The FAKtory DNA Sequence Fragment Assembly System

Susan Miller
Gene Myers

TR 99-03

The FAKtory DNA Sequence Fragment Assembly System*

Susan Miller
Gene Myers

TR 99-03

ABSTRACT

FAKtory is a software environment for supporting DNA sequencing projects. FAKtory runs on machines configured with the UNIX operating system. It is called FAKtory because it employs our FAKII Fragment Assembly Kernel that is a C-library of routines incorporating our best algorithms for solving the shotgun assembly problem. The system is highly configurable and can perform fragment clipping, prescreening, and tagging functions, shotgun assembly with or without constraints, and sequence finishing. An extensible input/output mechanism permits FAKtory to be inserted into any informatics environment.

February 3, 1999

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

***This work was supported in part by DOE Grant DE-FG03-94ER61911.**

The FAKtory DNA Sequence Fragment Assembly System

The following ten sections describe the FAKtory system. We do so, not only to provide a record of what has been done, but also because we think that the design of FAKtory is very innovative and distinct in many ways from current systems. We hope that readers will find the design interesting and worthy of support. We have built a very flexible and general framework that includes the following design features:

1. FAKtory can be configured with a very large range of options, making it flexible. The resulting complexity is shielded from *end users* by a default scheme which permits a *master user* to establish a lab-customized configuration that all end users see by default.
2. FAKtory has been designed to permit *co-operative work* on a given sequencing project.
3. FAKtory provides a general-purpose *fragment database*: the type and format of the data kept for each sequencing reaction is configurable, one may control the *vocabularies* for given fields and establish *default values* for them, one may sort and view records on any combination of fields, and one may select sets of fragment records using an SQL-like query mechanism that supports *string searches*.
4. FAKtory provides a *fragment pipeline* which is a configurable sequence of *processing stages* through which fragments must pass. There are stages for input, overlap detection, assembly, and prescreening activities such as vector removal, poor-data trimming, and Alu tagging. FAKtory support *modes* of processing from fully automatic to completely manual, and further provides a *review mechanism* for handling processing anomalies.
5. FAKtory uses the ideas of *input* and *output filters* to make it *extensible*, e.g., handling an unlimited range of input formats and linking to an unlimited range of post-analysis programs.
6. The FAKtory *prescreener* stages employ a very general framework that permits users to develop highly sophisticated criteria for clipping or tagging data.
7. FAKtory provides mechanisms that allow for the automatic association of *constraints* for pairs of fragments based on naming conventions. The constraints can be any combination of overlap, orientation, and distance constraints as realized by the FAK II assembly kernel.
8. FAKtory supports the idea of an *assembly scratchpad* where one can manipulate and interact with a set of possible alternate solutions to a given data set.
9. The FAKtory *finisher* automatically keeps track of edited regions of a multi-alignment, organizing the task of editing the penultimate solution. The finisher maintains a complete history of all editing sessions and allows unlimited undo and redo of edits.

This partial list is intended to convey the breadth and innovation in FAKtory's design. We proceed in the following sections to detail the various components of FAKtory. The reader may peruse parts or all of this at their whim.

1. Customizations & Preferences: Making FAKtory Flexible Yet Simple

A basic problem in building a single software tool to support DNA sequencing is that different laboratories use different sequencing protocols, maintain different information about their sequencing reactions, and so on. How do you build *one* software tool that can handle a wide range of situations?

To address this, FAKtory has been designed so that it can be *configured* in a number of ways. For example, one can configure a process pipeline for sequencing reaction data to consist of any number of stages, each of which may be configured to perform a different function, such as vector removal or poor-data trimming. One can also configure a database of the sequencing data where each entry has a particular set of fields of specified types of data. In FAKtory those configuration options which cannot be modified once data begins to enter a project are called *customizations*. Configuring the pipeline and establishing the form of the database are examples of customizations. All other configuration options are termed *preferences*. These range from fairly complex settings such as those that program a vector trimming operation in the pipeline, to simple options, such as setting colors for waveform trace displays.

