The exon models include both characteristics of the exons such as length distribution and composition as well as the splice site signals that are found immediately adjacent to the exons within the introns. D-states have generally simpler models.

In the model, all obligatory transitions are assigned probability 1.0 (e.g. the transition from Promoter to 5' UTR). Other transition probabilities are parameterized.

2.3 Component Models

Five related models are used for the components of the c-states.

2.3.1 Fifth Order Markov Chain

The best measure to discriminate whether an exon-sized segment of DNA (approximately 100 bp) is coding or non-coding is the reading frame-specific hexamer (i.e. six bases of DNA) composition (Fickett and Tung 1992). This leads to a model of coding regions as an inhomogeneous 3-periodic fifth-order Markov chain in which each position in the codon has a specific fifth-order Markov chain. For example, the coding positions of an internal exon whose first base is the second position in a codon are modeled using Markov chains $C^2C^3C^1C^2C^3$ etc.

The coding model does not apply to the positions at the edges of the exons that overlap the splice signal models. These models are described in the next several sections.

2.3.2 Weight Matrix Model

The weight matrix model (WMM) is the simplest probabilistic model used by Twinscan to describe finite-length biological signals. In the positive model nucleotides of a true

signal (e.g. the translation initiation signal) of length ℓ are assumed to be generated independently according to the position-specific probability distribution. Then the probability of generating any particular sequence, *x*, under the positive model is given by:

$$P\{x|+\} = P^{+}_{WMM}(x) = \prod_{i=1}^{\ell} P^{(i)}_{x_i}$$

which is typically estimated from observed frequencies.

The probability of generating the same sequence, *x*, under the analogous negative model derived from a population of pseudo-sites is also created. This allows for the differentiation of the real sites from pseudo-sites using a signal ratio:

$$r = \frac{P^{+}_{WMM}(x)}{P^{-}_{WMM}(x)}$$

The WMM is used in several ways. It scores the entire 6-bp poly-A state. It is also used for the 12-bp translation initiation signal, used at the start of the initial exon and single exon states, which includes six bases prior to the ATG and three after it. Similarly, the score for the 6-bp translation termination signal includes one of the three stop codons and three downstream bases. The WMM is also used as a component of the maximal dependence decomposition model (explained further below).

A special variation of the WMM is used for the 19-codon signal peptide. This model is a bipartite "codon-level" WMM that generates a codon (rather than a single base). The first part of the model applies to the first 4 codons and the second part applies to the final 15 codons. This is implemented for the initial and single exon states as a mixture model after the translation initiation signal, with 20% of the total probability from the signal peptide model and 80% from the fifth-order Markov chain used in coding sequence.

2.3.3 Weight Array Model

The weight array model (WAM) (Salzberg 1997; Zhang and Marr 1993) is a natural generalization of the WMM to an inhomogeneous n-th order Markov model. This allows for dependencies between adjacent positions.

$$P\{x|+\} = P^{+}_{WAM}(x) = P^{(1)}_{x_i} \prod_{i=2}^{} P^{(i-1,i)}_{x_{i-1},x_i}$$

As with the WMM, this probability is estimated from observed frequencies. Again, an analogous negative model is created. The WAM will capture adjacent nucleotide (i.e. dinucleotide) biases in bulk genomic DNA. Real sites may be distinguished from pseudo-site using a signal ratio as above.

2.3.4 Maximal Dependence Decomposition

The maximal dependence decomposition (MDD) model combines a binary decision tree with multiple WMMs to describe the donor splice site. This model is useful when significant non-adjacent dependencies exist.

The MDD is a collection of WMMs connected by a tree that branches based on the existence of specific nucleotides near the donor splice site. As shown in the adjacent figure, the donor splice site requires a GT consensus. The presence of specific additional bases allows tracking through the tree toward one of the shaded leaves. Each of these leaves contains a WMM that is used to score the donor splice site.

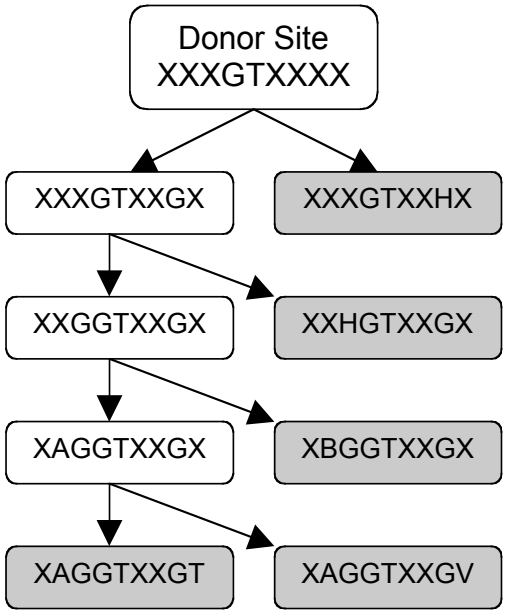

Figure 4. MDD implementation of the donor splice site. Nucleotide symbol H means A,C, or T. Symbol B means C,G, or T. Symbol V means A, C, or G.

For example, if CAGGTTAGT is a potential donor splice site, the WMM located at the leaf on the lower left of the diagram will be used.

This implementation performs better than a simple WMM or WAM, especially in the ability to give low scores to incorrect splice sites.

2.4 Conservation Sequence

Sequence similarity is modeled by a symbolic representation that pairs one of three symbols with each nucleotide of the target sequence. A conservation symbol is assigned to a specific base in the target sequence depending on whether that base is part of an alignment. Twinscan normally incorporates gapped alignments from WU-BLAST (Gish), but could use any local alignment method. Alignments are sorted, so that bases in the target sequence are assigned conservation symbols based on the best local alignment. Target sequence bases are assigned one symbol if they are within an alignment and exactly match the aligned base in the informant sequence; another symbol if they are within an alignment, but do match the corresponding base in the informant sequence; or a third symbol if they are unaligned. Gaps in the informant sequence become mismatch symbols in the conservation sequence, while gaps in the target sequence are ignored. For example, consider the following sequence:

```
123456789 position
GAATTCCGT target sequence
```

The alignment yields the following

```
345 6789  target position
ATT-CCGT  target alignment
||  || |  alignment match symbols
ATCACC-T  informant alignment
```

In this case the first two nucleotides in the target sequence are not aligned. Therefore, the resulting conservation sequence is

```
123456789 position
GAATTCCGT target sequence
..||:||:| conservation sequence
```

In the Twinscan implementation, the symbols in the conservation sequence are represented by numbers (i.e. 0,1,2).

2.5 Conservation Models

Twinscan combines the probability model of Genscan (i.e. the probability of generating a specific DNA sequence) with the probability of generating a specific conservation sequence. Twinscan uses the same set of component models to describe the conservation sequence, but not necessarily the same models for the same states.

Coding, UTR, intron, and intergenic states all model stretches of conservation sequence using homogenous fifth-order Markov chains. The translation initiation and termination sites are also modeled with fifth-order Markov chains. Models of conservation sequence at splice donor and acceptor sites are based on second-order WAMs. Following Genscan's definition, the donor site window is fixed at 9 bp and the acceptor site window is fixed at 43 bp.

2.6 Parameters

The entire model requires thousands of parameters, which must be specified before the calculation of a parse. Ideally these parameters are determined from the analysis of a training set that represents the distribution of examples over which the performance of the system will be measured (Mitchell 1997). In the case of Twinscan essentially all of the parameters are estimated from real data based on published methods (Burge 1997), or calculated from previous data analysis (Bucher 1990). Certain parameters have been tuned manually.

Like the size of a genome, the number of genes varies from species to species, as does the average length and average number of exons. Characteristics such as intron length are also species-specific, and within a species these values may vary depending on the regional GC content. For example, in the human genome, intron length decreases dramatically with increasing GC content, while exon length is essentially unchanged. (International Human Genome Sequencing Consortium 2001). Twinscan's predictive accuracy also varies depending on the GC content of the sequence (Flicek et al. 2003).

One model to address variations in the sequence with GC content considers the genome to be a "mosaic of isochores" – containing large regions of locally homogenous GC content and variations between disjoint regions (Bernardi 1989; Bernardi 2000; Bernardi et al. 1985). A modified version of this model with four isochores was introduced by Burge (1997) with some parameters taking on different values based on the total GC content of the sequence. The Twinscan model supports any number of isochors.

All d-states have geometric length distributions. Both the intergenic (N) and intron (I_x) states have isochore-specific geometric length parameters, while the 3' and 5' UTR states use the same parameter for all isochores. The promoter and poly A states have fixed-length distributions that will be discussed below. Exon length distributions are also independent of isochore and use an explicit distribution.

Non-coding states (5',3', N, I_x) are modeled using a homogenous fifth-order Markov matrix with transition probabilities derived from the non-coding portions of the genes.

2.7 Scoring Possible Exons

Almost all possible exons are scored according to appropriate models as discussed above (See Figure 5 for details). A possible exon has a number of specific characteristics. (1) It is defined as starting with either a start codon (initial exons and single exons) or immediately following an AG acceptor site (internal exons and terminal exons). (2) Possible exons end with one of the three stop codons (terminal exons and single exons) or immediately before a GT donor splice site. (3) Possible exons neither contain in-frame stop codons nor have non-canonical splice sites.

The score values are stored for use in the optimal parse determination. Twinscan uses a two-pass method to determine possible gene structures on the forward and reverse strands. This allows the code and the parameters that model the forward strand to be used twice. A bit of bookkeeping is required to ensure that all of the possible exons are properly stored for the optimal path determination.

Exon Type	Sequence	Signal	Model Used
Initial Exon	DNA	Translation initiation	WMM
	DNA	Donor splice site	MDD
	DNA	Signal Peptide	Codon-level WMM
	DNA	Coding sequence	3-periodic 5th order MC
	Conservation	Translation initiation	Homogenous 5th order MC
	Conservation	Coding sequence	Homogenous 5th order MC
	Conservation	Donor splice site	2nd order WAM
Internal Exon	DNA	Acceptor splice site	3rd order WWAM
	DNA	Donor splice site	MDD
	DNA	Coding sequence	3-periodic 5th order MC
	Conservation	Acceptor splice site	2nd order WAM
	Conservation	Donor splice site	2nd order WAM
	Conservation	Coding sequence	Homogenous 5th order MC
Terminal exon	DNA	Acceptor splice site	3rd order WWAM
	DNA	Translation termination	WMM
	DNA	Coding sequence	3-periodic 5th order MC
	Conservation	Acceptor splice site	2nd order WAM
	Conservation	Translation termination	Homogenous 5th order MC
	Conservation	Coding sequence	Homogenous 5th order MC
Single Exon	DNA	Translation initiation	WMM
	DNA	Signal Peptide	Codon-level WMM
	DNA	Coding sequence	3-periodic 5th order MC
	DNA	Translation termination	WMM
	Conservation	Translation initiation	Homogenous 5th order MC
	Conservation	Coding sequence	Homogenous 5th order MC
	Conservation	Translation termination	Homogenous 5th order MC

Figure 5. Table of all exon types and the models that are used for both the DNA sequence model and the conservation model.

To facilitate the optimal parse determination, possible exons are stored at the downstream—with respect to the forward strand—sequence index. This is accomplished by first finding and storing possible exon begin positions (acceptor sites or translation initiation sites) and then looking for possible exon end sites (donor sites or translation termination sites) and reconciling them with previously stored and still valid exon begin sites. In this way, exons with in-frame stop codons are not included in the list of possible exons. It is possible to introduce an in-frame stop codon across a splice site when exons are joined. This possibility is eliminated with the optimal parse determination below.

2.8 Determine the Optimal Parse

To determine the optimal parse of the sequence, the most probable path through the hidden states of the model must be determined. This process is termed "decoding" and is done with the Viterbi algorithm, a dynamic programming algorithm commonly used with HMMs.

The last step in determining the optimal parse of the sequence is the trace back. In this portion the most probable parse of the sequence is recorded. Safeguards are implemented at this step to ensure that in-frame stop codons are not introduced when adjacent exons are joined.

Chapter 3

Overview of the Code Base

3.0 Introduction

Twinscan is generally run from the command line or as part of an automated analysis pipeline. This section is meant to follow the inputs to the program and the actions of the code base through the analysis of a sequence with the full conservation model described in Chapter 2. Analysis of a sequence with the DNA only (Genscan-compatible) model or with the UtrCds model is similar to the presentation below and the details are left to the reader.

Twinscan requires four inputs: a genome parameter file, a conservation parameter file, the conservation sequence, and the target genomic sequence.

The major steps in the process as displayed in the activity diagram of Figure 6 are

1. Read in the target sequence.
2. Parse the genome parameter file and instantiate a `GenomeModel` class object.
3. Read the conservation sequence.
4. Parse the conservation parameter file and instantiate an `SpsConsModel` class object.
5. Instantiate the appropriate Trellis class object.
6. Calculate the most probable path through the trellis with the Viterbi algorithm.
7. Output the result.

Note that the activity diagram includes paths for all three Twinscan models, but the text will discuss only the full conservation (i.e. SpsCons) model.

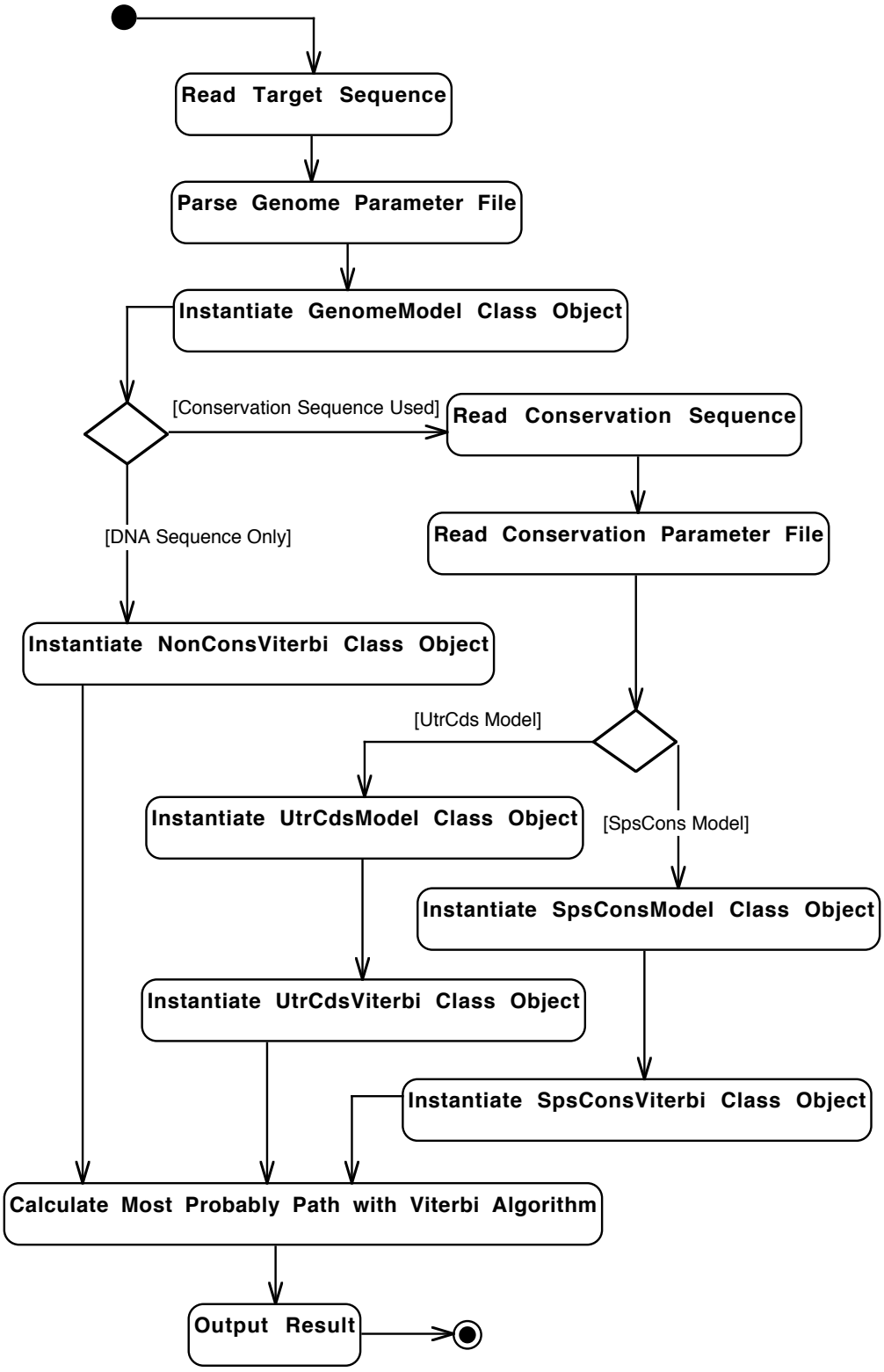

Figure 6. Twinscan activity diagram

3.1 Read in the Target Sequence

The target sequence in *fasta format* is read by functions found in

```
int main()
```
[2]

located in

```
Twinscan.cpp
```

to

```
string target_sequence;
```

The sequence is transformed to be all lowercase before the c+g percentage is determined. Sequences must be longer than 500 bp and the sequence cannot be all "N." The c+g percentage is the only required parameter for the instantiation of the `SmatParser`[3] object:

```
SmatParser parameter_parser(cg_percent);
```

The c+g percentage is specified so that the correct isochore-specific parameters will be copied from the genome parameter file into

```
GenomeModel genome_model;
```

object, which is instantiated as a friend of the

```
SmatParser
```

object.

3.2 Parse the genome parameter file and instantiate a `GenomeModel` class object.

The goal of this step is to ensure that all of the information in the genome parameter file is assigned properly within the `genome_model` object. The aggregate structure of the `GenomeModel` class is shown in Figure 9 at the end of this section.

---

[2] Following the style of Lippman and Lajoie (Lippman, S.B. and J. Lajoie. 1998. *C++ Primer*. Addison-Wesley, Reading, Massachusetts.) references to filenames, class names, and other C++ code will be typeset in `Monaco` font and generally set off from the rest of the text.

[3] There are different subclasses of the `Parser` class based on whether the genome parameters are read in from HunamIso.smat (Burge, C. New GENSCAN Web Server at MIT.) or from the Brent Lab format. Both subclasses have the same function.

Parsing the parameter file is accomplished by the following SmatParser member function called from int main():

```
parameter_parser.parse_genome_para_file(genome_para_filename,
genome_model);
```

The tasks of reading the text from the parameter file are handled by methods in the Parser class:

```
void filter_text();
void seperate_words();
```

The constructor for the SmatParser object invokes a number of additional methods including:

```
set_state_type();
```

to create

```
set<string> c_state_name
set<string> d_state_name
```

Each of the 27 state-names of the model is hard-coded in one of the two sets.

```
set_command_type();
```

creates

```
map<string,int> command_map
```

a method to distinguish whether the current line is a command, i.e. one of the various sections of the parameter file. The command_map attribute is also hard-coded with the details of the parameter file.

Two generic attributes are created and destroyed several times as information is ferried from the parameter file to the genome_model object:

```
Matrix<int>* score_matrix
vector<double>* exon_length
```

Although GenomeModel is an aggregate class made up of all the details of the HMM described in Chapter 2, it has only two attributes:

```
TransitionMatrix<CState> c_states
```

```
TransitionMatrix<DState> d_states
```

The relationship among the GenomeModel, TransitionMatrix and Parser classes is shown in Figure 7.

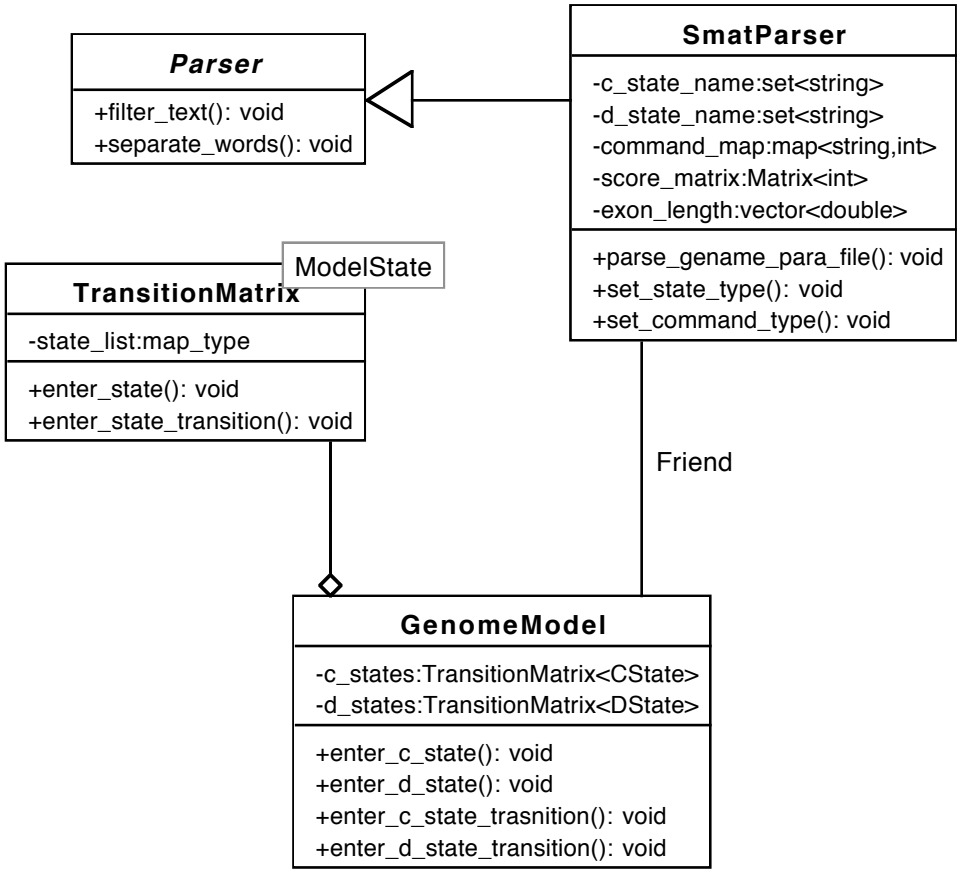

Figure 7. The relationship between SmatParser and GenomeModel.