The next design problem arises from the answer to the previous problem: now one has a system of significant complexity because of the large number of dimensions in which it is configurable. Most users will not have the time to learn to operate such a system. Moreover, one cannot be expected to set all these configuration options every time a new project is begun. What is desired is a mechanism whereby a group of *end-users* at a sequencing center are presented with a system that has been pre-configured by a *master user* who is expert in the workings of FAKtory. The end users see a relatively simple system, one that is essentially tailored to the production protocol employed at their site. The only preferences these users might wish to set themselves are those concerning colors, input file name conventions, or operating modes for the pipeline.

To achieve this, FAKtory uses the idea a *bilevel default scheme*. There are about a dozen different configuration panels in FAKtory, and the settings of each can be saved and loaded from either the *user* or *master* level. When a new project is first opened, FAKtory looks to see if there are any user-level defaults and loads any it finds for use in the new project. If any configuration options are still unset, FAKtory next looks at the master level for defaults specifying them and loads them if found. Finally, any remaining, unset configuration options are set to values hard-coded in the software. The execution of this default chain protocol on new project initiation has the following desirable properties. If a master user has established a configuration for his laboratory at the master level, then all end-users will see this configuration, unless they choose to establish their own defaults which always take precedence. This they may do in the parts where they desire to exert control by saving defaults at the user level. User level defaults are specific to each user, so each end-user sees only their customizations. Once a project has been created, its configuration settings are saved and loaded with the project thereafter.

2. Project Files: Making FAKtory Support Multiple Users

A *project file* is FAKtory's record of all work that has been performed on a particular sequencing data set. The project file contains all the sequencing data that has been entered, all configuration settings, and all assemblies and other work performed to date. An invocation of FAKtory works on one *active project* at a time. From a list of project files on the *FAKtory System* panel one may open a project making it the currently active project. All panel displays come to the screen exactly as they were when the project was last closed. One may close the currently active project at any time, either saving all work or discarding all work performed in the session. A new project may also be started at any time. The initial configuration for such a project is established as described in Section 1.

We designed FAKtory with the idea in mind that more than one user may be involved in determining a particular target sequence. First, all configuration information is classified as either project-based or user-based according to whether its setting affects the results obtained, or only the nature of the processing and presentation of data. For example, the stages in the pipeline, the fields in a fragment data entry, and the configuration of a prescreening stage of the pipeline are project-based, whereas the modes in which pipeline stages operate or the colors used by the system are user-based. Project-based configuration information is shared by all users working on a project, so if one user sets an option, all subsequent users see the changed option. On the other hand, a user-based configuration is specific to each user, i.e., FAKtory remembers each user's user-based settings separately. Thus if one user changes, say a color preference, other users on the *same* project do not see the change.

One may also open a FAKtory project in one of two modes: *read-only* or *edit-lock*. Only one user can have the project opened with an edit-lock at any given time. This user is free to enter, edit, and manipulate the data in the project. Any user can at any time open a project within read-only mode, but they are restricted to simply examining the state of the project. This mechanism prevents two users from simultaneously modifying a project at the same time. Finally, it is expected that users will use the file protections of the underlying operating system to insure the security of a project file and to restrict sharing to certain groups or individuals.

3. The FAKtory Fragment Database: Types, Vocabularies, and Default Input Values.

FAKtory can be configured to keep a wide variety of information about the results of each sequencing reaction, called here a *fragment record*. The name of each field in a fragment record is user-specifiable. Fields may be of type *string*, *integer*, *double*, *time*, or *waveform*. Time data is a date or time in one of several common possible formats. Waveform data is a special datatype specifically for 4-color sequencing reaction ladders as output by, say an ABI sequencer. Each field may be marked as being required or optional. Optional fields need not occur in external input files entered into the FAKtory database. The first field of every fragment record must be of type string, is required, and is assumed to be the designated *label* of the fragment. As such, the contents of this field are expected to be unique. In addition to a label-field, the last field of every FAKtory record must be of type string, is required, and is assumed to be the DNA *sequence* for the fragment.

To promote data consistency, one may specify a *vocabulary* for every field. For strings the vocabulary is a regular expression, and for numbers (i.e. integers or doubles) and time it is an interval (possibly open-ended) of the given type. When a vocabulary is specified, data entered into that field must match the vocabulary or a warning is signaled when the record is input. For example a field `Technician` might have `Joe|Susan|Arnie` as a vocabulary, a field `Date` might have `['June 1, 97','present']` as a vocabulary, and the sequence field, say `Sequence`, might have the vocabulary `'[acgtnACGTN]+'`.