The TransitionMatrix class instantiates the state objects and manages the interactions between them.

TransitionMatrix is a template class with the following attribute:

```
map <string, state pointer>
```

and method:

```
enter_state_transition (in_state_ptr, out_state_ptr,
transition_prob)
```

Both `State` and `TransitionMatrix` are very general and should be able to be used in any HMM application.

As shown in Figure 8, `State` is the base class of `ModelState`, which is the base class of both `CState` and `DState`.

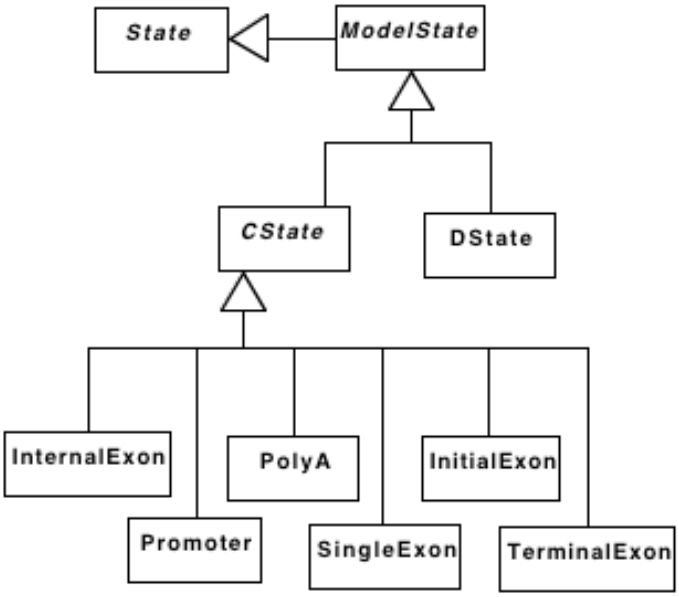

Figure 8. The `State` class heiarchy.

`State` class attributes include the state name and a list of all preceding and subsequent states with non-zero transition probabilities:

```
string state_name
state_list prev_states;
state_list post_states;
```

.

The `state_list` type is implemented through an internally defined member-class of `State`:

`StatePair`.

which has one attribute to store a pointer to a state and a transition probability:

```
class StatePair: private pair<State*, double> {
```

Although `ModelState` adds only a small amount of functionality

```
bool forward_strand_direction;
```

it is the first place in the `State` class hierarchy that include attributes specific for biological sequence analysis.

The `DState` class contains the parameters associated with the d-states. Recall that d-states are modeled with geometric distributions, so only a single parameter is required for each states and is used to determine probabilities and scores as follows:

```
double intron_length;          // L: Mean Length of the D state;
double intron_continue_prob;   // L/(L-1): Real Prob Value;
double intron_continue_score; // 10*log2(L/(L-1)): Corresponding
Score;
double intron_stop_prob;       // 1/L: Real Prob Value;
double intron_stop_score;      // 10*log2(1/L): Corresponding Score;
```

Thus, if the state has length N, the probability of continuing in that state is N/(N+1) and the probability of ending the state is 1/(N+1)

Note: The content model (i.e the null model) used in the intergenic and intron states in not explicitly stated in the parameter file. Only the log ratios (i.e. the scores) of the d-states are known. The probabilities are not known.

`CState` class is the base class for a number of specialized objects that represent the various c-states in the Genscan prediction model. These include the following:

```
class InitialExon : public CState{
class TerminalExon : public CState{
class InternalExon : public CState{
```

```
class SingleExon : public CState{
class Promoter: public CState{
class PolyA: public CState{
```

Although an abstract class, `CState` is one of the largest classes. Almost all of the functionality of its subclasses is contained within `Cstate`. It defines most of the probability models associated with the program. The various probability models in `CState` work together with the models defined in `SpsConsModel` to determine the total exon score, which the key value used by the Viterbi algorithm. The models are described in detail above and their function names are briefly noted here:

```
int wmm_model(Matrix<int>*, const string&, const int&, const int&);
int third_order_wam(Matrix<int>*, const string&, const int&, const
int&);
int markov_coding_region_score(const string&, const int&, const
int&);
int donor_score(const string& s, const int& pos);
```

Related functions may pull together several models to report the score associated with a single biological signal:

```
int signal_peptide_score(const string&);
int acceptor_score(const string& s, const int& pos);
int trans_init_score(const string& s, const int& pos);
int trans_term_score(const string& s, const int& pos);
```

The following two `CState` virtual functions take as input a pointer to the target sequence and a position within the sequence:

```
virtual int begin_score(const string& s, const int& pos) {
virtual int end_score(const string& s, const int& pos) {
```

An addition virtual function takes sequence start and end points as well as the sequence itself:

```
virtual int  content_score(const string&, const int&, const
int&) {
```

These functions are redefined in all of the subclasses of `CState` to return scores from the appropriate model. For example in the `InternalExon` class:

```
int begin_score(const string& s, const int& pos) {
   return acceptor_score(s, pos) ;
}

int end_score(const string& s, const int& pos) {
   return donor_score(s,pos);
}
int content_score(const string& s, const int& start_pos,
const int& end_pos)  {
   return markov_coding_region_score(s, start_pos+6, end_pos-3);
};
```

`CState` also keeps track of the reading frame, which is, for the purposes of the program, defined as the intron that each exon state transitions to. The reading frame for any given exon is determined by the previous intron. The initial exon has a reading frame defined as zero. Given both the identity of a c-state and its length, unique preceding and following d-states con be determined. The following functions return these states:

```
string& prev_d_state_name(const int& length);
string& post_d_state_name(const int& length);
```

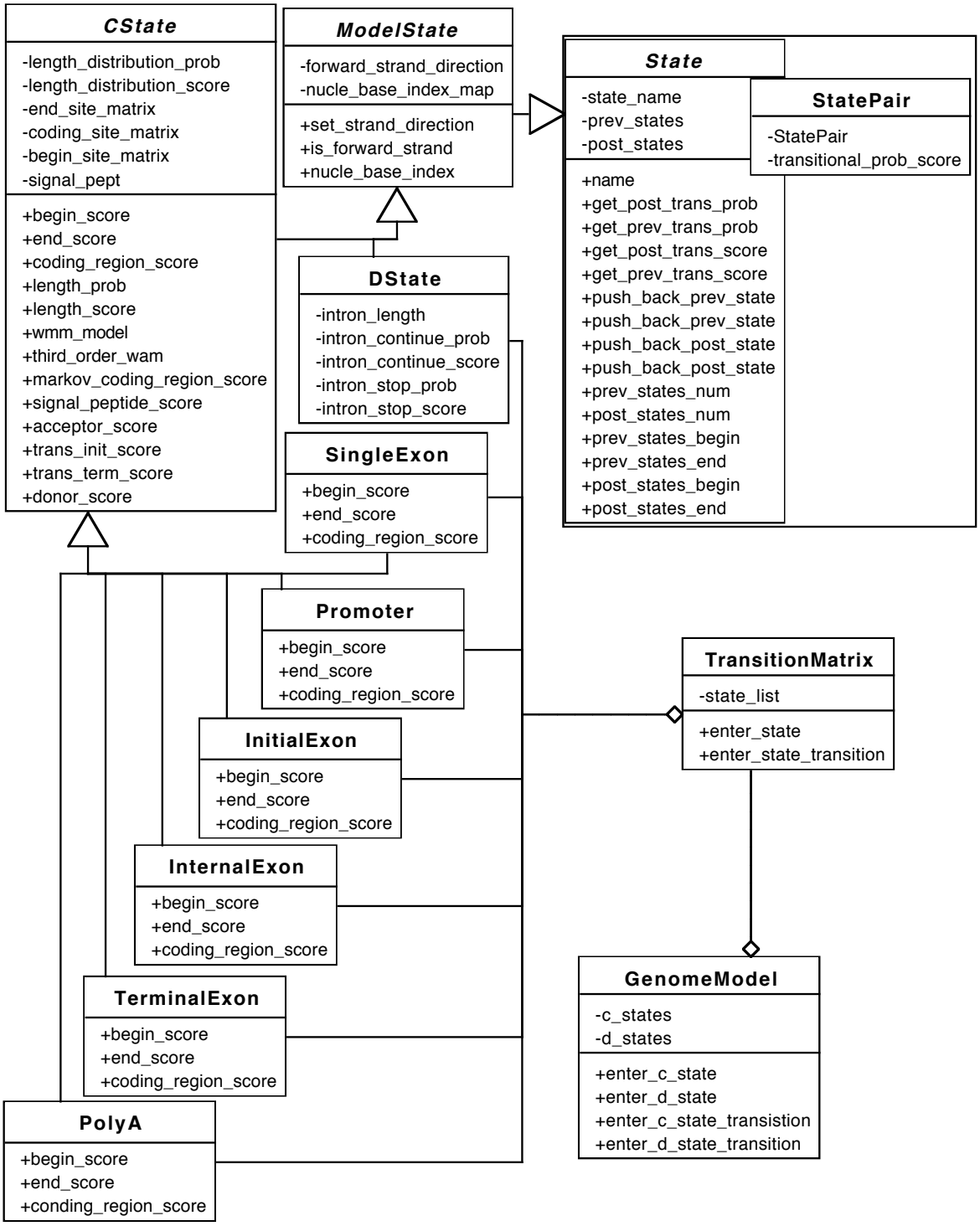

Figure 9. The aggregate structure of the `GenomeModel` class.

3.3 Read in the Conservation Sequence

The conservation sequence is read in by `int main()` to

```
string cons_sequence;
```

The conservation sequence is a one-line text file. Twinscan will exit

```
if (cons_sequence.size() != target_sequence.size()) {
```

3.4 Parse the Conservation Parameter File and Instantiate an `SpsConsModel` Class Object

The `SpsConsModel` class object is instantiated within `int main()`:

```
SpsConsModel* sps_cons_model = new SpsConsModel(cons_para_filename);
```

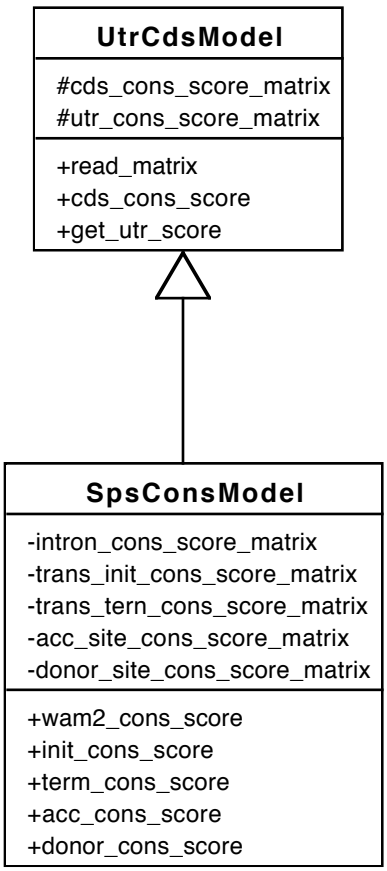

Figure 10. The conservation model classes

`UtrCdsModel` is the base class of `SpsConsModel` and it contains the methods required to parse the conservation parameter file and to organize the data within `sps_cons_model`.

The component models associated with the conservation sequence are pre-defined (see Section 2.5) and the size of each of the parameter matrices is known at compile time from the value of `CON_BITS`. This allows the matrices of the conservation parameters to be read in a far more straightforward way than the genome parameters. For example:

```
cds_cons_score_matrix = new Matrix<int>
(size, CON_BITS) ;
utr_cons_score_matrix = new Matrix<int>
(size, CON_BITS) ;
```

allocate memory for the parameter matrices associated with the coding sequence and untranslated region

conservation models, while the following function calls read the data from the file:

```
read_matrix ("UtrConsScore", *utr_cons_score_matrix);
read_matrix ("CdsConsScore", *cds_cons_score_matrix);
```

Note that since the `command_map` data structure is not used, the order of the parameter matrices is more important with the conservation parameters than with the genome parameters.

3.5 Instantiate the Appropriate `Trellis` Class Object

Figure 11 displays the `Trellis` class hierarchy. The `NonConsViterbi` class is a subclass of the `ModelTrellis` class, which itself is a subclass of `Trellis` class. The trellis associated with the full conservation model, `SpsConsViterbi`, is a subclass of `UtrCdsViterbi`, which is a subclass of `NonConsViterbi`.

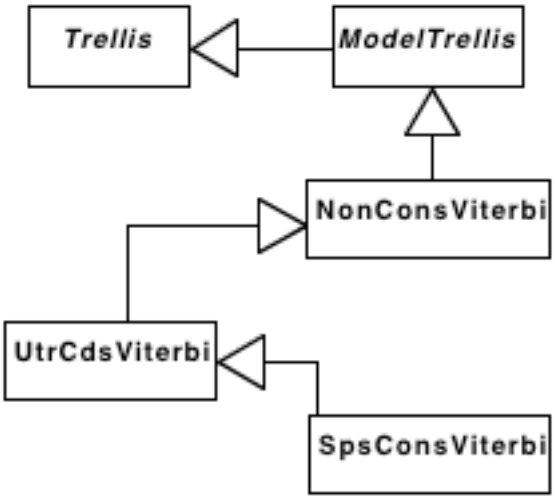

Figure 11. The structure of the `Trellis` class.

As in the `State` class hierarchy, the `Trellis` base class is very general and could be used for any HMM application. However, unlike the `State` class hierarchy, `ModelTrellis` has tremendous functional importance. It is the major computational worker in the code. The methods and attributes that are described below are used for the implementation of the full conservation model. The specific class where each method or attribute resides will be noted in the discussion. A more detailed diagram follows the discussion.

The trellis is composed of `Cell` class objects and the base `Trellis` class contains the data structure of the trellis itself:

```
template<class CellType>
class Trellis {
private:
   Matrix<CellType>  score_matrix;
```

The size of the trellis is 11 times the length of the sequence because only the d-states are part of the trellis.

The `Trellis` hierarchy is closely tied to the `Cell` hierarchy. Like `Trellis` and `State`, the base `Cell` class is very general and could be used with any HMM application. `ModelCell` is the first place in the hierarchy that includes data structures or methods related to biological sequence analysis. `ViterbiCell` is designed to interact with any of the three Viterbi classes for the `Trellis` hierarchy. See Section 4.1 for detailed information about the `Cell` class hierarchy.

`ModelTrellis` data structures:

The most import data structure is

```
exon_vec *forward_exon_info;
```

a composite type to store the details for all of the possible exons. The `exon_vec` is defined as follows:

```
typedef list<ExonInfo*>    exon_list;
typedef vector<exon_list*> exon_vec;
```

An `ExonInfo` object exists for every possible exon. They are created during the preprocessing of the sequence and used by the Viterbi algorithm. The key attribute of ExonInfo is

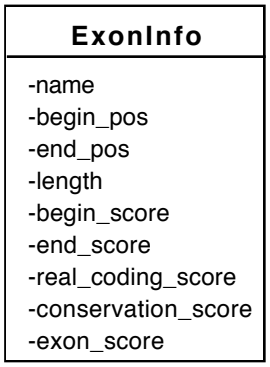

Figure 12. The `ExonInfo` class

exon_score. The score calculation contains several parts and, for the full conservation model, is calculated in several places. The following is located in the ModelTrellis class and calculates the portion of the score from the DNA model:

```
exon_score = state->length_score(exon_length) +
      prev_intron_stop_score +
      trans_score –
      exon_length * intergenic_continue_score +
      real_coding_score;

if (state->exon_name() == "P")
   exon_score += forward_promoter->real_promoter_score(
      end_score, begin_score);

else
   exon_score += begin_score + end_score;
```

The score is modified by the real_exon_score() function in the SpsConsViterbi class as follows:

```
new_exon_score = noncons_exon_score + conservation_score;
exon_info->set_exon_score(new_exon_score);
```

The exon_score and all of the other information related to the possible exon is stored at the downstream sequence index (with respect to the forward strand of the input sequence). For example, the information from a possible terminal exon on the forward strand is stored at the final base of the stop codon, and a possible terminal exon on the reverse strand is stored at the coding base immediately preceding the acceptor splice site.

```
map<CState*, list<int>> exon_begin_pos
```

stores locations of current possible exon begin positions (i.e. locations of AG or ATG sequences). If in-frame stop codons appear, possible exon start positions will be removed from this list. The procedure to remove no-longer-viable exons means that exon_begin_pos never gets very large.

```
map<CState*, ivec*>  *begin_score_saved; //C-state begin scores
```

stores the scores associated with the beginning of the c-states. For the cases of the exons, the begin score is either the accepter site score or the translation initiation score. In the case of the promoter and the Poly A, the begin score is defined as zero. Exon begin scores are stored whenever an AG or ATG appears in the sequence. Promoter and Poly A scores are calculated and stored at every position in the sequence from the `end_score` function. For example, in the following sequence:

NNNNAGNNNNNNAGNNNNNNAGNNNNN

The `begin_score_saved [E0+]` vector will have three elements (one at each start), which will have begin score at each appropriate position.

```
map<CState*, ivec*>  *end_score_saved;   //C-state end scores
```

stores the scores associated with the ending of the c-states. For the case of the exons, the end score is either the donor site score or the translation termination score. In the case of the promoter the end score is

```
int end_score(const string& s, const int& pos) {
   if (pos < CAPSITE_BEGIN_BOUND || pos > (int)s.size()-
         CAPSITE_END_BOUND)
      return ZERO_PROB;
   return  wmm_model(end_site_matrix,s.substr(pos-2,8),0,7);
}
```

For the poly A, the end score is:

```
int end_score(const string& s, const int& pos) {
   if (pos<POLYA_BEGIN_BOUND)
      return ZERO_PROB;
   return wmm_model(coding_site_matrix,s.substr(pos-5,6),  0, 5) ;
}
```

Exon end scores are stored whenever a GT or stop codon appears in the sequence. Promoter and Poly A scores are stored at every position in the sequence.

```
Matrix<bool> *term_exon_begin
```

is used to determine whether a possible terminal exon is valid. This structure prevents the inclusion of terminal exons with in-frame stop codons.

```
vector<bool> *sngl_exon_begin
```

is used to keep track of the whether a possible single exon gene is valid (does not have an in-frame stop codon). Because all single exons have phase 0 by definition, a vector can be used rather than a 3-row matrix as is used with `term_exon_begin`.

```
Matrix<int> *coding_score_saved
```

is three times the size of the sequence and is the running fifth-order Markov coding score for each frame. The coding score for any subsequence is calculated by another function that takes as inputs the appropriate c-state and its start and end points:

```
template<class CellType>
int ModelTrellis<CellType>::restore_coding_score(
   CState* state, const int& begin_pos, const int& end_pos) {
```

The value of `coding_score_saved` at the beginning of the subsequence is subtracted from the value at the end to get the correct score.

`ModelTrellis` methods:

```
template<class CellType>
void ModelTrellis<CellType>::pre_processing(){
```

pre-computes all of the probabilities associated with the DNA sequnece including the scores all possible promoters. It also calculates the coding score at all positions for all three phases and stores this information in the data structures described above.

Note that in the following two methods the term “exon” actually refers to any c-state:

```
template<class CellType>
void ModelTrellis<CellType>::detect_reverse_exon_begin(const  int&
pos)
template<class CellType>
void ModelTrellis<CellType>::detect_reverse_exon_end(const  int& pos)
```

The forward and reverse strands are processed in two passes. This allows the code that runs the forward model to be used twice. A complicated bit of bookkeeping is required to ensure that all of the possible exons are arranged for the Viterbi algorithm.

There data structures related to scores from the conservation sequence are contained in both UtrCdsViterbi:

```
UtrCdsModel*  utr_cds_model ;
vector<int>* forward_cds_cons_score;
vector<int>* reverse_cds_cons_score;
```

and SpsConsViterbi:

```
SpsConsModel* sps_cons_model;
vector<int>*  forward_init_cons_score;
vector<int>*  reverse_init_cons_score;
vector<int>*  forward_term_cons_score;
vector<int>*  reverse_term_cons_score;
vector<int>*  forward_acc_cons_score;
vector<int>*  reverse_acc_cons_score;
vector<int>*  forward_donor_cons_score;
vector<int>*  reverse_donor_cons_score;
```

All of the scores are pre-computed by methods found in their respective classes. For example, SpsConsViterbi contains these methods

```
void pre_process_init_cons_score();
void pre_process_term_cons_score();
void pre_process_acc_cons_score();
void pre_process_donor_cons_score();
```

After the pre-processing step in which all of the possible exons are scored and stored.

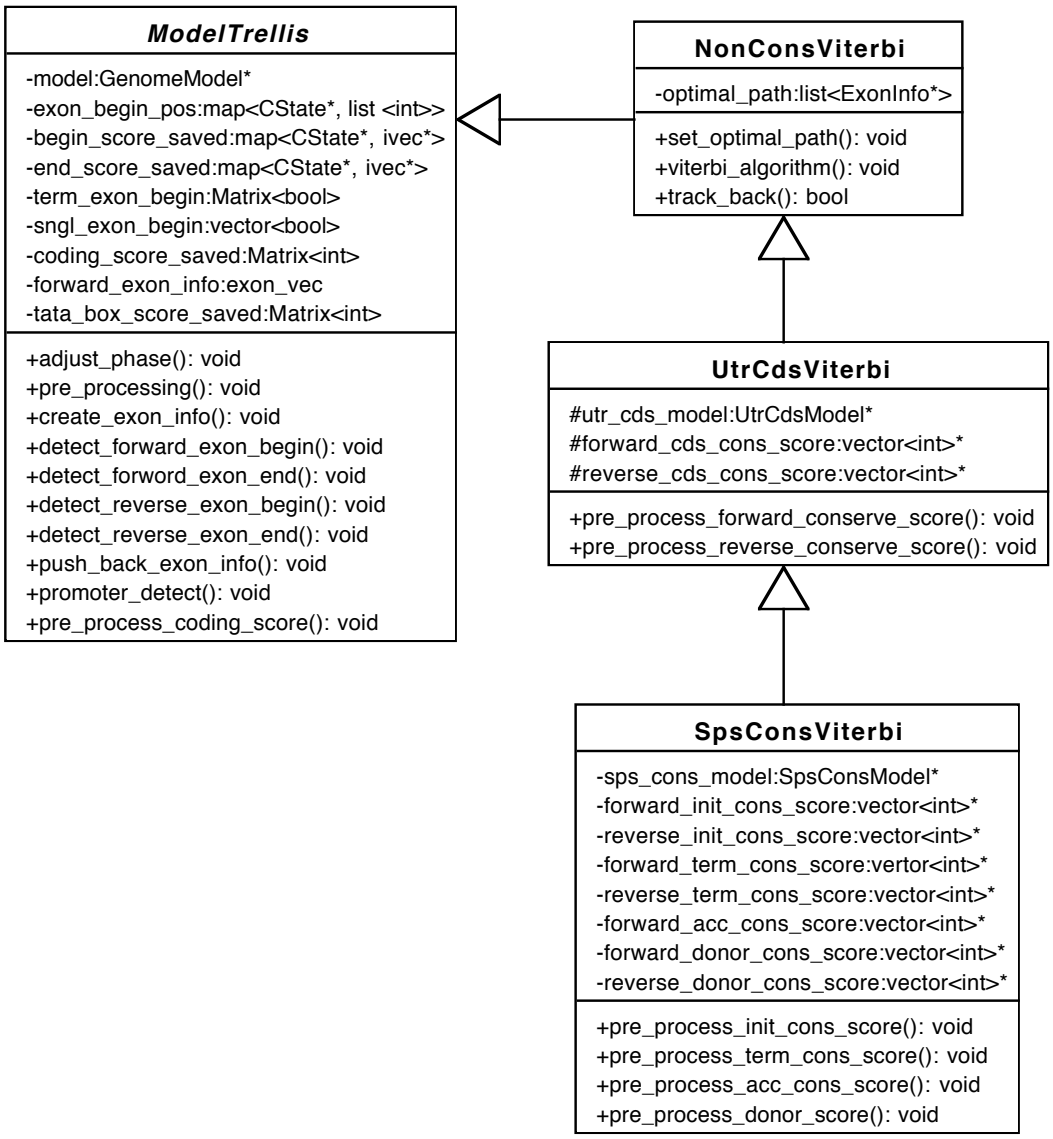

Figure 13. `ModelTrellis` and the Viterbi trellis object showing major attributes and member functions

3.6 Calculate the Most Probable Path Through the Trellis with the Viterbi Algorithm

The Viterbi algorithm is invoked from `int main()`:

```
viterbi_trellis->viterbi_algorithm(0);
```

All of the values in the trellis are calculated before `trace_back()` is called.

The complete implementation is shown here:

```
void
   NonConsViterbi::viterbi_algorithm(const int& startfrom) {
      DStateIter it = model->get_dstate_matrix().state_list_begin();
      DStateIter it_end = model->get_dstate_matrix().state_list_end();
      unsigned int i;

      // Initialization:
      if (startfrom == 0) {
         while (it != it_end) {
            set_cell(d_state_index(*it), 0, (*it)->init_score() +
               (*it)->continue_score());
            ++it;
         }
         i = 1;
      }
      else
         i = startfrom;

      //Induction :
      for ( ; i < sequence->size(); i++) {
         it=model->get_dstate_matrix().state_list_begin();
         while (it != it_end) {
            viterbi_type trellis = cal_trellis_cell(i, *it);
            set_cell(d_state_index(*it), i, trellis.first);
```

```
            set_cell_path(*it, i, trellis.second);
            ++it;
         }
      }
   }
```

Immediately following the calculation of the trellis, the `trace_back` method is called:

```
   bool good_trace = false;
   while ( ! good_trace)
      good_trace = viterbi_trellis->trace_back();
```

Because certain exons may be removed from the set of all possible exons if their addition to the optimal path creates a stop codon across a splice site, the `trace_back` method may re-invoke `viterbi_algorithm` from the point that the exon is removed:

```
      remove_exon(test_name, last_base_pos, first_base_pos);  }
   viterbi_algorithm(sequence_index);
   optimal_path.clear();
```

While the potential for computational explosion exists in the repeated calling of the Viterbi algorithm, clearing of the optimal path, and running the trace back; in practice on the whole human genome this procedure adds an insignificant amount of time.

3.7 Output the Result

Results are printed to standard out with the following

```
   cout << *viterbi_trellis << endl;
```

The output operator in this case is overloaded by a ModelTrellis method

```
   ostream&
   operator << (ostream& os, const NonConsViterbi& gvt) {
```

The output from Twinscan is designed to be “parsably-equivalent” to Genscan’s output. This format is normally converted to GTF before it is used in other analyses.

Chapter 4

Class Reference

4.0 Introduction

The class reference contains full class diagrams for all of the classes in Twinscan. These are meant to supplement the diagrams and the discussion in Chapter 3 with a more complete documentation of the class, their attributes and functions. They are presented in hierarchical order with base classes in alphabetical order. The behavior and characteristics of each class will be explained generally before presentation of attributes and operations. For those class hierarchies that have similar functionality in super- and sub-classes, explanations will only be given once. Additionally, if the general explanation makes clean the role of an attribute or function of an operation these will not be further discussed.

A number of files that are part of the total source code, but are not classes. They will be described briefly here.

`BenchMark.h`
`BenchMark.cpp`

Provides two functions used to determine the clock-time required for various portions of the code
`diffstr()`
`clockdiff()`

`Templates.cpp`
Defines the templates for the `TransitionMatrix` class.

`Twinscan.cpp`

The home of `int main()` and the central controlling routine for Twinscan. Also sets default values for all of the compiler and command line options and contains the Twinscan help text. The most important attribute is `viterbi_trellis`, a variable of type `NonConsViterbi` that is assigned the return value of `getTrellis()`, the major function in `Twinscan.cpp`.

`getTrellis()` calls the functions required to parse the parameter files, instantiate the `GenomeModel` object, and instantiate the appropriate `Trellis` class object.

checks that the number of arguments on the command line are appropriate to the model type selected.

measures the size of the target sequence and allocates the memory for the sequence in advance of reading in the data.

opens the target sequence file, discards the first line of the fasta file and reads the DNA sequence.

determines the cg percentage.

checks to make sure that the size of the target sequence is greater than 500 bp.

reads the first line of the parameter file to determine the source of the parameter file and instantiates the appropriate `Parser` class object.

calls `parse_genome_para_file()` from the appropriate `Parser` subclass.

reads and parses the conservation sequence and conservation parameter file, if the conservation model is used.

calls the constructor for the appropriate `Trellis` class and returns its value.

Once `getTrellis()` has returned, the following code directs the final portion of the analysis.

```
viterbi_trellis->viterbi_algorithm(0);
bool good_trace = false;
while ( ! good_trace)
   good_trace = viterbi_trellis->trace_back();
cout << *viterbi_trellis << endl;
```

4.1 Cell

Files: Cell.h

Parent of: ModelCell, ViterbiCell

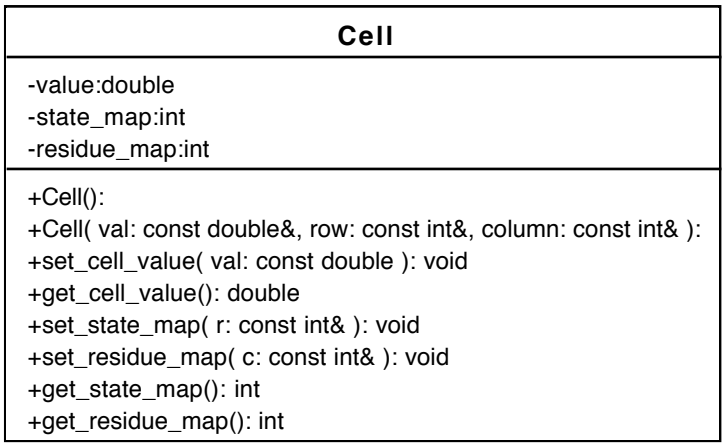

Figure 14. The `Cell` Class

`Cell` is the base unit for the trellis. A `Trellis` class object is made up of `Cell` class objects. Each `Cell` object is aware of its location in the trellis and knows its value. The interface allows for the location and value to be changed and reported.

Attributes:

`value`	cell value
`state_map`	row location in the trellis. In Twinscan there are 11 rows (d-states).
`residue_map`	column location in the trellis. This is the length of the sequence.

Operations:

`set_*`	assigns argument to the appropriate attribute.
`get_*`	returns the value of respective attribute.

Constructor:

`Cell()`	all three values can be initialized to zero or set to any value through the two constructor functions.

4.1.1 ModelCell

Files: ModelCell.h

Inherits from: Cell

Parent of: ViterbiCell

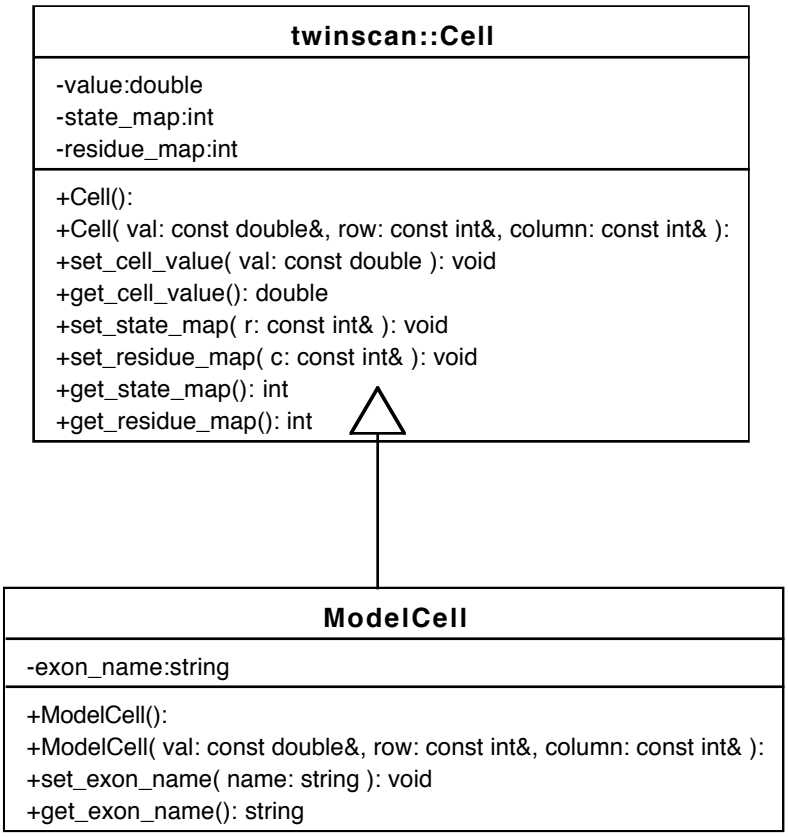

Figure 15. The `ModelCell` Class

`ModelCell` inherits almost all of its behavior from `Cell`. It includes only an exon name and the ability to set the name and return the value of the name as its interface.

Constructor:

`ModelCell()` when called, initializes `value`, `state_map`, and `residue_map` in `Cell`.

4.1.1.1 ViterbiCell

Files: ViterbiCell.h

Inherits from: Cell, ModelCell

Used by: ViterbiTrellis

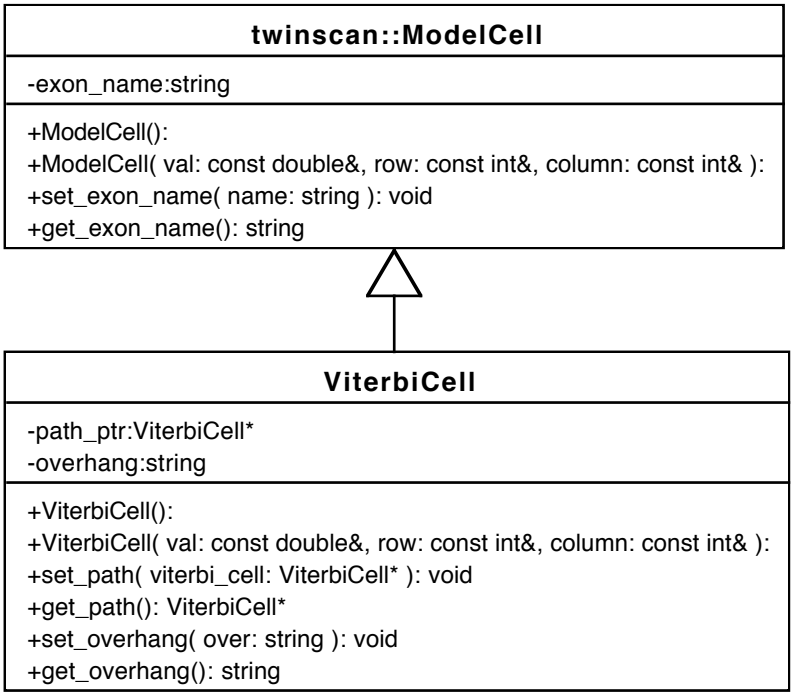

Figure 16. The `ViterbiCell` Class

`ViterbiCell` is used by the three subclasses of `VirterbiTrellis`: `NonConsViterbi`, `UtrCdsViterbi`, and `SpsConsViterbi`. Like `ModelCell` it inherits most of its functionality. It adds two functional pieces, only one of which is actually used.

`path_ptr` a pointer to another `ViterbiCell` class object.

`overhang` not used.

Constructor:

`ViterbiCell()` calls the `ModelCell()` constructor and initializes `path_ptr` to 0.

4.2 ExonInfo

Files: ExonInfo.H

Used by: ModelTrellis to store all possible exons, NonConsViterbi to store optimal parse.

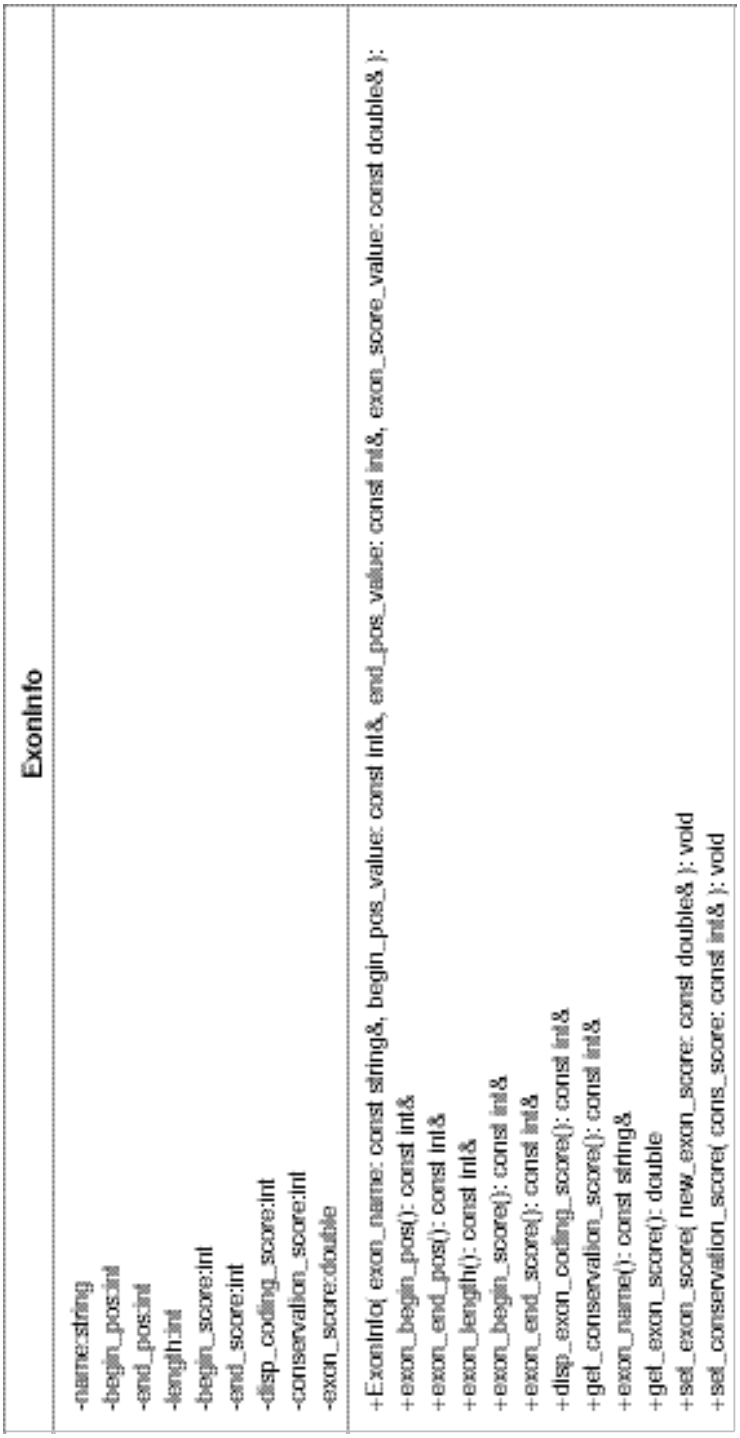

Figure 17. The `ExonInfo` Class

The ExonInfo class stores the relevant information about the possible and predicted exons. It is the key part of the exon_vec data structure: a length of the sequence vector that is composed of a list ExonInfo objects at each position. The exon_vec data structure is defined in the ModelTrellis class and populated by all possible exons (see Section 2.7). It this case exon name, begin position, end position and total exon score are stored.

A more expressive instance of ExonInfo is used to store the optimal path (i.e. exons identified by NonConsViterbi::traceback()). In addition to the above attributes, the signal scores, the coding score, and the conservation score are stored. The more expressive representation is primarily used for the output of the final gene prediction.

Attributes:

name	state name.
begin_pos	defined as the five prime end of the exon (lower index on the forward strand and higher index on the reverse strand). Note that all internal positional references take the first base of the sequence to have index 0. Annotation representations define the first base in the sequence as having index 1.
end_pos	the three prime end of the sequence.
length	one end point minus the other plus 1.
begin_score	the score associated with the signal model at the begin_pos end of the exon. For example, on the forward strand, begin_score for an initial exon is the start of translation signal; for internal exons begin_score is the acceptor splice site score.
end_score	the score associated with the end_pos end of the exon.
disp_coding_score	the coding score that is displayed in the output. Comes from the 5th-order Markov chain model. (See section 2.3 for more information).
conservation_score	the conservation score from the coding sequence model only.

	The other aspects of the coding score are included in exon_score, but not here. This value is displayed in the output.
exon_score	the total score for the exon. See Figure 5 for the source of the score in exons.

Operations:

The following functions return the appropriate attribute. All values are set by one of the two ExonInfo constructors.

exon_begin_pos()

exon_end_pos()

exon_begin_score()

exon_end_score()

disp_coding_score()

get_conservation_score()

exon_name()

get_exon_score()

There are two operations able to modify attributes; both reassign their respective attributes with the passed value.

set_exon_score()

set_conservation_score()

Constructors:

```
ExonInfo(const string& exon_name, const int& begin_pos_value,
         const int& end_pos_value, const double& exon_score_value)
```

Called from ModelTrellis for storing all possible exons in the exon_vec data structure.

```
ExonInfo(const string& exon_name, const int& begin_pos_value,
         const int& end_pos_value, const int& begin_score_value,
         const int&  end_score_value,
         const int& coding_score_value_1,
```

```
        const int& conservation_score_value,
        const double& exon_score_value)
```

Called from `NonConsViterbi` for storing the optimal parse in `NonsConsViterbi::optimal_path`.

4.3 GenomeModel

Files: GenomeModel.h, GenscanModel,cpp

GenomeModel
+SmatParser:friend class -c_states:TransitionMatrix<CState> -d_states:TransitionMatrix<DState>
+GenomeModel(): +get_dstate_matrix(): TransitionMatrix<DState>& +get_cstate_matrix(): TransitionMatrix<CState>& +get_c_state(name: const string&): CState* +get_d_state(name: const string&): DState* +enter_c_state(name: const string&): void +enter_d_state(name: const string&): void +enter_c_state_transition(in_state_name: const string&, out_state_name: const string&, trans_prob: const double&): void +enter_d_state_transition(in_state_name: const string&, out_state_name: const string&, trans_prob: const double&): void +d_state_number(): int +c_state_number(): int

Figure 18. The `GenomeModel` Class

`GenomeModel` is the large aggregate class that includes the structure and details of the model for the DNA sequence. See Figure 9 and Section 3.2 for more information about the structure of a `GenomeModel` object. The attributes are two instances of the `TransitionMatrix` class, one for the c-states and one for the d-states. The methods of `GenomeModel` allow for access of the states in the model by the objects from the `Trellis` class hierarchy as well as their initialization and creation by the `Parser` class objects. The parsers are friend classes to `GenomeModel`. `GenomeModel` is independent of the topology of the model and of the models of the states. However, it does require the concept of two kinds of states.

Although functions from the `TransitionMatrix` class manage the list of states in the model and the allowed transitions between them, each `State` class object contains its own models and allowed transitions.

Operations exist that return all of the states (`get_cstate_matrix()` and `get_dstate_matrix()`) or just one state (`get_c_state()` and `get_dstate()`).

Other operations:

`enter_c_state_transition()`

`enter_d_state_transition()` are called by the `Parser` class objects. Both work in approximately the same way:

As state information is parsed from the parameter file, the function appropriate to the current state is called with the state name, a transition probably to another state, and the name of that following state. These functions call both `enter_c_state()` and `enter_d_state()` (described below) with either the current state of the following state, depending on whether the current state is a c-state or a d-state.