For convenience, every field may have a designated *default input value*. On input, if the given field cannot be found in the input record, then this default value is assigned to the field. For example, it might be convenient to default a field for the technician to a particular individual or a field for the machine used to a particular machine. The value `present` denotes today's time and date if used as the default for a field of type time. In addition, the symbols # and @ generate unique positive integers and identifiers, respectively, when used in the default for the label field. For example, the default input value `frag#` will give un-labeled input records the labels `frag1`, `frag2`, `frag3`, etc.

The fields, types, vocabularies, and default input values for a FAKtory database are easily established in the `DatabaseCustomization` panel. Once data begins to flow into a particular project, the structure of the data records can no longer be changed. However, the vocabularies and default input values are not integral to this structure and can be changed at any time in the `InputPreferences` panel, which contains a mirror of the customization panel save that the field name and field type are not enabled. The default values are considered user-based preferences and are thus saved on an individual basis within the project (see Section 2 above). Thus each user may, for example, conveniently establish their name as the default value for the `Technician` field and the sequencing machine at their station as the default value for the `Machine` field. Because user-based preferences are saved on a user-by-user basis, the default values for each individual will be used when that individual is working with the project.

4. The Fragments Panel: Manipulating and Viewing Fragments

Once fragment records are brought into the system, one may manipulate the resulting database of fragments in FAKtory's *Fragments* panel. This is one of FAKtory's three primary panels, the other two being the *Layout Edit* and *Finishing* panels. Note that these three panels are generally the only ones seen or used by end-users.

The central portion of the large Fragments panel is a display area containing a vertically scrolled display of all the fragments in the database. By default this is just a tabular display of the labels of the fragments in alphabetical order. Ordinarily this would not be terribly useful, although the data set is generally small enough (several hundred to several thousand records), that scrolling through it looking for a particular fragment is quite viable (and we observe it being done). But there is much more. One can select any subset of the fields to be displayed. The display area automatically produces an appropriately tabularized display. If one asks for the sequence field to be displayed, an additional horizontal scroll bar appears, allowing one to scroll through the entirety of this rather length field. Thus one can get a snapshot of any cross-section of the data and scroll through it. The Fragments panel further permits one to sort the display on the basis of any field, resolving ties on the basis of a second selectable field, and so on. Technically speaking, one may lexicographically sort the display on any ordered subset of the fields. Thus if one wanted to quickly see the records produced by technician Joe during the month of February, one simply asks to sort first on the `Technician` field and secondarily on the `Date` field, and asks for the these two fields plus any others to be displayed. Simply scrolling through the display one can quickly arrive at the *contiguous* segment of the display containing the records of interest. If one wishes to see a nicely organized display of all the information of a given record, including the pipeline processing information described below, one merely double clicks on an entry in the display area and a window containing the information pops to the screen.

While one can quite intuitively navigate the fragment database with just the viewing and sorting capabilities above, FAKtory further provides an SQL-like query language for selecting subsets of the database whose description is more complex or which need to be selected frequently. The rules for forming queries are as follows:

- A *query* is a predicate or any boolean combination of said. The operator symbols used are: | for 'or', & for 'and', `-` for 'minus', and `!` for 'not'.

- A *predicate* may be a comparison between two operands using the conventional operator symbols `=`, `!=`, `>`, `<`, `<=`, and `>=`. The meaning of the comparison depends on the types of the operands: alphabetic order is used for string data, temporal order for time data, and numeric order for number data.
- The notations X `in` `[`Y`,`Z`]`, X `in` `(`Y`,`Z`]`, etc., are available as a shorthands for the predicates Y`<=`X`&`X`<=`Z, Y`<`X`&`X`<=`Z, etc.
- An *operand* may be either field names or constants.
- A numeric operand may also be any arithmetic expression composed from `*`, `/`, `+`, and `-` over numeric operands.
- The notation X `in` Y where X is an operand of type string and Y is a constant of type string, is a predicate which returns true only if X is a string matched by Y when it is interpreted as a regular expression.
- The built-in function `len(`X`)` returns the length of a string operand X.
- The built-in function `undef(`X`)` where X is a field name, is a predicate which returns true only if field X is undefined for the fragment record in question.
- A string operand may also be X`[`I`,`J`]` where X is a string operand, and I and J are integer operands. The result is the relevant substring of X where negative indices denote an index from the *end* of the string (e.g. `'abcde'[2,-2]` is `bcd`)