Once the states have been instantiated, the `TransitionMatrix::enter_state_transition()` function is called to store the transition probability.

`enter_c_state()` takes as its argument the name of the state, checks to see if that state object exists and, if not, instantiates the appropriate c-state sub-class (e.g. `InitialExon` or `PolyA`) by calling the `TransitionMatrix::enter_state()` function.

`enter_d_state()` takes as its argument the name of the state, checks to see if that state object exists and, if not, instantiates a `DState` object by calling the `TransitionMatrix::enter_state()` function.

`d_state_number()` returns the number of d-states in the model. This is used in trellis management.

`c_state_number()` returns the number of c-states in the model. This function is not used.

4.4 Matrix

Files: Matrix.h

Used by: Trellis, SmatParser, ParaParser, CState, ModelTrellis, UtrCdsModel, SpsConsModel

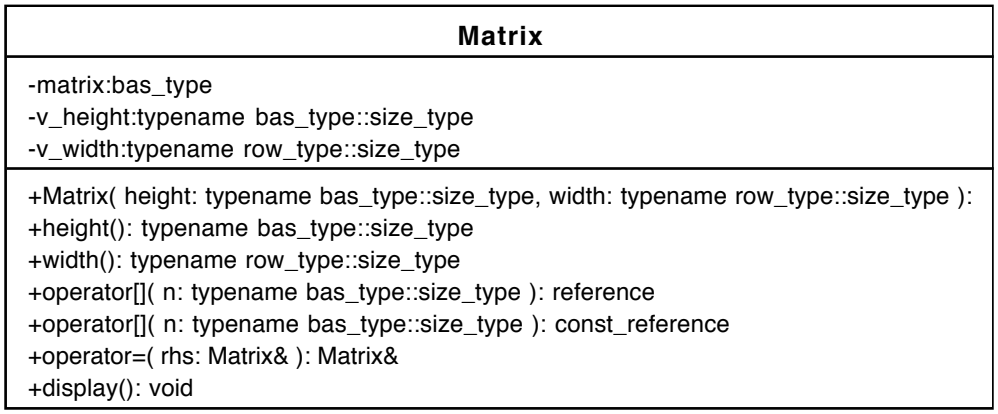

Figure 19. The `Matrix` Class

`Matrix` is a library class used in various ways by Twinscan. As a template class, it stores everything from integers in parameter score matrices to the `Cell` class objects that make up the `Trellis` class object. Individual values or objects in the `Matrix` data structure can be accessed using the following syntax (see note below):

```
matrix[i][j]
```

Attributes include the matrix itself as well as its dimensions.

Operations allow for the return of the height and width of the matrix, the access of the data, and the ability to determine if two `Matrix` class objects are the same.

IMPORTANT NOTE:

`Matrix` is implemented "backwards" in so far as standard mathematical notation has the subscript for the row before the column as in Matrix(row,column). In this implementation, the subscripts are Matrix(column,row). Because of this all of the methods and attributes in Twinscan that use or interact with `Matrix` have this "backward" character. It's a feature, really.

4.5 Parser

Files: Parser.h, Parser.cpp

Parent of: ParaParser, SmatParser

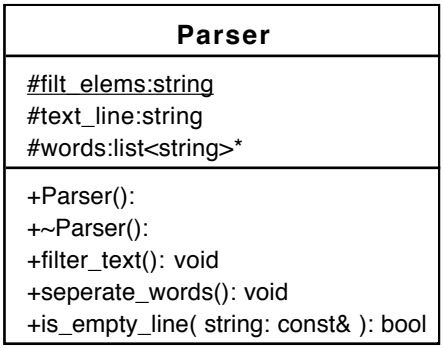

Figure 20. The `Parser` Class

`Parser` is an abstract class that provides basic text processing functionality for use by its subclasses.

Attributes:

`filt_elems`	characters that are filtered from parsed text. They are ",;:[]<>{}()
`text_line`	the current line from the parameter file.
`words`	a list the data or text created from `text_line`.

Operations:

`filter_text()`	removes by `string::erase()` characters from `filt_elems`.
`separate_words()`	separates `text_line` at spaces or tabs and stores result in words.
`is_empty_line()`	allows skipping of blank lines.

Constructor/Destructor:

`Parser()`	allocates memory for `words`.
`~Parser()`	garbage collection for `words`.

4.5.1 ParaParser

Files: ParaParser.h, ParaParser.cpp

Inherits from: Parser

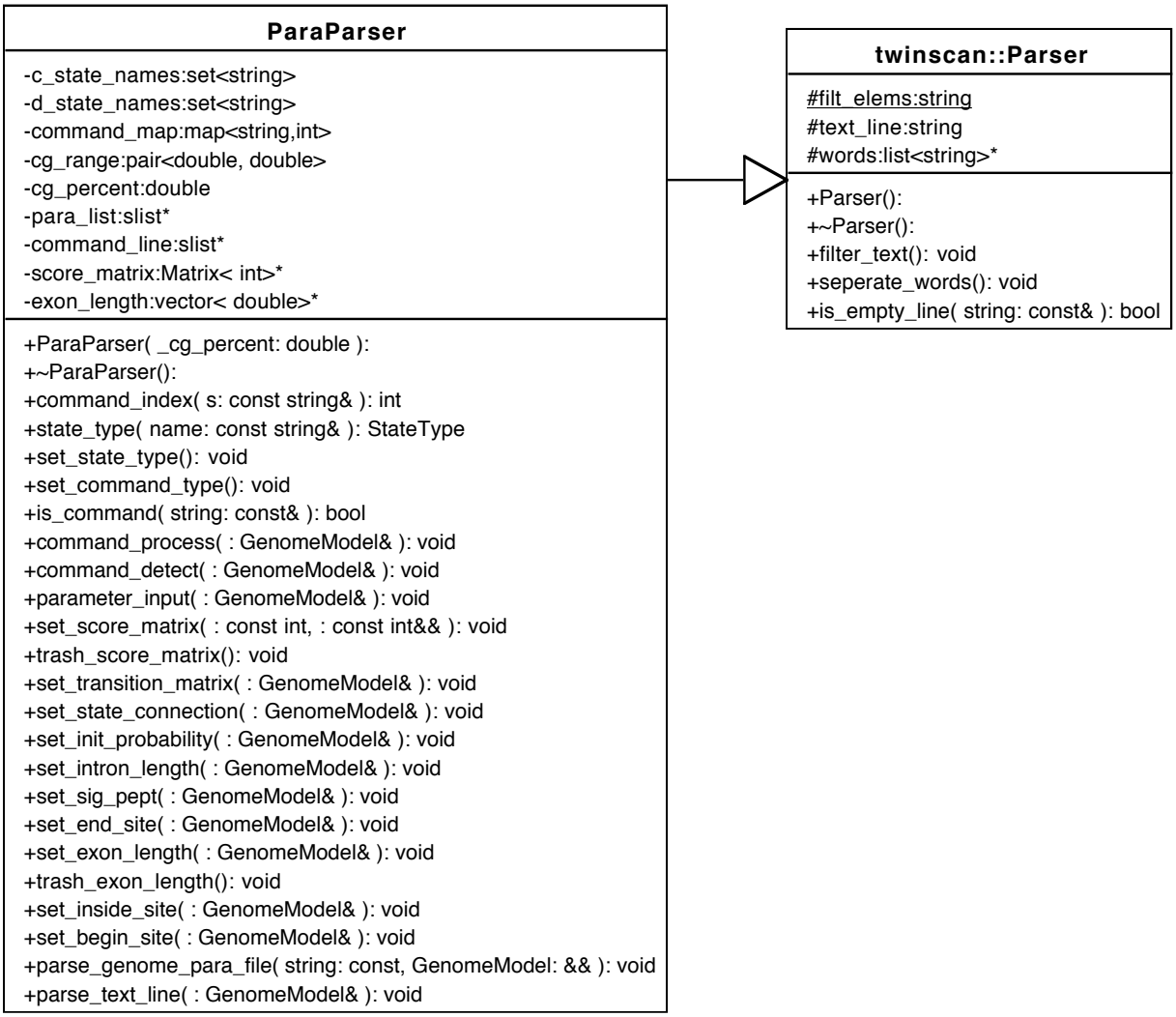

Figure 21. The `ParaParser` Class

`ParaParser` is used to parse the genome parameter files created by the Brent Lab Group and ensure that the parameter data is stored in the correct structure within the `GenomeModel` class object. For example, the acceptor splice site model parameters are stored by `begin_site_matrix`, an inherited `CState` attribute, in `InternalExon` and `TerminalExon`. `ParaParser` implicitly assumes the topography of the HMM (see

Section 2.2) and contains a number of other hard-coded features that inhibit the flexibility of the rest of the code.

Parsing involves the concept of commands—header lines set off from the rest of the file with angle brackets (<>)—that define a section of the file and, if necessary, the corresponding c-states. For example, the command line for the translation initiation model is

```
<StartCodon      [12      4]       Einit+  Esngl+  Einit-  Esngl->
```

Reading across the command line the information in a command line is the model name, size of the model, relative location of the first base in the model to the first base in the sequence, and the c-states that use the model.

Commands are nested for the case of parameters that vary based on the CG percentage of the target sequence. The commands used and their functions are

`TransitionMatrix`	defines the transitions between all of the states. The format for these lines is `current state (transition probability next state).`
`InitialProbability`	the initial probability value for each of the d-states.
`IntronLength`	the geometric length distribution parameters for the d-states.
`DonorSite`	parameters associated with the MDD model of the donor site (see Section 2.3.4).
`ExonLength`	matrix of probability values for the four exon states. Each of the values corresponds to the length of a codon.
`CodingRegion`	parameters associated with the 5th-order Markov chain.
`AcceptorRegion`	parameters for the WAM model of the acceptor splice site.
`PolyASignal`	parameters for the WMM of the poly A signal.
`StartCodon`	parameters for the WMM model of the translation initiation signal.
`StopCodon`	parameters for the WMM model of the translation termitation signal.

TATA	parameters for the WMM model of the TATA portion of the promoter model.
PB_CAP	parameters for the WMM model of the cap portion of the promoter model.
C+G	defines the CG percentage range for the parameters, i.e. <C+G 57.0 100.0>.
End	the end of a section. Always used with a section name. i.e. <End C+G> and <End TransitionMatrix>.
SigPept	parameters for the WMM model of the signal peptide (see Section 2.3.2).

Figure 22 shows the state diagram for the ParaParser object. Most of the time is spent in the Parsing state, which is shown in much more detail in Figure 23.

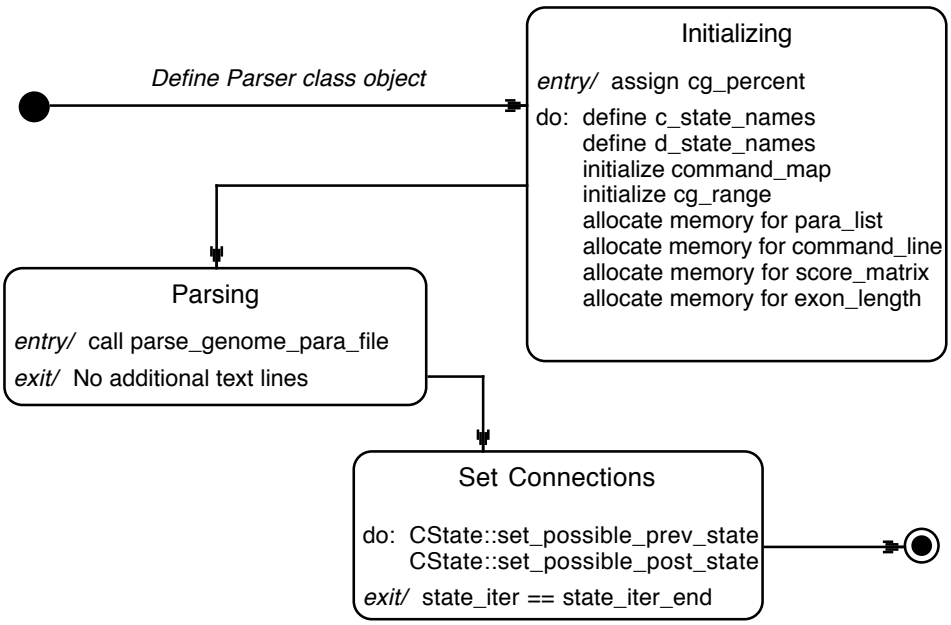

Figure 22. State Diagram for ParaParser object

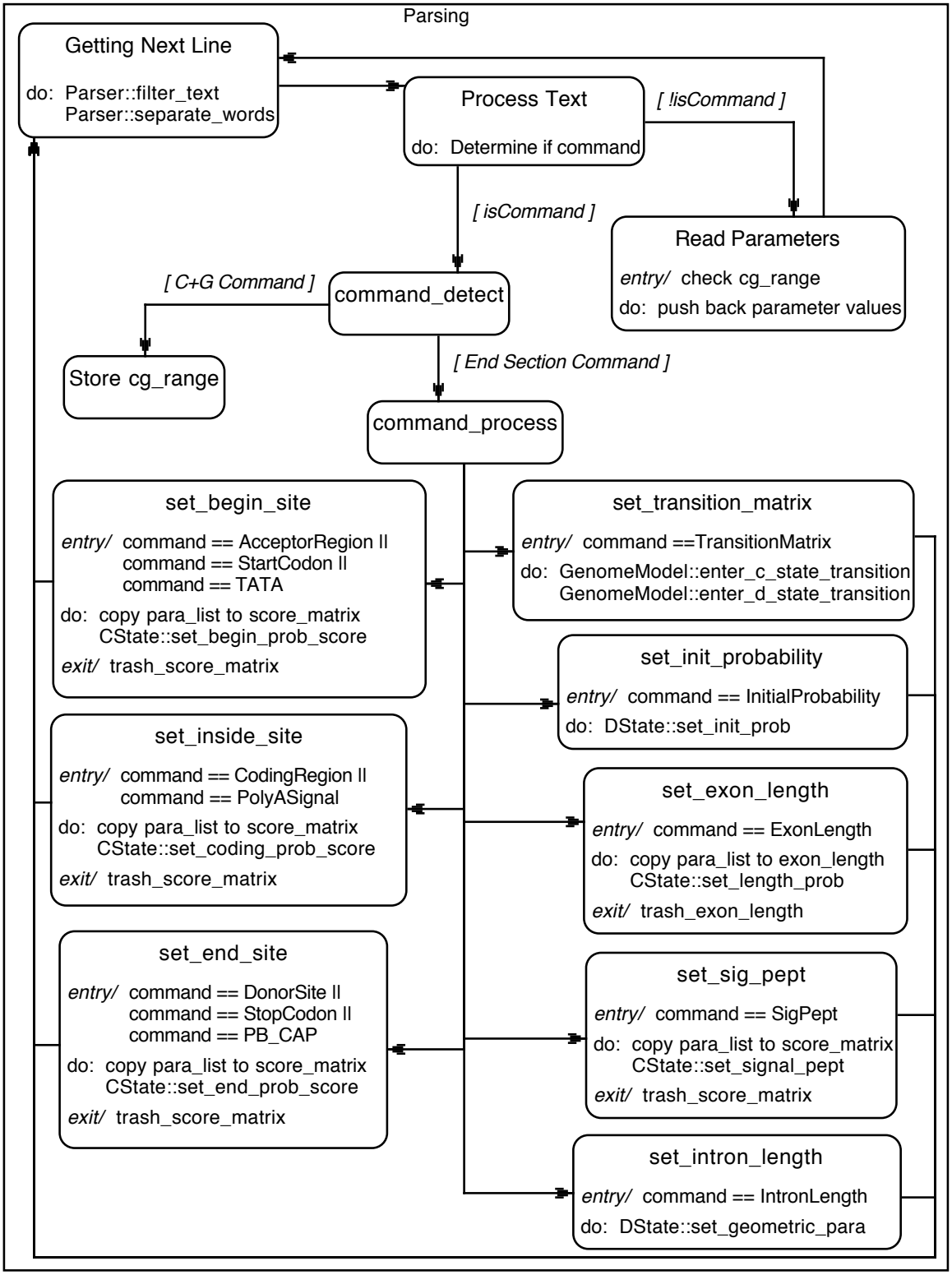

Figure 23. State Diagram for Parsing state showing substates

Attributes:

c_state_names	an stl::set of strings for storing c-state names.
d_state_names	an stl::set of strings for storing d-state names.
command_map	an stl::map from a command string to an integer.
cg_range	an stl::pair for storing the boundaries for the C+G% range of the current section of parameter file.
cg_percent	the C+G percentage of the target sequence.
para_list	a list of strings for temporary storage of the parameter values read from the parameter file.
command_line	a list of strings for storage of beginning command line of the current section of the parameter file.
score_matrix	a Matrix class object for temporary storage of the parameter matrices.
exon_length	a vector for temporary storage of the exon length probability values.

Operations:

set_state_type()	initializes c_state_names and d_state_names with the names of the c-states and the d-states. This is done by creating a string array with the state names and inserting the array elements into the set.
set_command_map()	initializes the mapping (i.e. command_map) from the 15 commands to integers.
parse_genome_para_file()	opens the parameter file and while lines remain in the file, calls Parser::filter_text(), Parser::separate_words(), and parse_text_line() for each line. When the parameter file has been completely read, calls set_state_connection().
parse_text_line()	calls command_detect() if return value of is_command() is true. Calls para_input() if return

	value of `is_command()` is false.
`is_command()`	returns true if parameter is an element of `command_map` and false if not.
`command_detect()`	calls `command_process()` if command line contains an end command. Clears `para_list` if command line is the beginning of a section. Assigns `cg_range` at a beginning C+G command and re-initializes `cg_range` at an end C+G command.
`command_process()`	a large case statement to control which of the following functions are called. See Figure 23 for a graphical representation of this and the previous two functions.
`parameter_input()`	copies from words to `para_list` if the `cg_percent` is within the `cg_range`.
`set_transition_matrix()`	calls `GenomeModel::enter_c_state_transition()` and `GenomeModel::enter_d_state_transition()` with state and transition information from the `<TransitionMatrix>` section of the parameter file.
`set_state_connection()`	ensures that the connections between all of the states in the model are correctly set. Repeatedly calls `CState::set_possible_prev_state()` and `CState::set_possible_post_state()`. The state connections are hard coded in this function.
`set_init_probability()`	determines `state_name` and `init_prob` from `para_list`. Calls `DState::set_init_prob()` with `init_prob`.
`set_intron_length()`	determines `state_name` and `intron_length` from `para_list`. Calls `DState::set_geometric_para()` with `intron_length`.
`set_score_matrix()`	allocates memory for `score_matrix`. Copies data from `para_list` into `score_matrix`.

set_end_site()	calls set_score_matrix() with the size of the matrix. Determines the appropriate c-states from command_line and calls CState::set_end_prob_score() with score_matrix for each appropriate c-state. Calls trash_score_matrix().
set_sig_pept()	calls set_score_matrix() with the size of the matrix. Determines the appropriate c-states from command_line and calls CState::set_signal_pept() with score_matrix for each appropriate c-state. Calls trash_score_matrix().
set_exon_length()	allocates memory for exon_length and copies data from para_list to exon_length. Determines the appropriate c-states from command_line and calls CState::set_length_prob() with exon_length for each appropriate c-state. Calls trash_exon_length().
set_inside_site()	calls set_score_matrix() with the size of the matrix. Determines the appropriate c-states from command_line and calls CState::set_coding_prob() with score_matrix for each appropriate c-state. Calls trash_score_matrix().
set_begin_site()	calls set_score_matrix() with the size of the matrix. Determines the appropriate c-states from command_line and calls CState::set_begin_prob() with score_matrix for each appropriate c-state. Calls trash_score_matrix().
trash_score_matrix()	de-allocates memory for score_matrix.
trash_exon_length()	de-allocates memory for exon_length.

Constructor:

Assigns `cg_percent` the value calculated in `Twinscan.cpp`. Calls `set_state_type()` and `set_command_type()`. Initializes `cg_range`. Allocates memory for `para_list` and `command_line`. Initializes elements of `score_matrix` and `exon_length` to zero.

The following attributes are hard-coded by functions in the constructor:

`c_state_names`

`d_state_names`

`command_map`

4.5.2 SmatParser

Files: SmatParser,H, SmatParser.cpp
Inherits from: Parser
Used by: GenomeModel

`SmatParser` is used to parse Genscan's genome parameter file: `HumanIso.smat`. This class should be able to parse any parameter file constructed in the style of `HumanIso.smat`, but that functionality has not been tested and is not supported.

Like `ParaParser`, the overall goal of `SmatParser` is to harvest required information from the parameter file and put it into the appropriate `GenomeModel` data structures. However, because the Twinscan code was designed to read part of the model structure from the parameter file, a number of additional assumptions about the structure of the model are hard-coded into `SmatParser` to facilitate getting parameter information from `HumanIso.smat` to the `GenomeModel` data structures.

`SmatParser` is closely related to `ParaParser` and shares many of the same attributes and operations. Only those attributes and operations that are unique to `SmatParser` will be noted here; the reader is referred to the previous section for details on the others.

Attributes:

`format_map`	an `stl::map` from header lines in the parameter file to integers. Supplements the information in `command_map`
`transitions`	a list of `stl::string` to temporarily store state names. States names are stored in pairs to mimic the format used in the `<TransitionMatrix>` portion of the Brent Lab Group parameter file format. Then `GenomeModel::enter_c_state_transition()` and `GenomeModel::enter_d_state_transition()` are called as appropriate with consecutive elements of the list.