There are a few more additional forms to FAKtory's query language, these are deferred until the necessary concepts are introduced. The bullet list should make clear that the language is equivalent in power to the query language of a commercial relational database. However, it is simply implemented as an interpreted search. The performance for such is more than adequate as even a bacterial shotgun project involves only tens of thousands of fragments. The complexity and optimization of systems designed to handle tens of millions of, say airline reservation records, is not necessary in our context. Moreover, FAKtory's query language offers regular expression matching over string fields and string operators, something not found in conventional RDBMs.

FAKtory keeps track of a subset of the fragments in the database, called the *current fragment selection*. The current selection may be modified manually or by queries. All selected fragments are highlighted in the fragment display window, if visible. The utility of the current fragment selection is that it provides the set of fragment records to be operated on in a variety of circumstances. For example, which fragments to advance through the pipeline, or which fragments to add to an assembly.

There are buttons on the console to clear the current selection, to make it consist of all the fragments in the database, or to invert its contents. Clicking on an unhighlighted fragment record in the display, causes it to be added to the current selection. Clicking on a highlighted fragment, causes it to be removed. One may also sweep over a collection of fragments with the left or right mouse button depressed to add or remove them in a single step. In addition to these manual methods of altering the current selection, one may use FAKtory's query mechanism to do so. On can set the current selection to be the result of any query. This is done by pushing the console button labeled `Select Query ...`, where upon a dialog in which one can formulate a query expression pops to the screen. Alternately, one can union the results of a query with the current selection, or intersect it, or subtract it, using the appropriately labeled buttons. This permits one to build up the current selection in a number of independent steps.

Often one finds that they have need to perform a particular query repeatedly, or they would like to remember a particular set of fragments. FAKtory permits users to save the current selection as a named *fragment set*. If the current selection was produced solely using queries then a query is saved as the functional representation of the set. Otherwise there were some manual operations involved in arriving at the current selection, and an explicit list is saved as the representation of the set. To distinguish between these two cases, after a user creates a set with the name, say `Foo`, it appears in the list of user-defined fragment sets as `Foo()` if it was saved as a query (i.e. functionally), and as `Foo` if it was saved explicitly. Once a fragment set F has been defined, one may re-select it later by pushing the appropriate console buttons or by referring to the special predicate `%`F in a query expression. The important thing to note, is that the functional sets re-evaluate their query over the current database to arrive at the set, giving possibly different result over time, whereas the explicit sets always deliver the same list of fragments that were explicitly saved. In addition to user-defined sets, FAKtory provides builtin functional queries that select the fragments currently in a given stage of the pipeline.

5. The FAKtory Pipeline: Processing and Assembling Fragments

FAKtory realizes a general framework for controlling a process pipeline in which fragments pass through a number of processing *stages*, each of which augments a fragment's record. The pipeline may be customized to contain any number of stages of different types, and each stage may be individually programmed to perform a specific task, such as vector prescreening, low-quality trimming, common repetitive element detection, etc. Each stage can operate on fragments in one of three *modes* depending on the extent to which the user wishes to supervise its operation. Fragments can be *advanced* through stages of the pipeline, *rolled back* to previous stages for reprocessing, and *discarded* if found defective. Problems that occur during the processing of fragments can be reviewed and remedied during the operation or can be logged for later inspection. This framework is quite general and independent of FAKtory's specific application domain. It permits us to add additional types of stages within this framework as might be required by evolving DNA sequencing protocols or customer demand.

One may customize project pipelines to consist of any number of stages where the first stage is always an *Input* stage, and the last two stages must be an *Overlap* followed by an *Assembly* stage. The intervening stages can be of type *clip*, *vector*, *tag*, or *prescreen* and can occur in any number and order. The exact nature of these stages will be discussed later, but we note for now that the clip, vector, and tag stage types are all actually special instances of the more general prescreen stage type. Before data enters a project, setting up the pipeline is simply a matter of opening the *Pipeline Customization* panel and then adding or deleting cartoon boxes of the types of stages from a picture of the current pipeline. Each stage in the cartoon of the pipeline must be given a unique stage name. After the stages, their names, and orders have been settled on, each stage must be individually configured. Once fragments begin to flow through the pipeline, it is no longer reconfigurable.

During pipeline processing, a fragment may arrive at a stage whose operation cannot be applied to the fragment or whose operation is applicable but the results seem suspect. The former events are termed *errors* and the later *warnings*. Collectively they are called *exceptions*. As a fragment passes through a stage, the status of the operation on that fragment — `OK`, `Warning`, or `Error` — is recorded and attached to the fragment along with any diagnostic messages in the case of an exception. Fragments tagged with an error in a given stage cannot advance through later stages in the pipeline, but fragments tagged only with warnings can. At any moment FAKtory knows for each fragment: (1) the most recent stage it has passed through, and (2) the *stage status* and messages (if any) for every stage it has passed through. Item (1) can actually be used in a FAKtory query with the built-in function `stage()`.

Before a set of fragments is ordered to move through some pipeline stages, a user can select, on the Fragments panel, to monitor the process in one of three *modes* — `Auto`, `Supervised`, or `Manual`. In `Auto` mode, all processing takes place without any supervision from the user. Any errors or warnings are noted by placing them on FAKtory's global *review log*. The user is not otherwise informed that any exceptions have occurred. In `Supervised` mode, the user is presented with each warning or error situation, and given an opportunity to either trouble shoot it immediately or defer making a decision until later by noting it as in `Auto` mode. Finally, in `Manual` mode, the user inspects the results of every stage on every fragment, being given the opportunity to modify the results if desired. FAKtory pipeline processing proceeds stage-by-stage rather than fragment-by-fragment, so that in the two interactive modes, the manager can conveniently focus on a *review packet* of the fragments requiring examination for a given stage, one stage at a time, in pipeline order. One may also select to process fragments in a `Custom` mode in which case fragments pass through the stages in the, possibly different, modes specified for each on the *Pipeline Modes Preferences* panel.

The flow of fragments through a given pipeline is effected via controls on the Fragments panel. In a forward flow through the pipeline, called an *advance*, the fragments involved can either come from FAKtory's database or from external input files, or both, depending on the control settings. If some of the fragments are being input, then depending on settings in the *Input Preferences* panel, either (1) a dialog appears in which the user can specify the files to be input, or (2) FAKtory automatically imports all input files in a current directory (also settable) that have not yet been imported. If the source of fragments is the FAKtory database then the desired fragments should be the contents of the current fragment selection. To initiate the advance one simply picks a pipeline stage say S in the `Advance` menu. The selected fragments do not all have to be in the same initial pipeline stage, each fragment is advanced from whatever stage it last passed through until it passes through stage S. Fragments already in an error state, or those that become so during processing, do not proceed beyond the offending stage. If a selected fragment has already been through stage S then the command has no effect upon it. Fragments being imported through files always pass through the first *Input* stage whereupon they become part of FAKtory's database. Fragments may also be rolled back to previous states or discarded from the database (conceptually a roll back to a stage prior to *In-*

put). While fine-grained control of the fragment flow is possible with this schema, note that selecting to advance to the *Assembly* stage with the source specified (by default) to be all un-imported files, causes all available files to be input, processed, and assembled.

Both the review packets generated during interactive mode pipeline advances and the FAKtory review log of noted events, can be examined in order to oversee operations or make decisions about how to deal with pipeline exceptions. Both review packets and the global review log are conceptually lists of (fragment,stage) pairs we call *events*. These review lists can be processed with the *Review Summary* panel and *Review Events* panel. The Review Summary panel presents a review list as a scrollable list of the events with associated messages if they involve exceptions, further organized and displayable according to the stage(s) involved. The Review Events panel displays one event from the review list at any given moment, and permits the user to move forward and backward through the review list with a set of console buttons on its lower border. Within the Review Events panel is an *editing panel* specific to the stage for the event currently under examination. For example, for a clip-stage the editing panel shows the waveform or sequence of the fragment and the regions clipped by the stage. The user can modify the clip if desired. Thus the Review Summary is a way to manipulate the entire review list and the Review Events panel is a way to walk through a review list, editing the results of a stage on a fragment if desired. Note carefully, that the two panels work in concert: one can switch from one panel to the other and back again during the processing of a given review list.