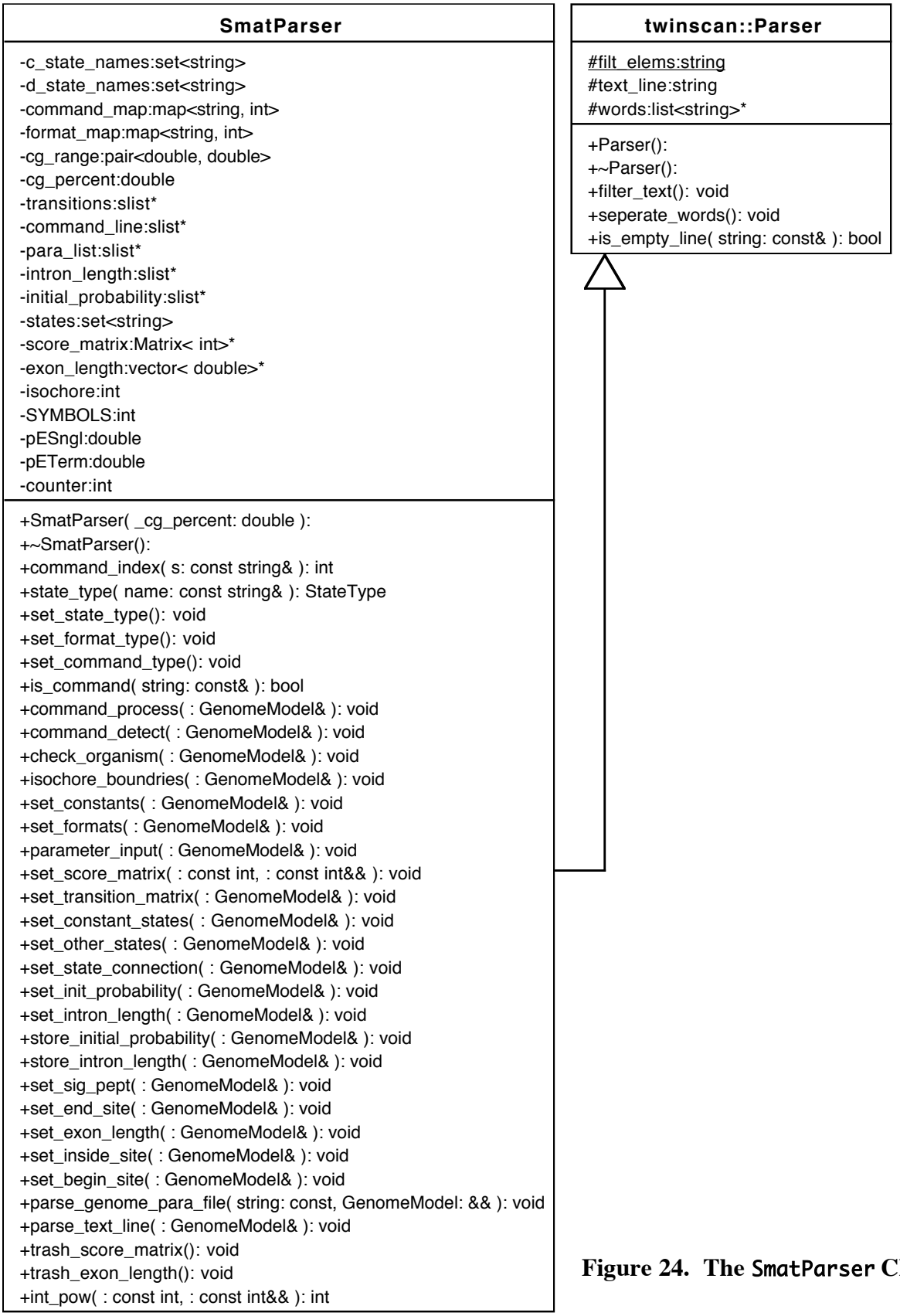

Figure 24. The SmatParser Class

intron_length	a list of stl::string to temporarily store d-state names and their associated geometric parameter values. Like transitions, intron_length stores pairs of values in successive elements. The set_intron_length() function uses the values in intron_length when making repeated calls to the Dstate::set_geometric_para() function.
initial_probability	a list of stl::string to temporarily store d-state names, an initial probability value related to that state, and two correction factors necessary to correctly compute all of the initial probability. This complicated structure is required to mimic the d-state initial probability table in the Brent Lab Group genome parameter file. All of the initial probability values are not explicitly specified in HumanIso.smat, but the values required to determine them are. Thus when all of the required data is stored in initial_probability by store_initial_probability(), the set_init_probability() function is called. Eventually the correct values are used to call Dstate::set_init_prob().
states	an stl::set of stl::string to temporarily store the c-state names associated with each parameter matrix in HunamIso.smat. This data structure is used to reproduce the list of c-states in the command lines in the Brent Lab Group genome parameter files.
isochore	an integer to store the isochore of the sequence to ensure that the isochore-dependent parameters are parsed correctly.
SYMBOLS	set to 4 (for a,c,g,t).
pESngl	transition probability value read from the parameter file
pETerm	transition probability value read from the parameter file

counter	keeps track of the number of lines since one of command_map or format_map values has been seen to signal when the entire parameter matrix has been stored. The size of each of the parameter matrices is hard coded.

Operations:

set_format_type()	a list of keywords from the header lines between different section of the HumanIso.smat file. This function is closely related to ParaParser::set_command_map().
isochore_boundries()	sets the value of isochore.
set_formats()	a large case statement. Compare to ParaParser::command_process().
check_organism()	checks that the file is actually HumanIso.smat. Twinscan does not support the use of the other parameters that come with the standard Genscan distribution (Arabopsis.smat and Maise.smat).
set_constant_states()	sets transition probabilities between those states with constant probability (e.g. 1.0) transitions through repeated calls to GenomeModel::enter_c_state_tranition() and GenomeModel::enter_d_state_transition().
set_other_states()	sets transition probabilities between those states with isochore-dependent probability distributions through repeated calls to GenomeModel::enter_c_state_tranition() and GenomeModel::enter_d_state_transition().
store_initial_probability()	stores information in initial_probability.
store_intron_length()	stores information in intron_length.

The following operations were extensively rewritten for `SmatParser` and no longer function the same as the identically named functions in `ParaParser`. The new functionality is described.

`set_command_type()`	rewritten for two stage procedure that includes both `command_map` and `format_map`.
`set_transition_matrix()`	transition matrix is taken to be known rather than partly read from the parameter file as is the case in `ParaParser`.
`set_init_probability()`	rewritten to interact with `initial_probability`.
`set_end_site()`	rewritten to use information from the FORMAT lines in `HumanIso.smat` to determine the height of `score_matrix`.

4.6 State

Files: State.h State.cpp

Parent of: ModelState, DState, CState, InitialExon, InternalExon, PolyA, Promoter, TerminalExon

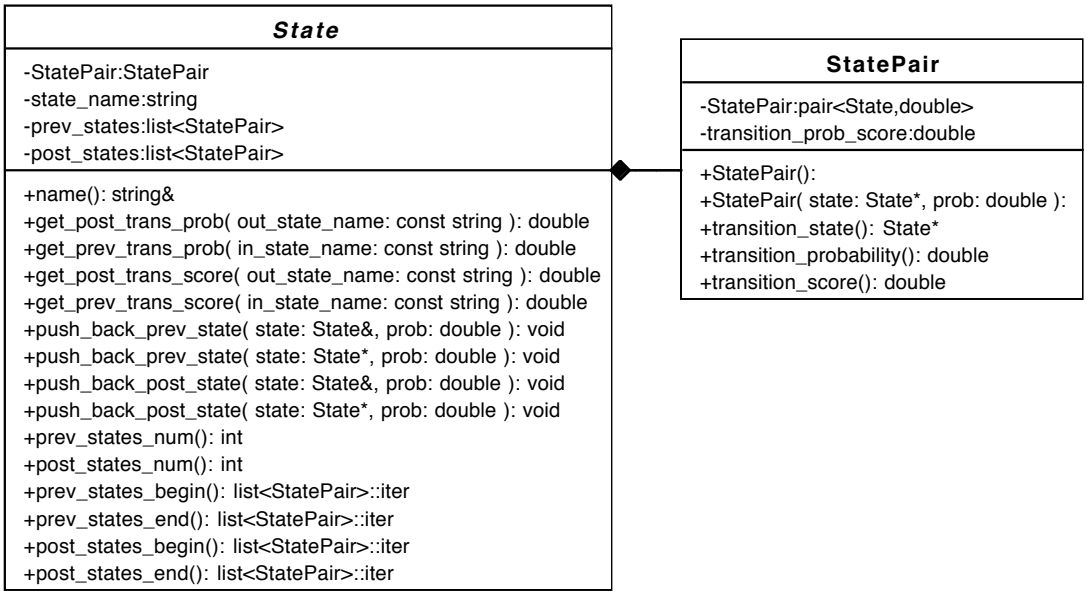

Figure 25. The `State` Class

The `State` class is the base class of all the states in the model. It contains basic functionality for defining the state and connections to other states. Within the `State` class is the member class `StatePair` data structure, a class that contains a `State` object, a transition probability, and a transition score calculated from the transition probability.

Attributes:

`prev_states`	a `list` of `StatePair` objects containing all states with non-zero transition probabilities to the `State` object.
`post_states`	a `list` of `StatePair` objects containing all states with non-zero transition probabilities from the `State` object.
`state_name`	name of the state.

Operations:

The interface for State provides for access to `prev_states` and `post_states` one element at a time.

`get_post_trans_prob()`	returns `transition_probability` from the appropriate `StatePair` object.
`get_prev_trans_prob()`	returns `transition_probability` from the appropriate `StatePair` object.
`get_post_trans_score()`	returns `transition_score` from the appropriate `StatePair` object.
`get_prev_trans_score()`	returns `transition_score` from the appropriate `StatePair` object.
`push_back_prev_state()`	adds additional `StatePair` object to `prev_states.`
`push_back_post_state()`	adds additional `StatePair` object to `post_states.`

The interface for State supports iterator access over `prev_states` and `post_states`:

`prev_states_num()`	size of the `prev_states` list.
`post_states_num()`	size of the `post_states` list.
`prev_states_begin()`	
`prev_states_end()`	
`post_states_begin()`	
`post_states_end()`	

4.6.1 ModelState

Files: ModelState.h, ModelState.cpp

Inherits from: State

Parent of: DState, CState, InitialExon, InternalExon, PolyA, Promoter, SingleExon, TerminalExon

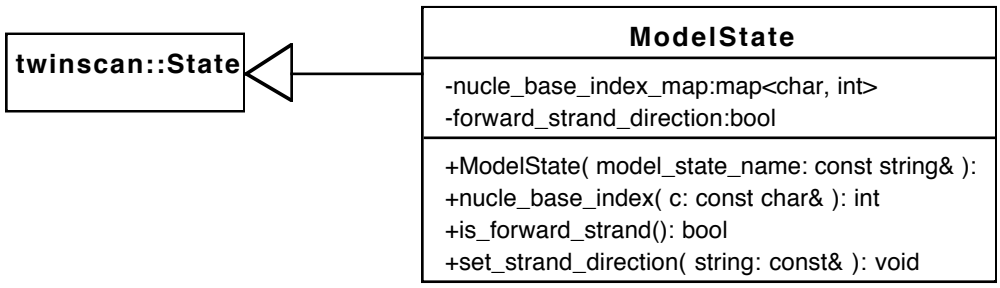

Figure 26. The `ModelState` Class

`ModelState`, like `ModelCell`, inherits most of its functionality from the base class. The added characteristic in `ModelState` is the notation of strand direction and a mapping from the four bases to numerical values. This mapping allows for subclasses to efficiently look up information from parameter scoring matrices.

Attributes:

`nucle_base_index_map`	a mapping between a,c,g,t and 0,1,2,3.
`forward_strand_direction`	true if state is on the forward strand.

Operations:

`nucle_base_index()`	returns numerical mapping from character base.
`is_forward_strand()`	returns `forward_strand_direction`.
`set_strand_direction()`	strand direction is determined from the state name.

Constructor:

Takes a state name, instantiates a `State` object, calls `set_strand_direction()` with the state name and defines the `nucle_base_index_map`.

4.6.1.1 CState

Files: Cstate.h, Cstate.cpp

Inherits from: State, ModelState

Parent of: InitialExon, InternalExon, PolyA, Promoter, TerminalExon

The `Cstate` objects contain the majority of the models associated with scoring exons. The data for these models is read from the genome parameter file into the appropriate locations

Attributes:

`length_distribution_prob`	matrix read in from the genome parameter file.
`length_distribution_score`	calculated score associated with the length distribution.
`begin_site_matrix`	the parameter matrix associated with the signal model at the start of the c-state. Depending on the state, this data structure may contain the parameter matrix associated with the acceptor site, the translation initiation site, or the TATA box model.
`end_site_matrix`	the parameter matrix associated with the biological signal at the end of the c-state.
`coding_site_matrix`	the 5th order Markov coding parameters from the genome parameter file.
`signal_pept`	the parameter matrix associated with the signal peptide model.
`possible_prev_state`	a vector of the allowed previous d-states.
`possible_post_state`	a vector of the allowed following d-states.
`exon_name_without_strand`	the state name without the last character (“+” or “-”).
`mm5_index_map`	a mapping from a sequence of 5 bases to the line in the parameter file containing the corresponding information. Compare to `UtrCdsModel::mm5_index_map`.

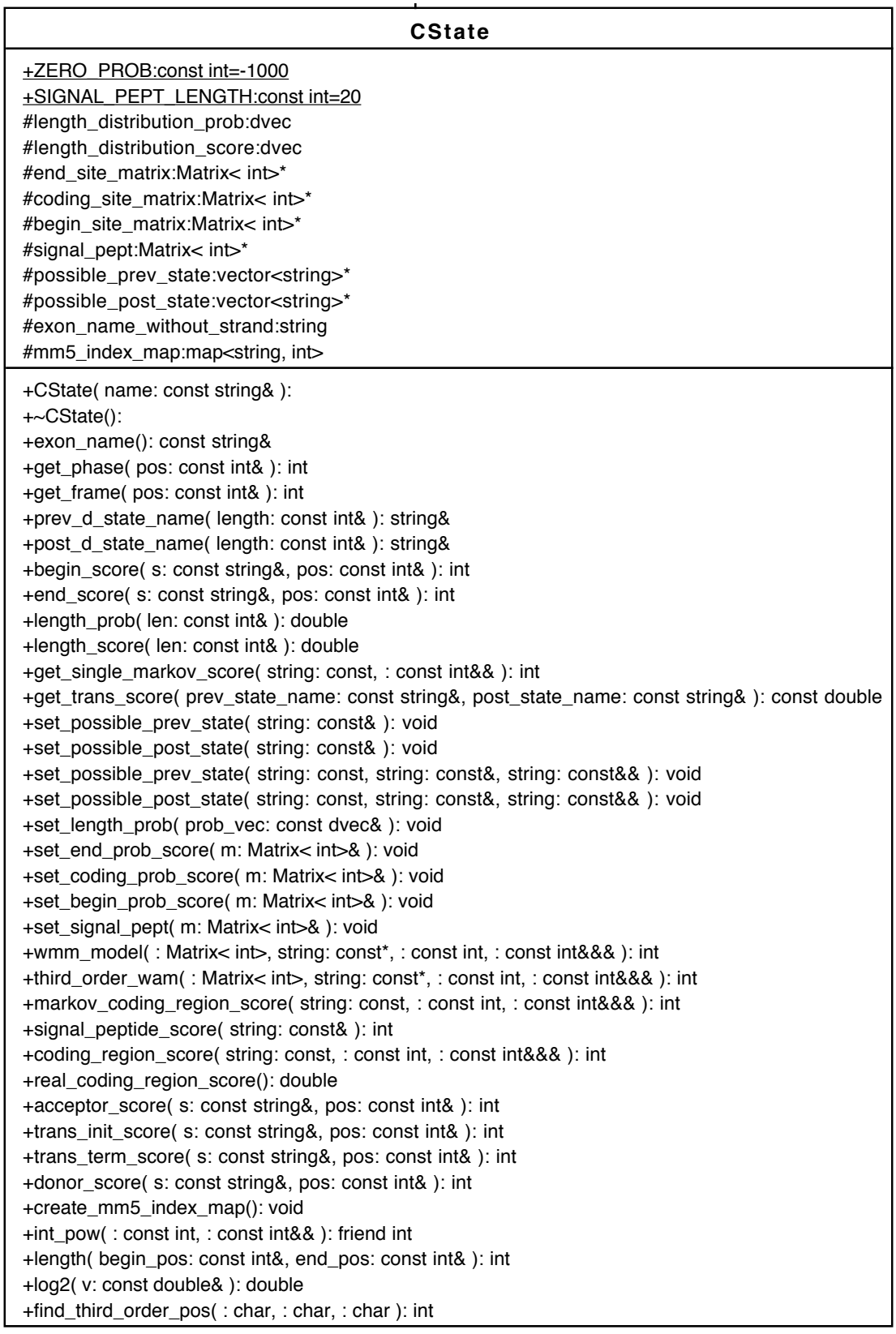

Figure 27. The `CState` Class

Operations:

Parser class objects call a number of the `CState` functions as the information from the parameter files populates the `GenomeModel` class object. These provide the interface for incorporating the model parameters into the `CState` class objects.

`set_possible_prev_state()`	has two interfaces; the first takes a single previous d-state for those c-states that have a unique previous state (such as the forward strand initial exon). The second interface takes three previous d-states as is the case with the forward strand terminal exon.
`set_possible_post_state()`	has two interfaces; the first takes a single following d-state for those c-states that have a unique next state (such as the forward strand terminal exon). The second interface takes three following d-states as is the case with the internal exons.
`set_length_prob()`	stores the length probability data from the parameter file in `length_distribution_prob` and converts the probability values to scores that are stored in `length_distribution_score`. Probability values for exons up to 2000 codons (6000 bp) are contained in the parameter file. This function expands `length_distribution_prob` and `length_distribution_score` to 6000 codons (18000 bp), with minimal probability and score, based on (International Human Genome Sequencing Consortium 2001). `ModelTrellis` functions prevent longer exons from being considered.
`set_end_prob_score()`	allocates memory for and copies parameter file data to `end_site_matrix`.
`set_begin_prob_score()`	allocates memory for and copies parameter file data to `begin_site_matrix`.

`set_coding_prob_score()`	allocates memory for and copies parameter file data to `coding_site_matrix.`
`set_signal_pept()`	allocates memory for and copies parameter file data to `signal_pept.`

There are several virtual functions defined in `CState` and redefined as necessary in each of the `CState` subclasses.

`virtual int get_phase()`	returns the phase of the exon.
`virtual int get_frame()`	returns the reading frame of the exon.
`virtual int begin_score()`	returns the value of the score of the signal assuming that the c-state itself, rather than the respective signal, begins at the parameter position.
`virtual int end_score()`	returns the value of the score of the signal assuming that the c-state itself, rather than the respective signal, ends at the parameter position.
`virtual double length_prob()`	returns the probability value from the parameter file, if one is given in the parameter file.
`virtual double length_score()`	returns the log odds score associated with the length probability.
`virtual const double get_trans_score()`	returns difference between the return values of `State::get_prev_trans_score()` and `State::get_post_trans_score().`
`virtual int content_score()`	returns the portion of the c-state score not associated with the length, biological signals, or transitions.
`virtual double real_coding_region_score()`	returns the score for the entire coding region including the signal peptide score, if appropriate.

Functions that define all of the c-state component models are defined in the `CState` class (see Section 2.3 for descriptions of the models).

`wmm_model()`	iterates through the sequence and the WMM parameter matrix to determine the total score of the signal. For every "n" in the signal, –1 is added to the total score.
`third_order_wam()`	iterates through the sequence and the WAM parameter matrix to determine the total score of the signal. If an "n" is the current position or one of the previous three bases, a score of –1 is added to the total score. Thus a single "n" will result in –4 added to the total score.
`markov_coding_region_score()`	determines the codon position of the first base of the exon, repeatedly calls `get_single_markov_score()` with 6-bp sequences until the last position in the exon and returns the total score.
`get_single_markov_score()`	takes a 6-bp sequence and codon position and returns the appropriate score from `coding_site_matrix`. If the sequence contains an "n," –10 is returned.

Related to the model functions are the functions that define the signal scores. These functions call one or more of the above component model functions.

`acceptor_score()`	returns `ZERO_PROB` if the two nucleotides prior to the current position are not "ag." Otherwise returns `third_order_wam()` with `begin_site_matrix` and the 46-bp sequence starting 43 bases before the current position.

`trans_init_score()`	returns `ZERO_PROB` if the three nucleotides starting at the current position are not "atg." Otherwise returns `wmm_model()` with `begin_site_matrix` and 12-bp sequence starting six bases before the current position.
`trans_term_score()`	returns `ZERO_PROB` if the three bases ending at the current position are not one of the three stop codons. Otherwise returns `wmm_model()` with `end_site_matrix` and 6-bp sequence starting two bases before the current position.
`donor_score()`	returns `ZERO_PROB` if the two bases following the current position are not "gt." Otherwise, returns `wmm_model()` with appropriate portion of `end_site_matrix` (see Section 2.3.4 for more information) and the 9-bp sequence starting two bases before the current position.
`signal_peptide_score()`	iterates through the sequence codon by codon and return the total score. The total score is the sum of appropriate scores from `signal_pept`. If a codon contains an "n," nothing is added to the total score.

The following auxiliary functions are required to support the `CState` core functionality

`create_mm5_index_map()`	create mapping from sequences of five bases to a line in the parameter matrix (e.g. AAAAA – line 1). Compare to `UtrCdsModel::create_mm5_index_map()`
`find_third_order_pos()`	returns the line in the third order WAM matrix that corresponds to a given DNA triplet.