To determine the disposition of the events in a review list, the user can mark each event with a status of `Note`, `Approve`, or `Discard` from either Review panel. When one exits either panel, thus ending a *review session*, all noted events are posted to FAKtory's review log (if they are not already on it), the fragments of all discarded events are discarded from FAKtory's database, and the fragments of all approved events are given a stage status of `OK` for the stage of the approved event and resubmitted to the pipeline.

Finally, in overview the review system functions as follows. During an interactive mode advance, the Review Events panel is brought up on each review packet as it arises, effectively initiating an interactive review session. The user reviews the events, possibly editing and then approving some events, marking others as discardable, and postponing making decisions on yet others by marking them as noted. If need be they can flip to the Review Summary view of the packet if it helps them visualize their decision process. The review session is completed by exiting the current Review panel. At that point, the events in the review packet are disposed of according to their event marks and the packet disappears. The other scenario involves a user, at a time of their convenience, deciding to review the events that have been posted to FAKtory's review log. This is done by pressing a button on the Fragments panel to invoke either of the Review panels on this special review list. The remainder and exit from the review session are identical to the interactive case.

6. Input/Output: Making FAKtory Extensible

Another key problem in producing a useful system is that of making sure it can handle a wide range of input and output data formats, and that it can be linked to other programs. To address this, FAKtory uses the idea of user specifiable *pre-* and *post-processors*.

FAKtory proper is capable only of importing files in a single, simple format that is a superset of the FASTA format. However, one of the configuration options for the Input stage of the pipeline, is the specification of a command-level program (and options if desired) to be used as a preprocessor for all file imports. Whenever fragments are to be input to FAKtory, it first calls this program (if specified) with each file to be input, and grabs the output of the program as the FAKtory-formated file to input to the system proper. That is, the program is assumed to be a translator from the given data format to FAKtory format. Anyone can write such a translation preprocessor whenever the need arises. For example, we receive data from different labs to use for testing purposes. It generally takes us less than an hour to write a program that will translate between the supplied format and FAKtory format. We have also written a translator, *ABI2fak* that takes ABI sequencer files, extracts basic information, the comment line, and the waveform, and produces a FAKtory input file describing this data. The translator and its specification in the *Input Preferences* panel can be done by a master user. End-users simply ask to import files, oblivious to the presence of this mechanism.

Exporting information is handled similarly. In an *Output Preferences* panel, one may specify a list of post-processing programs, assigning a menu name to each and specifying whether the program is to receive a consensus sequence or a multi-alignment in FAKtory's format. The list of menu names becomes the contents of an *Output Menu* that occurs on both the Layout Edit and Finishing panels. Selecting one of these names from the menu causes the corresponding post-processing program to be invoked on the currently selected assembly or contigs. The post-processing program can be any program capable of reading the FAKtory-formated information passed to it. Moreover, the program is run in a separate process from FAKtory so that the two can be running simultaneously. For example, we are currently preparing programs that will submit Blast searches, will log results with Terry Gaasterland's Magpie system, and run Grail on consensus sequences. The Grail postprocessor actually opens up its own window on the screen and runs concurrently with FAKtory.

In summary, the simple pre- and post-processor scheme we employ allows one to easily extend FAKtory to accept any input format and to run any post-analysis software concurrently with FAKtory. We will supply an initial small library of such programs, anticipating that users will contribute additional extension programs over time.

7. Prescreeners: A Unified Framework for Clipping, Screening, and Tagging

The pipeline must always consist of an initial Input stage that imports fragment records from the file system, an Overlap stage which computes all overlaps between fragment sequences, and a final Assembly stage that melds the fragments into a reconstruction of the target. Between the Input and Overlap stages may be any number and combination of Prescreener stages. In the design of FAKtory, we chose to develop a single, unifying framework in which one could formulate a wide range of criteria and recipes for clipping, screening, and tagging fragment sequences. This framework involves a set of five types of pattern recognizers and a small expression language for flexibly combining the results of these recognizers.