Constructor:

Calls `ModelState` constructor with the state name. Calls `create_mm5_index_map()`. Initializes `exon_name_without_strand` from state name. Initializes following attributes:

`end_site_matrix`

`coding_site_matrix`

`begin_site_matrix`

`signal_pept`

Allocates memory for following attributes:

`possible_prev_state`

`possible_post_state`

4.6.1.1.1 InitialExon

Files: InitialExon.h, InitialExon.cpp

Inherits from: State, ModelState, CState

Used by: TransitionMatrix, GenomeModel

InitialExon
-signal_pept_score:int -overlap_markov_score:int -rear_markov_score:int
+InitialExon(name: const string&): +~InitialExon(): +get_frame(length: const int&): int +begin_score(s: const string&, pos: const int&): int +end_score(s: const string&, pos: const int&): int +content_score(s: const string&, start_pos: const int&, end_pos: const int&): int +real_coding_region_score(): double +length_prob(length: const int&): double +length_score(length: const int&): double +get_trans_score(prev_state_name: const string&, post_state_name: const string&): const double

Figure 28. The `InitialExon` Class

`InitialExon` (like all of the subclasses of `CState`) is fairly small and represents a small amount of objectification. It does, however, make the implementation of `CState` cleaner in that it moves the calculation of functions that are calculated in a different way for at least two of the `CState` subclasses into the subclasses.

`InitialExon` and `SingleExon` states both include the signal peptide model, which is applied over the first 20 codons of the exon. This length is specified by the `SIGNAL_PEPT_LENGTH` constant in `CState`. If the exon is less than 20 codons, the signal peptide model is applied over the entire exon. The signal peptide model does not "carry over" into internal exons. The nucleotides scored with the signal peptide model are also scored by the 5th-order Markov Chain model for coding sequence to determine the `overlap_markov_score`. These nucleotides are represented in the total exon score by a mixture model of 20% of the signal peptide score and 80% of the Markov coding score. The `rear_markov_score` is the 5th-order Markov Chain coding model calculated for that part of the exon sequence not covered by the signal peptide model.

Attributes:

`signal_pept_score`	WMM-based score from the signal peptide model.
`overlap_markov_score`	5th-order Markov coding score from the signal peptide region.
`rear_markov_score`	5th-order Markov coding score from the coding region following the signal peptide region.

Operations:

`get_frame()`	for the initial exon the frame is simply the length mod 3.
`begin_score()`	calls `CState::trans_init_score()`.
`end_score()`	calls `CState::donor_score()`.
`content_score()`	returns the value of the `rear_markov_score` added to the greater of the `signal_pept_score` or the `overlap_markov_score`. This function is used to determine the initial exon coding score that is displayed in the output. It also has the effect of initializing the three attributes by calling `CState::signal_peptide_score()` and `CState::markov_coding_region_score()` as appropriate.
`real_coding_region_score()`	returns the mixture model score for the signal peptide region (see above) added to the `rear_markov_score`. Cannot be called without fist calling `content_score()`.

Constructor:

Calls `CState` constructor with exon name.

4.6.1.1.2 InternalExon

Files: InternalExon.h

Inherits from: State, ModelState, CState

Used by: TransitionMatrix, GenomeModel

InternalExon
-phase_no:int
+InternalExon(name: const string&): +~InternalExon(): +get_phase(length: const int&): int +get_frame(length: const int&): int +begin_score(s: const string&, pos: const int&): int +end_score(s: const string&, pos: const int&): int +content_score(s: const string&, start_pos: const int&, end_pos: const int&): int +length_prob(length: const int&): double +length_score(length: const int&): double

Figure 29. The `InternalExon` Class

`InternalExon` is the simplest of the `CState` subclasses because all of its operations are fully implemented by `CState`.

Attributes:

`phase_no`	phase number designation from the exon name.

Operations:

`get_phase()`	returns `phase_no`.
`get_frame()`	returns exon length mod 3.
`begin_score()`	returns `CState::acceptor_score()`.
`end_score()`	returns `CState::donor_score()`.
`content_score()`	returns `CState::markov_coding_region_score()`.

Constructor:

Calls `CState` constructor with exon name and determines `phase_no` from exon name.

4.6.1.1.3 PolyA

Files: PolyA.h

Inherits from: State, ModelState, CState

Used by: TransitionMatrix, GenomeModel

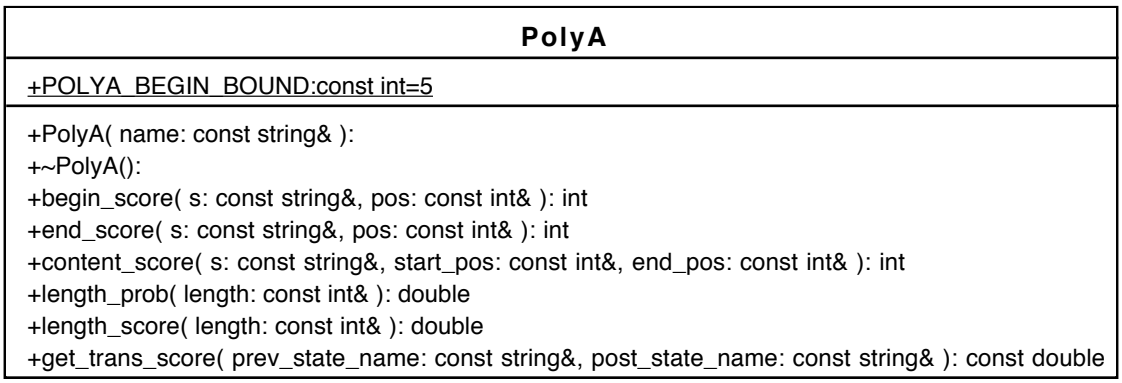

Figure 30. The `PolyA` Class

`PolyA` objects are defined at every position in the sequence except the number of bases at the beginning of the sequence defined by `POLYA_BEGIN_BOUND`. The Poly A model is a 6-bp WMM (see Section 2.3.2 for more details).

Attributes:

`POLYA_BEGIN_BOUND`	defined as 5.

Operations:

`begin_score()`	returns 0.
`end_score()`	returns `CState::wmm_model()` with the Poly A specific parameters.
`content_score()`	returns 0.
`length_prob()`	returns 1 if length is 6, 0 otherwise.
`length_score()`	returns 0 if length is 6, `CState` defined `ZERO_PROB` otherwise.

`get_trans_score()`	returns the sum of `State::get_prev_trans_score()` and `State::get_post_trans_score()` minus a correction factor of 5. The correction factor was derived by a comparison to the Genscan output.

Constructor:

Calls `CState` constructor with state name.

4.6.1.1.4 Promoter

Files: Promoter.h

Inherits from: State, ModelState. CState

Used by: TransitionMatrix, GenomeModel

Promoter
+TATA_END_BOUND:const int=15 +CAPSITE_BEGIN_BOUND:const int=38 +CAPSITE_END_BOUND:const int=6
+Promoter(name: const string&): +~Promoter(): +tata_box_score(s: const string&, pos: const int&): int +begin_score(s: const string&, pos: const int&): int +end_score(s: const string&, pos: const int&): int +content_score(s: const string&, start_pos: const int&, end_pos: const int&): int +real_promoter_score(maximum_tata_box_score: const int&, cap_site_score: const int&): double +length_prob(length: const int&): double +length_score(length: const int&): double

Figure 31. The `Promoter` Class

The promoter model contains two distinct parts separated by between 14 and 20 bases of intergenic region. The upstream part of the model is the 15-bp TATA box model. The downstream portion of the model is the 8-bp CAP site. Between these two portions of the promoter is the 14-20 base pair stretch modeled by the intergenic model.

A significant portion of the Promoter score calculation occurs in the `ModelTrellis` class operation `promoter_detect()` (see Section 4.8.1).

Attributes:

`TATA_END_BOUND`	defined as 15.
`CAPSITE_BEGIN_BOUND`	defined as 38.
`CAPSITE_END_BOUND`	defined as 6.

Operations:

`tata_box_score()`	returns `CState::wmm_model()` with the TATA portion of the parameter file.
`begin_score()`	returns 0.
`end_score()`	returns `CState::wmm_model()` with the cap site portion of the parameter file.
`content_score()`	returns 0.
`real_promoter_score()`	returns the result of the promoter mixture model. In the model, 70% of the promoters contain the TATA-box structure, which may be at one of seven positions.
`length_prob()`	returns 1 if length is 40, 0 otherwise.
`length_score()`	returns 0 if length is 40, `CState` defined `ZERO_PROB` otherwise.

Constructor:

Calls `CState` constructor with state name.

4.6.1.1.5 SingleExon

Files: SingleExon.h, SingleExon.cpp

Inherits from: State, ModelState, CState

Used by: TransitionMatrix, GenomeModel

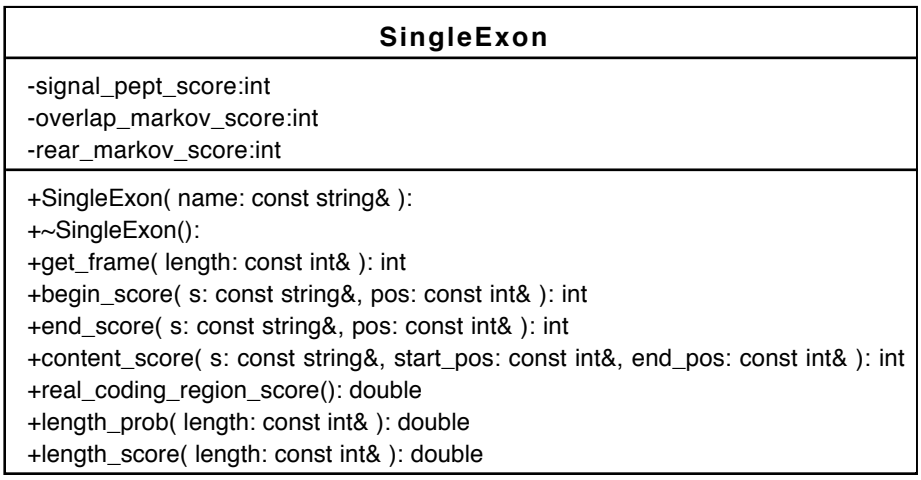

Figure 32. The `SingleExon` Class

Like `InitialExon`, `SingleExon` contains the single peptide model. It is implemented in exactly the same way as `InitialExon` (see Section 4.6.1.1.1) and will not be described again. The only difference in the two classes is the return value for the `end_score()` function. Here `end_score()` returns `CState::trans_term_score()`.

4.6.1.1.6 TerminalExon

Files: TerminalExon.h, TerminalExon.cpp

Inherits from: State, ModelState, CState

Used by: TransitionMatrix, GenomeModel

TerminalExon
+TerminalExon(name: const string&): +~TerminalExon(): +get_phase(length: const int&): int +get_frame(length: const int&): int +begin_score(s: const string&, pos: const int&): int +end_score(s: const string&, pos: const int&): int +content_score(s: const string&, start_pos: const int&, end_pos: const int&): int +length_prob(length: const int&): double +length_score(length: const int&): double

Figure 33. The `TerminalExon` Class

Like the other `CState` sub-classes, `TerminalExon` mostly redefines `CState` virtual functions to appropriate `CState` model functions.

Operations:

`get_phase()`	returns 1 if length mod 3 is 2, 2 if length mod 3 is 1, 0 if length mod 3 is 0.
`get_frame()`	same as `get_phase()`.
`begin_score()`	returns `CState::acceptor_score()`.
`end_score()`	returns `CState::trans_term_score()`.
`content_score()`	returns `CState::markov_coding_region_score()`.

Constructor:

Calls `CState` constructor with exon name.

4.6.1.2 DState

Files: DState.h, DState.cpp

Inherits from: State, ModelState

Used by: TransitionMatrix, GenomeModel

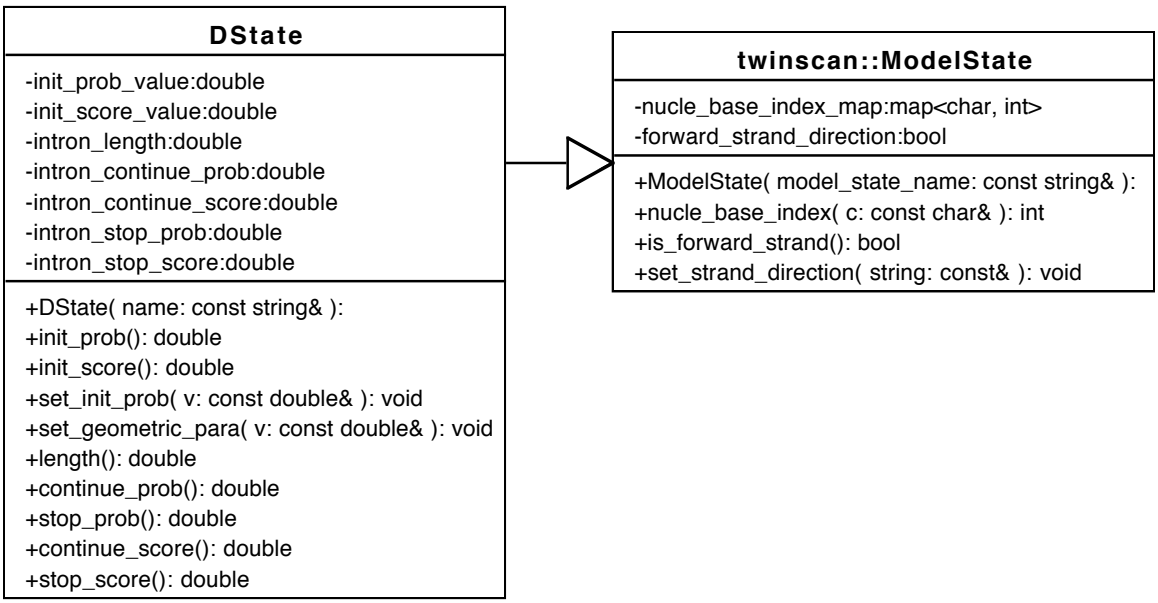

Figure 34. The `DState` Class

The d-states (i.e. non-coding states) in the Twinscan model are much simpler than the c-states. All are modeled with geometric length distributions. The initial probability and geometric length parameter are specified in the genome parameter file, all other values are calculated by `DState` operations called from one of the `Parser` class objects.

Attributes:

`init_prob_value`	value from the genome parameter file.
`init_score_value`	$10 \log_2$ (`init_prob_value`).
`intron_length`	geometric parameter from the genome parameter file.
`intron_continue_prob`	`intron_length` ÷ (`intron_length` – 1).
`intron_continue_score`	$10 \log_2$ (`intron_continue_prob`).
`intron_stop_prob`	1 ÷ `intron_length`.
`intron_stop_score`	$10 \log_2$ (`intron_stop_prob`).

Operations:

The following are called from `Parser` class objects:

`set_init_prob()`	sets attributes associated with `init_prob_value`.
`set_geometric_para()`	sets attributes associated with `intron_length`.

Additional operations exist that return each of the attributes.

`init_prob()`

`init_score()`

`length()`

`continue_prob()`

`continue_score()`

`stop_prob()`

`stop_score()`

Constructor:

Calls `ModelState` constructor with state name and initializes `init_prob_value`, `intron_continue_score`, and `intron_stop_score` to zero.

4.7 TransitionMatrix

Files: TransitionMatrix.h, TransitionMatrix.cpp

Used by: GenomeModel

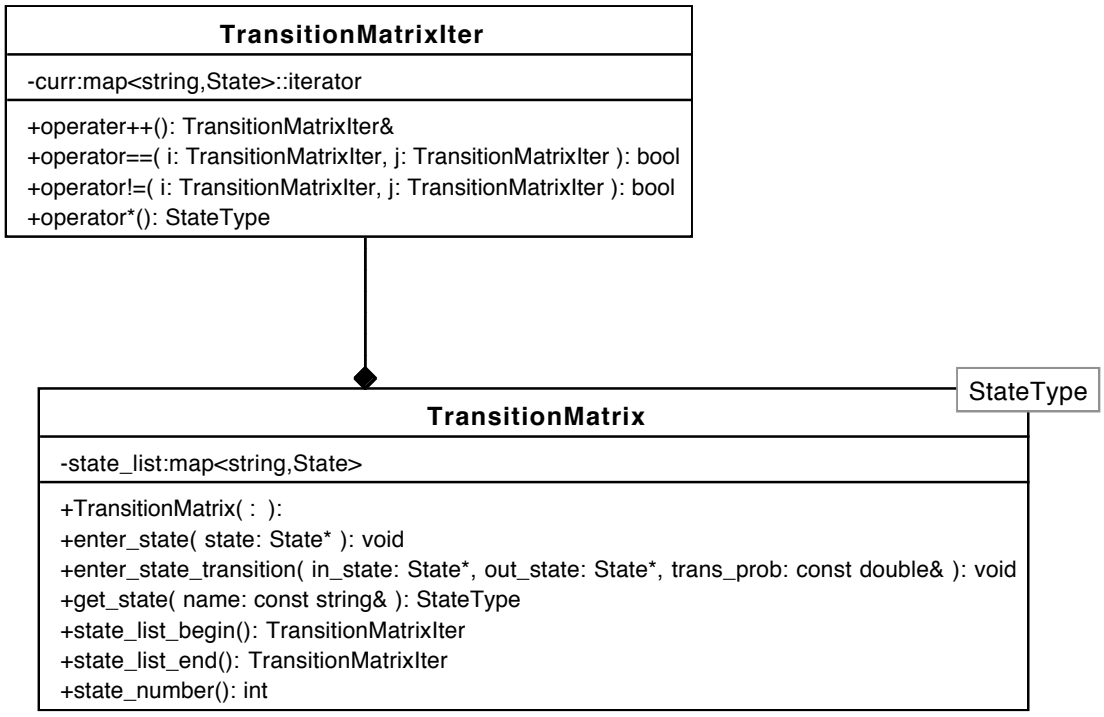

Figure 35. The `TransitionMatrix` Class

The `TransitionMatrix` class is responsible for managing the states, but it is not responsible for their creation or destruction. After the states have been instantiated, information regarding the state name and its allowed transitions is added to the lone attribute `state_list`.

`TransitionMatrix` is parameterized based on whether the states are `CState` objects or `DState` objects. Two instances of the `TransitionMatrix` class are the only attributes of the `GenomeModel` class. Within the class definition a specialized iterator class is also defined.

Attribute:

`state_list`	a map from state names to `State` class objects.

Operations:

`enter_state_transition()`	calls `State::push_back_post_state()` and `State::push_back_prev_state()` to store transition information within the `State` class objects.
`enter_state()`	adds a state to `state_list`.
`get_state()`	returns `State` class object given a state name.
`state_number()`	returns the number of states.

The following operations as well as the `TransitionMatrixIter` class give iterator functionality to `state_list`:

`state_list_begin()`

`state_list_end()`

4.8 Trellis

Files: Trellis.h

Used by: ModelTrellis, NonConsViterbi, UtrCdsViterbi, SpsConsViterbi

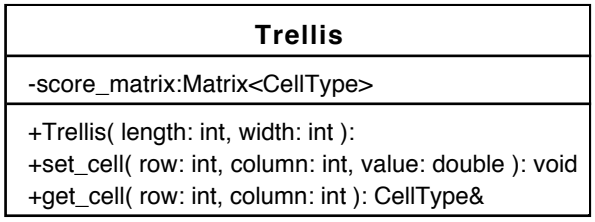

Figure 36. The `Trellis` Class

`Trellis` is the base class for the Viterbi objects that are actually instantiated. The attribute is the actual trellis matrix constructed of `Cell` class objects in a `Matrix` class data structure. Operations are available to set the `Cell` object's value and to return an individual `Cell` object. The `Trellis` constructor sets the `state_map` and `residue_map` attributes of each `Cell` object.

4.8.1 ModelTrellis

Files: ModelTrellis.h, ModelTrellis.cpp
Inherits from: Trellis
Parent of: NonConsViterbi, UtrCdsViterbi, SpsConsViterbi

`ModelTrellis` is one of the largest and most functional classes. A large portion of the methods used by `NonConsViterbi`, `UtrCdsViterbi`, and `SpsConsViterbi` are implemented here. The major goal of `ModelTrellis` is to find, score, and store all of the possible exons. As such, the most important `ModelTrellis` data structure is `forward_exon_info`, a length-of-the-sequence vector containing a list of `ExonInfo` objects at each position in the sequence. All possible exons are stored in `forward_exon_info`.

The operations and attributes in `ModelTrellis` support a "two-pass" methodology for scoring all possible exons. The forward and reverse strands are scored separately using `sequence` and `complement_sequence`, respectively (the reverse strand is scored first). This allows much of the same computational machinery to be used for both strands, although the controlling functions `detect_reverse_exon_begin()` `detect_forward_exon_end()` are strand specific.

Additional data structures store the scores associated with the biological signals and the coding region.

Figure 37. The `ModelTrellis` Class

ModelTrellis attributes include pointers to all of the c-states in the model:

DState*	intergenic_ptr;
InternalExon*	forward_intr_exon_0;
InternalExon*	forward_intr_exon_1;
InternalExon*	forward_intr_exon_2;
InternalExon*	reverse_intr_exon_0;
InternalExon*	reverse_intr_exon_1;
InternalExon*	reverse_intr_exon_2;
InitialExon*	forward_init_exon;
InitialExon*	reverse_init_exon;
TerminalExon*	forward_term_exon;
TerminalExon*	reverse_term_exon;
SingleExon*	forward_sngl_exon;
SingleExon*	reverse_sngl_exon;
Promoter*	forward_promoter;
Promoter*	reverse_promoter;
PolyA*	forward_polya;
PolyA*	reverse_polya;

Other Attributes:

sequence	stl::string read as input.
complement_sequence	stl::string created by ModelTrellis constructor.
index_to_d_state	a vector containing all of the DState objects.
d_state_to_index	a map from each DState class object to an integer.
exon_begin_pos	a map from each CState class object to a list of sequence positions that can create valid possible exons. The size of this list grows at each beginning signal (e.g. acceptor splice site signal) and shrinks at every stop codon.
forward_exon_info	a size-of-the-sequence vector. Each vector position

	contains an `stl::list` of `ExonInfo` objects.
`begin_score_saved`	a map from each `CState` class object to a size-of-the-sequence vector that stores the value returned by `begin_score()` for each of the `CState` subclasses.
`end_score_saved`	a map from each `CState` class object to a size-of-the-sequence vector that stores the value returned by `end_score()` for each of the `CState` subclasses.
`term_exon_begin`	a size-of-the-sequence by three `Matrix` object used to prevent identical terminal exons from being stored in `forward_exon_info`. The matrix structure is required because of the three reading frames.
`sngl_exon_begin`	a size-of-the-sequence vector to prevent identical single exons from being stored in `forward_exon_info`.
`tata_box_score_saved`	a size-of-the-sequence vector for storing TATA scores.
`coding_score_saved`	a size-of-the-sequence by three `Matrix` object used to store the running 5th-order Markov coding score for each of the three reading frames.
`THRESHOLD`	defined as –100. Biological signals with values less than this value will not be considered.