We start by describing such a general-purpose Prescreener stage whose configuration panel presents the user with the full power of the framework. A Prescreener stage consists of several *prescreeners*, each of which can be programmed to either cut off a 5'- or 3'-end of a fragment's sequence, or to tag substrings of the sequence with a specifiable color and symbolic name. The interval(s) of a fragment's sequence which will be clipped or tagged by a presceener are specified by an *interval expression* which is basically a pattern that *matches* a set of disjoint substrings, specified as intervals of character positions. We will describe interval expressions and the intervals they match in a bottom up fashion by starting with the simplest:

- An interval expression can be a *recognizer*. Each recognizer matches either a single interval or a collection of disjoint intervals of positions in the fragment sequence to which it is being applied. There are five types of *recognizers*:

    1. An *interval* recognizer is just a fixed interval `[`I`,`J`]`. For example, `[0,20]` matches the first 20 bases of a sequence, `[-20,-0]` matches the last 20 bases. One may also specify this in percentage terms, so `[0,20]%` matches the first 20% of the bases in a sequence.
    2. A *regular expression* recognizer is a regular expression, an error tolerance, and a designation that one wants the 3'-most, 5'-most, or all matches to the expression. The recognizer returns an interval or intervals that match the regular expression within the given number of errors. It is useful for short patterns such as restriction enzyme cut sites.
    3. There are several *signal* recognizers that match intervals of a sequence based on the measure of the signal/noise ratio, peak-height/max-height ratio, and peak width in a window of specifiable length.
    4. A *frequency* recognizer matches intervals in which the frequency of specifiable bases (including N's) is above or below a given level in a window of a given length.
    5. An *overlap* recognizer matches any intervals that overlap, within a certain match stringency, a reference sequence from a user-specified library of such sequences. The library typically contains things such as consensus Alu and Line elements, and vectors such as variants of PUC commonly used in the lab. These recognizers are useful for tagging repeats and identifying vector sequence that needs to be trimmed.

    These base recognizers are configured in a *Recognizers* sub-panel to each *Prescreener* panel that is dedicated to that purpose. Each recognizer is given a name so that it can then be referred to in an interval expression for a prescreener.

- Any set-theoretic combination, *X op Y*, where X and Y are interval expressions, matches the appropriate combinations of the intervals matched by X and Y. The operator symbols used are **|** for union, & for intersection, and `-` for minus. Also `!`*X* matches the complement of *X*'s intervals with respect to the fragment sequence.

- The expression *X*+*c*, where X is an interval expression and c is an integer constant, matches the intervals matched by X all shifted in the 3' direction by c positions. The expression *X*-*c* similarly shifts *X*'s intervals in the 5' direction.
- The expression [*X*,*Y*], matches *the* interval that starts at the 5'-end of 5'-most interval matched by X and ends at the 3'-end of 3'-most interval matched by Y. If X doesn't match anything then the expression is equivalent to [*Y*,*Y*], if Y doesn't match anything, to [*X*,*X*], and if both don't match anything then the expression doesn't match anything. One may also specify open intervals at either end by replacing `[` with `(`, and `]` with `)`.
- The expression *X* ? *Y* : *Z* matches Y if X matches something and Z otherwise. Similarly *X* ? *Y* matches Y if X is matches something and doesn't match anything otherwise, and *X* : *Z* matches Z if X doesn't match anything and matches X otherwise.
- The expression *X*(*Y*) matches the everything matched by X when evaluated over the substrings of the fragment's sequence matched by Y.

This simple interval expression language is sufficient to describe quite complex clipping or tagging criteria. For example, if one wanted to clip at the clone insertion restriction site, or after the first 50 bases if such a site cannot be found because of poor signal quality in the initial part of the read, then one can express this with the interval expression `[0,Site(Intv):Intv]` where `Intv` is the interval recognizer `[0,50]`, and `Site` is a regular expression recognizer for the cut site with say 1 mismatch allowed and optioned to return the 5'-most instance.