Operations:

`set_d_state_index()`	creates `index_to_d_state` from the `GenomeModel` class object.
`d_state_index()`	returns `d_state_to_index`.
`get_begin_pos()`	returns `exon_begin_pos`.
`size_of_begin_list()`	returns the number of values in `exon_begin_pos`.
`begin_iter()`	returns the first `list::iterator` of `exon_begin_pos`.
`end_iter()`	returns the last `list::iterator` of `exon_begin_pos`.
`adjust_phase()`	removes values from `exon_begin_pos` to prevent exons with in-frame stop codons from being included in `forward_exon_info`. From `begin_iter()` to

	`end_iter()`, if the observed stop codon would be in-frame, the possible begin position is removed.
`pre_processing()`	the major operation of `ModelTrellis`. First for `complement_sequence` and then for `sequence`, calls `promoter_detect()`, `pre_process_coding_score()` with each of the three phases, `detect_*_exon_begin()` and `detect_*_exon_end()`. Also initializes every element of `term_exon_begin` and `sngl_exon_begin` to false.
`detect_forward_exon_begin()` `detect_reverse_exon_begin()`	find the signals that begin possible exons (i.e. translation initiation at the start of initial and single exons and the acceptor site at the start of the internal and terminal exons) and store their scores and positions. These controlling functions call other methods (as described below) with the c-state pointers and `sequence` or `complement_sequence` as appropriate to the forward or reverse strand. - Poly A State: Calls `save_begin_score()` with 0 - Promoter State: Determines `maximum_tata_score` by comparing the value stored by `tata_box_score_saved` at the sequence index that parameterizes these functions and the next 5 index positions. If `maximum_tata_score` is greater than `THRESHOLD`, `save_begin_score()` is called and the sequence index position is stored in `exon_begin_pos` for `*_promoter`. - Internal and Terminal Exon States: If `InternalExon::begin_score()` is greater than `THRESHOLD`, `save_begin_score()` is called and the sequence index position is stored in `exon_begin_pos`

for *_intr_exon_0, *_intr_exon_1, *_intr_exon_2, *_term_exon.
- Initial and Single Exons States: If InitialExon::begin_score() is greater that THRESHOLD, save_begin_score() is called and the sequence index position is stored in exon_begin_pos for *_init_exon and *_sngl_exon.

detect_forward_exon_end()
detect_reverse_exon_end()

find the signals that end possible c-states (i.e. translation termination at the start of terminal and single exons and the donor site at the end of the initial and internal exons) and call functions to store their scores and c-state information. These controlling functions call other methods (as described below) with the c-state pointers and sequence or complement_sequence as appropriate to the forward or reverse strand.
- Poly A State: If PolyA::end_score() is greater than THRESHOLD, both save_end_score() and create_exon_info() are called for *_polya.
- Promoter State: If Promoter::end_score() is greater than THRESHOLD, both save_end_score() and create_exon_info() are called for *_promoter.
- Internal and Initial Exon States: If InternalExon::end_score() is greater than THRESHOLD, both save_end_score() and create_exon_info() are called for *_intr_exon_0, *_intr_exon_1, *_intr_exon_2, and *_init_exon.
- Terminal and Single Exon States: If TerminalExon::end_score() is greater than THRESHOLD, both save_end_score() and

	create_exon_info() are called for *_term_exon and *_sngl_exon.
create_exon_info()	a complex function that organizes the information about each c-state and calls push_back_exon_info().
get_forward_exon_info()	returns forward_exon_info.
push_back_exon_info()	a complex function that determines exon_score, creates a new ExonInfo object if exon_score > EXON_THRESHOLD, and adds the ExonInfo object to forward_exon_info.
remove_exon()	removes a possible exon from forward_exon_info.
restore_begin_score()	returns begin_score_saved for the given CState object and sequence index.
restore_end_score()	returns end_score_saved for the given CState object and sequence index.
save_begin_score()	stores value for given CState object and sequence index in begin_score_saved.
save_end_score()	stores value for given CState object and sequence index in end_score_saved.
promoter_detect()	calls CState::tata_box_score() at every position in the sequence except the 40 bases at each end of the sequence. Saves the return result in tata_box_score_saved.
pre_process_coding_score()	calls CState::get_single_markov_score() with the first five bases in the sequence and the phase parameter. Continues along sequence, one base at a time, calling CState::get_single_markov_score() with consecutive, overlapping 6-mers and with successive phase values. The return value added to the value at the previous position and saved for each sequence index in coding_score_saved. Thus the coding score between and two sequence positions is the difference between

	the their coding_score_saved values. Compare to UtrCdsViterbi::pre_process_forward_conserve_score().
restore_coding_score()	returns the coding score of an exon when given the start and end points of the exon. The score is calculated from six bases beyond the start point to three bases before the end point. Exons less than 9 base pairs are assigned score 0.

Constructor:

Calls Trellis constructor with number of d-states and the size of the sequence. Creates pointers to the target sequence and to the GenomeModel class object. Calls set_d_state_index(). Creates complement_sequence. Allocates memory for the following attributes

forward_exon_info
term_exon_begin
sngl_exon_begin
begin_score_saved
end_score_saved
coding_score_saved
tata_box_score_saved
term_exon_begin
begin_score_saved
end_score_saved

Casts all of the object pointers for the CState subclasses and calls pre_processing().

4.8.1.1 NonConsViterbi

Files: NonConsViterbi.h, NonConsViterbi.cpp

Inherits from: Trellis, ModelTrellis

Parent of: UtrCdsViterbi, SpsConsViterbi

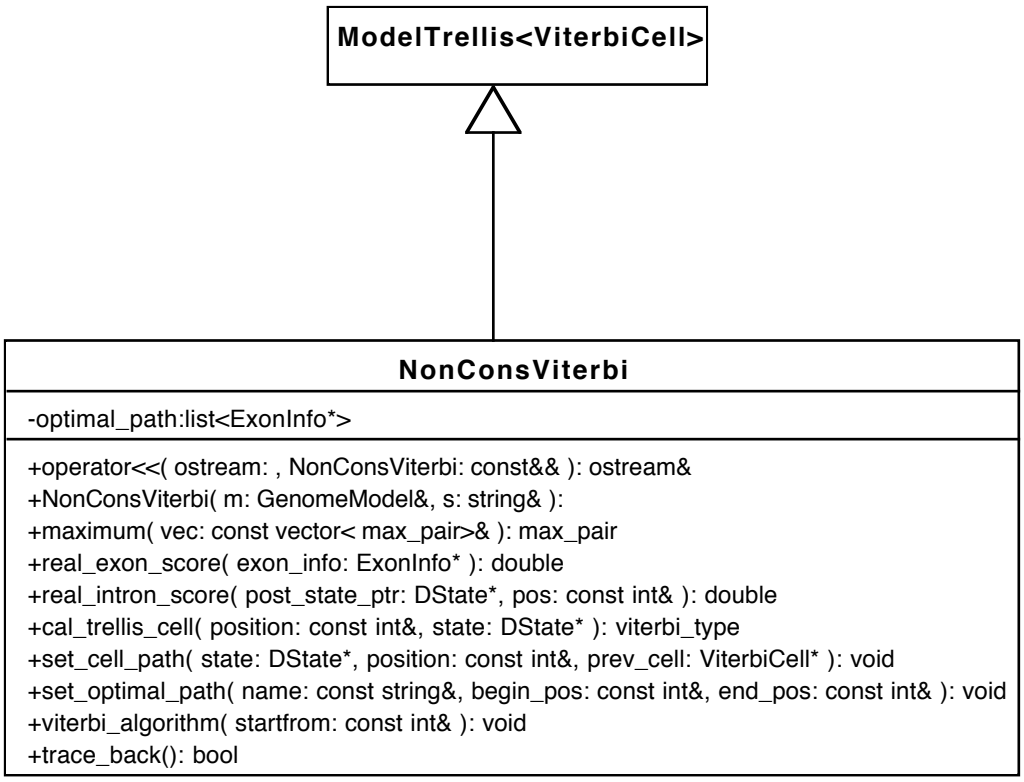

Figure 38. The `NonConsViterbi` Class

`NonConsViterbi` contains the code for the Viterbi algorithm as well as the trace back. As such, much of the computation time is spent here. The `trace_back()` function contains almost all of the code to deal with unexpected in-frame stop codons. The only attribute, `optimal_path`, stores the gene prediction. The gene prediction is output by the overloaded << operator.

Attribute:

`optimal_path`	an `stl:list` of `ExonInfo` objects.

Operations:

`viterbi_algorithm()`	dynamic programming algorithm. The initialization step calls `Trellis::set_cell()` with the sum of the return values from `DState::init_score()` and `DState::continue_score()`. The induction step calls `call_trellis_cell()`, `Trellis::set_cell()`, and `set_cell_path()` for every cell in the trellis matrix. The function may be called multiple times if in-frame stop codons are found by `trace_back()`. In these cases, `viterbi_algorithm()` will be called from some intermediate point in the sequence and the initialization step is not performed.
`trace_back()`	actually determines and stores the optimal parse (see Sections 2.8 and 3.8) of the sequence after the calculations of `viterbi_algorithm()`. This operation has the additional task of ensuring that in-frame stop codons are not introduced across splice site boundaries. The tasks of optimal parse storage and in-frame stop codon avoidance will be described separately. For the optimal parse the `ViterbiCell` object associated with the final d-state of the highest scoring path is determined with `maximum()` at the final sequence index. Starting from the final cell, the most-likely path will either continue in the same d-state to the `ViterbiCell` object at the previous sequence index or transition through an allowed c-state to a `ViterbiCell` object representing the same or another d-state (see Chapter 2). A c-state transition occurs if the value for `Cell::get_residue_map()` of the current `ViterbiCell::get_path()` is not the previous sequence index value. Each c-state transition found results in a call to `set_optimal_path()`. To prevent gene-structure predictions with in-frame stop codons, after the discovery of each c-state transition, the top `ExonInfo` object of `optimal_path` is

	queried. Based on the c-state name, the appropriate number of bases from the exons on each side of the intron is combined to form the in-frame codon across the splice site. If this codon is one of the three stop codons, `ModelTrellis::remove_exon()` is called with the `ExonInfo` object with the lower return value of `ExonInfo::get_exon_score()`. After the `ExonInfo` object is removed from `ModelTrellis::forward_exon_info`, `viterbi_algorithm()` is called from the point of the removal.
`set_optimal_path()`	stores the final gene prediction in `optimal_path`. The lower sequence index is always assigned to `ExonInfo::begin_pos`.
`set_cell_path()`	part of `viterbi_algorithm()` implementation. Calls `ViterbiCell::set_path()` with current cell.
`call_trellis_cell()`	returns the value and index of the most probable exon or the value and index of the previous cell. This is accomplished at each sequence index by iterating through the list returned by `ModelTrellis::get_forward_exon_info()` to find those c-states (if any) compatible with each d-state. For each d-state, the maximum score of the compatible c-states is compared with the return value of `real_intron_score()`. The information associated with the larger of these scores is returned.
`real_intron_score()`	returns the value of `Cell::get_cell_value()` from the previous cell in the same row of the trellis added to the value returned by `DState::continue_score()` for the d-state associated with the current trellis row.
`maximum()`	returns the maximum value and index of a vector.
`operator<<`	prints the gene prediction (i.e. optimal parse) to standard out in a format that is "parsably-equivalent" to Genscan's output. This operator is defined as a friend to `NonConsViterbi`.

Constructor:

Calls the `ModelTrellis` class constructor with `ViterbiCell` template and a reference to both the `GenomeModel` class object and the target DNA sequence.

4.8.1.1.1 UtrCdsViterbi

Files: UtrCdsViterbi.h, UtrCdsViterbi.cpp

Inherits from: Trellis, ModelTrellis, NonConsViterbi

Used by: SpsConsViterbi

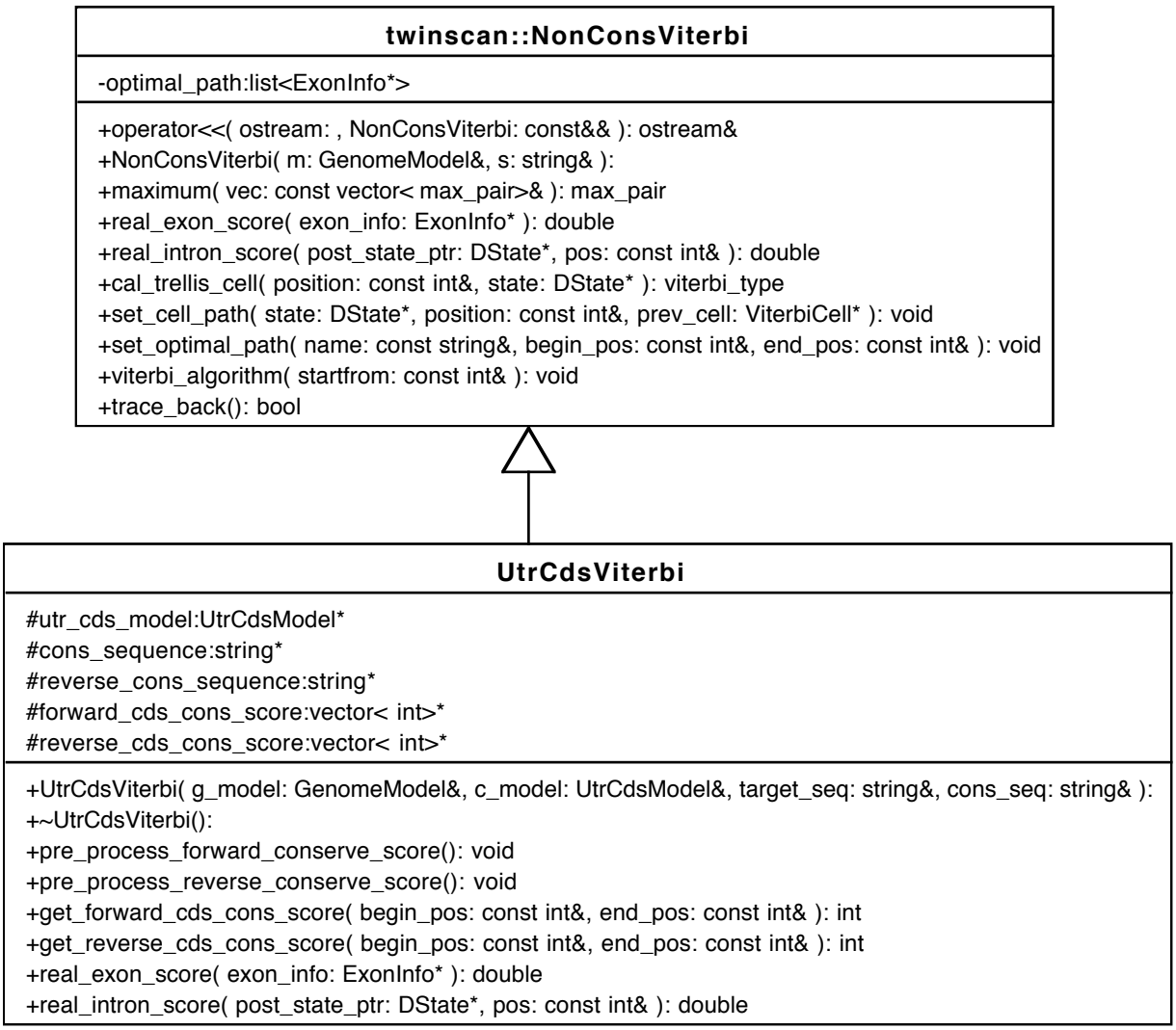

Figure 39. The `UtrCdsViterbi` Class

The `UtrCdsViterbi` class contains the attributes that store the conservation sequence and the reverse conservation sequence. In contrast to the DNA sequence where the reverse strand contains bases complementary to the forward strand (i.e. A on the forward strand is T on the reverse strand), conservation sequence is identical on both strands. Thus, for

the reverse strand the DNA sequence is both reversed and complemented, while the conservation sequence is simply reversed.

Attributes:

`cons_sequence`	the conservation sequence read in from file.
`reverse_cons_sequence`	reversed `cons_sequence`.
`forward_cds_cons_score`	a length-of-the-sequence vector containing the running sum of the forward strand coding sequence conservation score.
`reverse_cds_cons_score`	a length-of-the-sequence vector containing the running sum of the reverse strand coding sequence conservation score.

Operations:

`pre_process_forward_conserve_score()`

`pre_process_reverse_conserve_score()`

These operations work in a similar manner to `ModelTrellis::pre_processing()`. In this case `UtrCdsModel::cds_cons_score()` is called repeatedly with overlapping 6-mers of conservation symbols starting with the 6th position in the sequence. The return value added to the value at the previous position and saved for each sequence index in `forward_cds_cons_score` or `reverse_cds_cons_score` depending on the function. Thus the conservation score between any two sequence positions is the difference between the their `*_cds_cons_score` values.

`get_forward_cds_cons_score()`	returns the difference between two index positions in `forward_cds_cons_score`.

`get_reverse_cds_cons_score()`	returns the difference between two index positions in `reverse_cds_cons_score`.
`real_exon_score()`	returns the total exon score including the information from the genome and coding sequence conservation model. Based on the name of the exon and its strand designation, the appropriate conservation score is added to the value returned by `NonConsViterbi::real_exon_score()`. (See Figure 5 for details about the association between conservation models and exons.) Also calls `ExonInfo::set_exon_score()` with the return value. This version of the function is superceded by the version in `SpsConsViterbi` when the full conservation model is used.
`real_intron_score()`	returns the total intron score for the 5' UTR state and the 3' UTR state by adding the return value from `NonConsViterbi::real_intron_score()` to the value returned by `UtrCdsModel::get_utr_score()`.

Constructor:

Calls the `NonConsViterbi` constructor and assigns `utr_cds_model` and `cons_sequence`. Creates `reverse_cons_sequence` and allocates memory for `forward_cds_cons_score` and `reverse_cds_cons_score`. Calls `pre_process_forward_conserve_score()` and `pre_process_reverse_conserve_score()`.

4.8.1.1.1.1 SpsConsViterbi

Files: SpsConsViterbi.h, SpsConsViterbi.cpp

Inherits from: Trellis, ModelTrellis, NonConsViterbi, UtrCdsViterbi

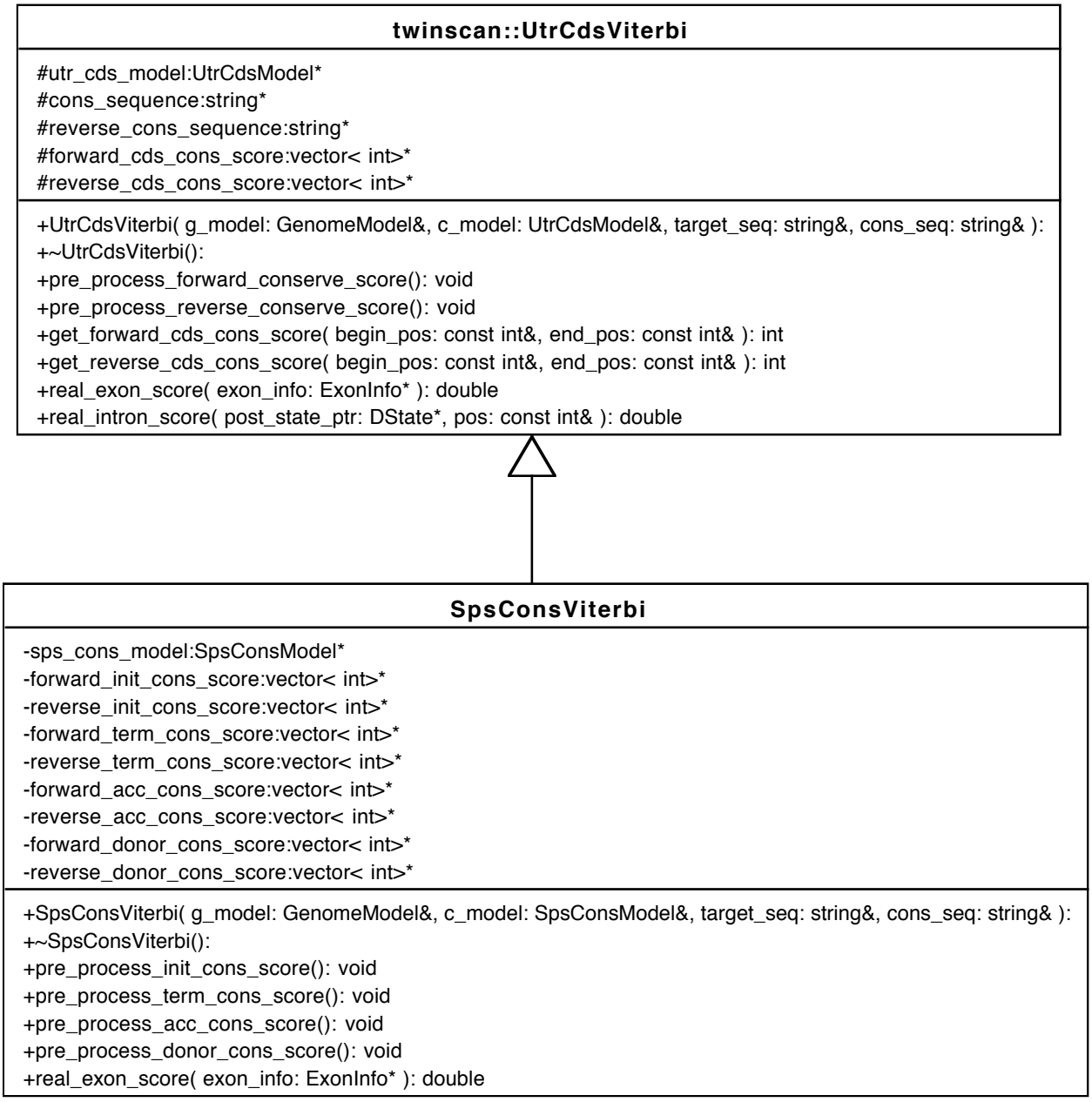

Figure 40. The `SpsConsViterbi` Class

Like the other Viterbi Trellis classes, almost all of the functionality of `SpsConsViterbi` is managed through its constructor. For each of the pre-processing methods, first `cons_sequence` and then `reverse_cons_sequence` is considered. The only redefined `ModelTrellis` method is `real_exon_score()`. All of the attributes are integer vectors that are the length of the sequence.

Operations:

`pre_process_init_cons_score()`	calculates the score for the translation initiation signal in the conservation sequence at the location of every "atg" in the target DNA sequence. The value is stored at the appropriate index in `forward_init_cons_score` or `reverse_init_cons_score`. Because of the length of the signal, the 11 bases at each end are not considered.
`pre_process_term_cons_score()`	calculates the score for the translation termination signal in the conservation sequence at the location of every stop codon in the target DNA sequence. The value is stored at the appropriate index in `forward_term_cons_score` or `reverse_term_cons_score`. Because of the length of the signal, the 5 bases at the start of the sequence and the 6 bases the end of the sequence are not considered.
`pre_process_acc_cons_score()`	calculates the score for the acceptor site in the conservation sequence at the location of every "ag" in the target DNA sequence. The value is stored at the appropriate index in `forward_acc_cons_score` or `reverse_acc_cons_score`. Because of the length of the signal, the 43 bases at the start of

	the sequence and the 6 bases the end of the sequence are not considered.
`pre_process_donor_cons_score()`	calculates the score for the donor site in the conservation sequence at the location of every "gt" in the target DNA sequence. The value is stored at the appropriate index in `forward_donor_cons_score` or `reverse_donor_cons_score`. Because of the length of the signal, the 8 bases at the start of the sequence and the 9 bases the end of the sequence are not considered.
`real_exon_score()`	returns the total exon score including the information from the genome and conservation models. Based on the name of the exon and its strand designation, the appropriate conservation scores are added to the value returned by `NonConsViterbi::real_exon_score()`. (See Figure 5 for details about the association between conservation models and exons.) Also calls `ExonInfo::set_exon_score()` with the return value.

Constructor:

Calls `UtrCdsViterbi` constructor and assigns `sps_cons_model`. Allocates memory for all eight attributes. Calls the following functions:

`pre_process_init_cons_score()`

`pre_process_term_cons_score()`

`pre_process_acc_cons_score()`

`pre_process_donor_cons_score()`

4.9 UtrCdsModel

Files: UtrCdsModel.h, UtrCdsModel.cpp

Used by: SpsConsModel

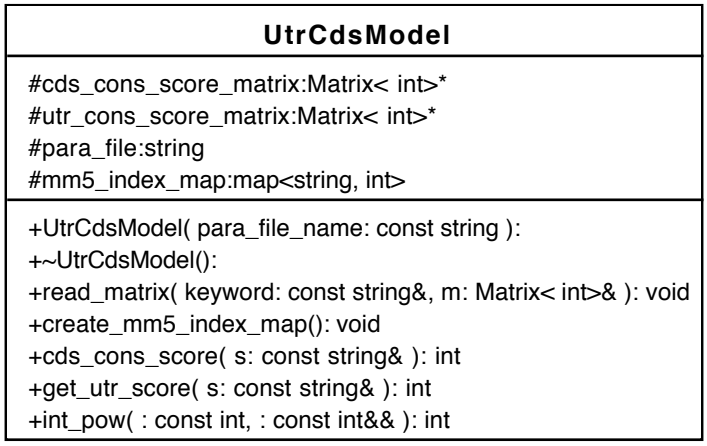

Figure 41. The `UtrCdsModel` Class

`UtrCdsModel` is the base class of the conservation model structure. The conservation model classes are much simpler than the `GenomeModel` class. Both `UtrCdsModel` and its subclass, `SpsConsModel` are designed to work closely with `GenomeModel`.

Attributes:

`para_file`	name of the conservation parameter file.
`mm5_index_map`	a mapping from a sequence of conservation symbols to the line in the parameter file containing the corresponding information. Compare to `CState::mm5_index_map`.
`cds_cons_score_matrix`	
`utr_cons_score_matrix`	

Operations:

`read_matrix()`	reads parameter matrix from the conservation parameter file. The size of the matrix is known in advance.

`create_mm5_index_map()`	create mapping from sequences of conservation symbols to a line in the parameter matrix. Compare to `CState::create_mm5_index_map()`.
`cds_cons_score()`	returns appropriate score from the `cds_cons_score_matrix` when given 6 conservation symbols.
`get_utr_score()`	returns appropriate score from the `utr_cons_score_matrix` when given 6 conservation symbols.

Constructor:

Defines the size of the parameter matrices as the value of `CON_BITS` raised to the fifth power. Assigns `para_file` to the name of the conservation parameter file. Calls `create_mm5_index_map()`.

Allocates memory for the `cds_cons_score_matrix` and `utr_cons_score_matrix`. Reads the data from the conservation parameter file into the attributes using `read_matrix()`.

4.9.1 SpsConsModel

Files: SpsConsModel,h, SpsConsModel.cpp

Inherits from: UtrCdsModel

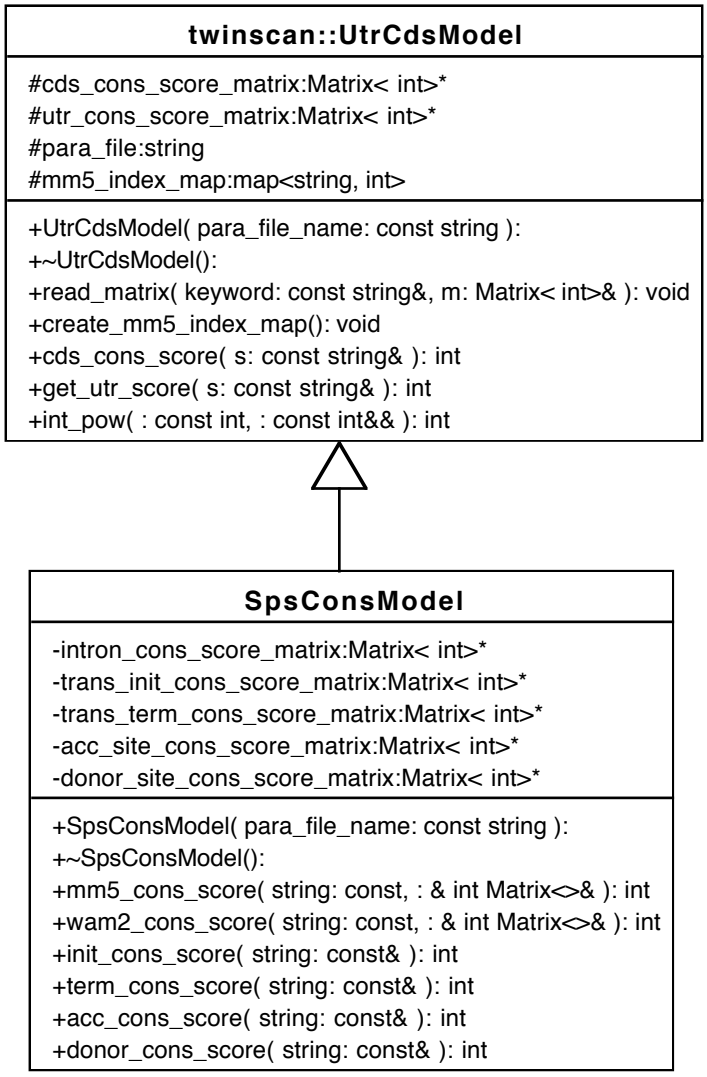

Figure 42. The `SpsConsModel` Class

`SpsConsModel` contains the data structures for implementing the Twinscan conservation model. The sizes of the various score matrices are based on the value of `CON_BITS`. For all of the biological signals modeled in `SpsConsModel`, the 5th-order Markov model of the conservation sequence in intron sequence is used as the null model. Thus, the conservation score returned for the translation initiation and termination signals is the

5th-order Markov score from the signal model subtracted by the 5th-order Markov score from the intron model. For the case of the splice site models, the score returned is the score from signal-specific second-order WAM model subtracted by the 5th-order Markov score from the intron model.

Attributes:

`intron_cons_score_matrix`
`trans_init_cons_score_matrix`
`trans_term_cons_score_matrix`
`acc_site_cons_score_matrix`
`donor_site_cons_score_matrix`

Operations:

`mm5_cons_score()`	returns the score given a sequence and a 5th-order Markov model parameter matrix.
`wam2_cons_score()`	returns the score given a sequence and a second order Weight Array Model parameter matrix.
`init_cons_score()`	returns the `mm5_cons_score()` from the `trans_init_cons_score_matrix` subtracted by the `mm5_cons_score()` from the `intron_cons_score_matrix`.
`term_cons_score()`	returns the `mm5_cons_score()` from the `trans_term_cons_score_matrix` subtracted by the `mm5_cons_score()` from the `intron_cons_score_matrix`.
`acc_cons_score()`	returns the `wam2_cons_score()` from the `acc_site_cons_score_matrix` subtracted by the `mm5_cons_score()` from the `intron_cons_score_matrix`.
`donor_cons_score()`	returns the `wam2_cons_score()` from the `donor_site_cons_score_matrix` subtracted by the `mm5_cons_score()` from the `intron_cons_score_matrix`.

Constructor:

Calls `UtrCdsModel` constructor with the name of the file containing the conservation parameters. Allocates memory for all of the `SpsConsModel` attributes and reads the data from the conservation parameter file into the attributes using `UtrCdsModer::read_matrix()`.

Chapter 5

Known Bugs and Other Information

1. In rare cases (approximately 1 in 4000 predictions on the human genome) Twinscan will predict genes that do not translate in any frame. In these cases the phase is inconsistent across the exons. This may be a rare trace back problem, but its actual cause and location are not currently known.

2. In the Genscan compatibility mode, Twinscan does not predict promoters at the same locations as Genscan. In addition, some exons do not get the same score as they do in Genscan. This is believed to be associated with long introns. We have shown that these minor differences do not significantly alter performance (Korf et al. 2001).

Glossary of Terms

aggregation – a specific type of association in which one class is a component of another.

association – a relationship between two classes.

attribute – a property of a class. An attribute describes a range of values the property may hold in objects.

class – a category or group of things that have similar attributes and common behavior; it's a template for creating objects.

codon – sequences of three consecutive nucleotides in DNA or mRNA that specifies a particular amino acid during protein synthesis.[+]

conservation bits – number of distinct symbols in the conservation sequence.

exon – segments of a eukaryotic gene that reach the cytoplasm as a part of a mature mRNA[+]

fasta format – a standard format for biological sequence information consisting of an initial single line description beginning with the greater-than symbol (">") followed by line of sequence data.

genome – total genetic information carried by an organism.[+]

GFF – General Feature Format: a protocol to transfer biological sequence annotation information (Durbin and Haussler).

GTF – Gene Transfer Format. A restrictive subset of GFF.

intron – part of the DNA encoding a gene that is removed by splicing during RNA processing and is not included in the mature, functional mRNA.[+]

mRNA (messenger RNA) – any RNA that specifies the order of amino acids in a protein. It is produced by transcription of DNA.[+]

object – an instance of a class that has values for each of the class's attributes.

 Definition taken or adapted from Schmuller Schmuller, J. 2002. *Sams Teach Yourself UML in 24 Hours*. Sams Publishing, Indianapolis, Indiana.

[+] Definition taken or adapted from Lodish et al. Lodish, H.F., A. Berk, S.L. Zipursky, p. Matsudaira, D. Baltimore, and J. Darnell. 2000. *Molecular cell biology*. W.H. Freeman, New York.

proteome – the complete catalog of all of an organism's proteins.

read – the sequence of a small piece of DNA.

synteny – the conservation of gene order between two species.

transcription – process whereby one strand of a DNA molecule is used as a template for synthesis of an mRNA.[+]

translation – the production of a polypeptide chain whose amino acid sequence is specified by an mRNA.[+]

Acknowledgments

I'd like to thank the following for their help and support during the course of this project. Melissa Norton provided unending support including comments on early drafts of this document. Michael Brent and Ian Korf developed many of the critical ideas that made Twinscan a reality. Daniel Duan wrote most of the code bases' initial classes. Matt Snover created the first usable version of `Twinscan.cpp` to tie together all of the probability models. Chris Burge answered just about every question we asked (which wasn't too many) and his answers were instrumental in getting the scoring for the initial exon correct. All of the UML diagrams were created with Object Plant 3.1.x (Arctaedius 2002). Twinscan is currently developed using Project Builder on Mac OS X.

References

GTF2: Mouse/Human Annotation Collaboration: Submission Format.

Adams, M.D. S.E. Celniker R.A. Holt C.A. Evans J.D. Gocayne P.G. Amanatides S.E. Scherer P.W. Li R.A. Hoskins R.F. Galle R.A. George S.E. Lewis S. Richards M. Ashburner S.N. Henderson G.G. Sutton J.R. Wortman M.D. Yandell Q. Zhang L.X. Chen R.C. Brandon Y.H. Rogers R.G. Blazej M. Champe B.D. Pfeiffer K.H. Wan C. Doyle E.G. Baxter G. Helt C.R. Nelson G.L. Gabor J.F. Abril A. Agbayani H.J. An C. Andrews-Pfannkoch D. Baldwin R.M. Ballew A. Basu J. Baxendale L. Bayraktaroglu E.M. Beasley K.Y. Beeson P.V. Benos B.P. Berman D. Bhandari S. Bolshakov D. Borkova M.R. Botchan J. Bouck P. Brokstein P. Brottier K.C. Burtis D.A. Busam H. Butler E. Cadieu A. Center I. Chandra J.M. Cherry S. Cawley C. Dahlke L.B. Davenport P. Davies B. de Pablos A. Delcher Z. Deng A.D. Mays I. Dew S.M. Dietz K. Dodson L.E. Doup M. Downes S. Dugan-Rocha B.C. Dunkov P. Dunn K.J. Durbin C.C. Evangelista C. Ferraz S. Ferriera W. Fleischmann C. Fosler A.E. Gabrielian N.S. Garg W.M. Gelbart K. Glasser A. Glodek F. Gong J.H. Gorrell Z. Gu P. Guan M. Harris N.L. Harris D. Harvey T.J. Heiman J.R. Hernandez J. Houck D. Hostin K.A. Houston T.J. Howland M.H. Wei C. Ibegwam M. Jalali F. Kalush G.H. Karpen Z. Ke J.A. Kennison K.A. Ketchum B.E. Kimmel C.D. Kodira C. Kraft S. Kravitz D. Kulp Z. Lai P. Lasko Y. Lei A.A. Levitsky J. Li Z. Li Y. Liang X. Lin X. Liu B. Mattei T.C. McIntosh M.P. McLeod D. McPherson G. Merkulov N.V. Milshina C. Mobarry J. Morris A. Moshrefi S.M. Mount M. Moy B. Murphy L. Murphy D.M. Muzny D.L. Nelson D.R. Nelson K.A. Nelson K. Nixon D.R. Nusskern J.M. Pacleb M. Palazzolo G.S. Pittman S. Pan J. Pollard V. Puri M.G. Reese K. Reinert K. Remington R.D. Saunders F. Scheeler H. Shen B.C. Shue I. Siden-Kiamos M. Simpson M.P. Skupski T. Smith E. Spier A.C. Spradling M. Stapleton R. Strong E. Sun R. Svirskas C. Tector R. Turner E. Venter A.H. Wang X. Wang Z.Y. Wang D.A. Wassarman G.M. Weinstock J. Weissenbach S.M. Williams WoodageT K.C. Worley D. Wu S. Yang Q.A. Yao J. Ye R.F. Yeh J.S. Zaveri M. Zhan G. Zhang Q. Zhao L. Zheng X.H. Zheng F.N. Zhong W. Zhong X. Zhou S. Zhu X. Zhu H.O. Smith R.A. Gibbs E.W. Myers G.M. Rubin and J.C. Venter. 2000. The genome sequence of Drosophila melanogaster. *Science* **287:** 2185-2195.

Arctaedius, M. 2002. Object Plant, Stockholm.

Bernal, A., U. Ear, and N. Kyrpides. 2001. Genomes OnLine Database (GOLD): a monitor of genome projects world-wide. *Nucleic Acids Research* **29:** 126-127.

Bernardi, G. 1989. The isochore organization of the human genome. *Annual Review of Genetics* **23**.

Bernardi, G. 2000. Isochores and the evolutionary genomics of vertebrates. *Gene* **241:** 3-20.

Bernardi, G., B. Olofsson, J. Filipski, M. Zerial, J. Salinas, G. Cuny, M. Meunier-Rotival, and F. Rodier. 1985. The mosiac genome of warm-blooded vertebrates. *Science* **228:** 953-958.

Bucher, P. 1990. Weight matrix descriptions of four eukaryotic RNA polymerase II promoter elements derived from 502 unrelated promoter sequences. *J Mol Biol* **212:** 563-578.

Burge, C. New GENSCAN Web Server at MIT.

Burge, C. 1997. Identification of genes in human genomic DNA. Stanford University.

Burge, C. and S. Karlin. 1997. Prediction of Complete Gene Structures in Human Genomic DNA. *Journal of Molecular Biology* **268:** 78-94.

Crollius, H.R., O. Jaillon, C. Dasilva, C. Ozouf-Costaz, C. Fizames, C. Fischer, L. Bouneau, A. Billault, F. Quetier, W. Saurin, A. Bernot, and J. Weissenbach. 2000. Characterization and repeat analysis of the compact genome of the freshwater pufferfish Tetraodon nigroviridis. *Genome Res* **10:** 939-949.

Durbin, R., S. Eddy, A. Krogh, and G. Mitchenson. 1998. *Biological Sequence Analysis*. Cambridge University Press.

Durbin, R. and D. Haussler. The Sanger Institute: GFF.

Eddy, S.R. 1998. Profile hidden Markov models. *Bioinformatics* **14:** 755-763.

Fickett, J.W. and C.S. Tung. 1992. Assessment of protein coding measures. *Nucleic Acids Res* **20:** 6441-6450.

Flicek, P., E. Keibler, P. Hu, I. Korf, and M.R. Brent. 2003. Leveraging the Mouse Genome for Gene Prediction in Human: From Whole-Genome Shotgun Reads to a Global Synteny Map. *Genome Research* **13**.

Gish, W. 2002. WU-BLAST Archives.

International Human Genome Sequencing Consortium. 2001. Initial sequencing and analysis of the human genome. *Nature* **409:** 860-921.

Korf, I., P. Flicek, D. Duan, and M.R. Brent. 2001. Integrating Genomic Homology into Gene-structure Prediction. *Bioinformatics* **17:** S140-S148.

Kyrpides, N.C. 1999. Genomes OnLine Database (GOLD): a monitor of complete and ongoing genome projects world-wide. *Bioinformatics* **15:** 773-774.

Lippman, S.B. and J. Lajoie. 1998. *C++ Primer*. Addison-Wesley, Reading, Massachusetts.

Lodish, H.F., A. Berk, S.L. Zipursky, p. Matsudaira, D. Baltimore, and J. Darnell. 2000. *Molecular cell biology*. W.H. Freeman, New York.

Mitchell, T.M. 1997. *Machine Learning*. McGraw-Hill, New York.

Mouse Genome Sequencing Consortium. 2002. Initial Sequencing and Comparative Analysis of the Mouse Genome. *Nature* **420:** 520-562.

Salzberg, S. 1997. A method for identifying splice sites and translational start sites in eukaryotic mRNA. *Computer Applications in the Biosciences* **13:** 365-376.

Sanger, F. and A.R. Coulson. 1978. The use of thin acrylamide gels for DNA sequencing. *FEBS Lett* **87:** 107-110.

Sanger, F., S. Nicklen, and A.R. Coulson. 1977. DNA sequencing with chain-terminating inhibitors. *Proc Natl Acad Sci U S A* **74:** 5463-5467.

Schmuller, J. 2002. *Sams Teach Yourself UML in 24 Hours*. Sams Publishing, Indianapolis, Indiana.

The *C. elegans* Sequencing Consortium. 1998. Genome sequence of the nematode C. elegans: a platform for investigating biology. *Science* **282:** 2012-2018.

Venter, J.C., M.D. Adams, and others. 2001. The sequence of the Human Genome. *Science* **291:** 1304-1351.

Watson, J.D. and F.H.C. Crick. 1953. Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid. *Nature* **171:** 737-738.

Zhang, M.Q. 2002. Computational prediction of eukaryotic protein-coding genes. *Nat Rev Genet* **3:** 698-709.

Zhang, M.Q. and T.G. Marr. 1993. A weight array method for splicing signal analysis. *Comput Appl Biosci* **9:** 499-509.

Appendix A

Integrating genomic homology into gene-structure prediction.

Ian Korf, Paul Flicek, Daniel Duan, and Michael R. Brent.

Bioinformatics. 17(S1). S140-S148. 2001.

Appendix B

Leveraging the mouse genome for gene prediction in human: from whole-genome shotguns reads to a global synteny map.
Paul Flicek, Evan Keibler, Ping Hu, Ian Korf, and Michael R. Brent.
Genome Research. 13(1). 46-54. 2003.

Class Index