In designing the general Prescreener-type stages above, we again came up against the problem of the desire for generality resulting in a mechanism that required significant skill to utilize. Often, however, the full power and concomitant complexity of the full framework is not needed. To alleviate this problem, we set about designing simpler, specialized interfaces called Clip, Screen, and Tag stages that are sub-classes of prescreeners directly suited to expressing common clipping, vector screening, and element tagging functions. We give a quick overview of each of these special panels:

1. *Vector*: This panel is restricted to building a set of clipping prescreeners each of which is a single overlap recognizer. The design and layout of the panel have been tailored for this simple subset of the prescreener capability. In essence, the user is presented with a panel where they select the reference vector sequences they wish to screen out.
2. *Tag Panels*: This panel is restricted to building a set of tagging prescreeners, each of which is a single regular expression, frequency, or overlap recognizer that is automatically optioned to report all matches. Like the other panels its design and layout are tailored to present a simple interface to this subclass.
3. *Clip Panels*: This panel is restricted to building a set of 5' clipping prescreeners and a set of 3' clipping prescreeners each of which is a single interval, regular expression, signal, or frequency recognizer that is automatically optioned for matching the 3'- or 5'-most occurrence, respectively. The 5' clipping prescreener is guaranteed to clip from the start of the fragment sequence to the 3' end of its recognizers match (if any). Symmetrically, the 3' clipping prescreener clips from the end of the fragment to the 5' end of its recognizers match (if any).

We find that in practice these simple sub-classes suffice to express most of the preprocessing needed on fragment sequences before computing overlaps between them and then assembling them into contigs. Only on occasion is the full power of interval expressions required. As a final note, every clip, tag, and vector panel can be viewed as a general prescreener if desired. The prescreeners therein may then be modified using the more powerful console of the Prescreener panel. One may always flip back to the original subclass, *providing* the specification has not changed. This permits users to learn about the Prescreener by seeing how Clip, Vector, and Tag specifications are codified as Prescreener specifications.

8. Constraints: Making FAKtory Support Many Sequencing Protocols

One of the unique features of FAKtory is its ability to automatically generate and utilize a collection of constraints that model additional information about the particular protocol being employed to sequence a target DNA strand. For example, many labs sequence both ends of their clone inserts in which case one knows that the pair of fragment sequences are on opposite strands and should be within a certain distance from each other in any proposed assembly. As another example, in transposon-mapped sequencing, the two fragments primed from a given transposon have a 4 or 5 base-pair overlap and are on opposite strands (we say they have the opposite *orientation*). Moreover, fragments from adjacent transposons are expected to overlap and again be in the opposite orientation. The FAKII assembly kernel [MJAM99], upon which FAKtory is based, is currently the only software suite to provide a mechanism for describing protocol-induced constraints on the range of possible solutions, and to employ it, *a priori*, in computing its answers. All other systems either ignore such information or simply tell the users about violations of such constraints, *a posteriori*.

The basis of the constraint framework is that the additional protocol-dependent information can be expressed as a set of *overlap*, *orientation*, and *distance* constraints between *pairs* of fragment sequences. While we can't prove that every conceivable protocol can be expressed in such a framework, it is the case that all the protocols commonly in use today are expressible in this framework. FAKtory has a *Constraint* panel in which a user can specify and give a symbolic name to a constraint relationship between a pair of fragments. For example, one might define `Dual_ends` to be the constraint that a pair of fragments be (1) in the opposite orientation, and (2) that their 5' ends be a distance, say 800 to 5000 base pairs, apart. As another example, one might define `Nested_deletion` to be the constraint that the 3' end of the first fragment overlap the 5' end of the second. In general, a *constraint* may be any set of overlap, orientation and distance constraints that a user might wish to simultaneously be true about the relationship between two fragment sequences in an assembly. Overlap constraints may be oriented in the sense that the first must be 5' of the second. Moreover, the type of an overlap relationship can be controlled in a detailed way, and distance relationships can be with respect to any *anchor* position relative to the fragments. Users must also assign a color to every constraint for later display in assemblies.

Now that one can define different types of constraint relationships between fragments, the problem arises as to how one designates that a particular pair of fragments has such a constraint between them. FAKtory's solution is embodied in the idea of *matchers* that automatically associate a constraint between two fragments on the basis of their labels. For example, it is common practice to label sequences obtained from opposing ends of an insert something like, `Aname.f` and `Aname.r`, and if one later produces a long read of the `.f`-end of the insert with special chemistry or a different machine, to label that fragment something like `Aname.fi`. As another example, when doing nested deletion sequencing, one typically labels successive reads as, say `Dname.100`, `Dname.101`, `Dname.102`, and so on. Clearly, with such labeling schemes in place, one should be able to build a simple pattern matching mechanism that identifies appropriately labeled pairs of fragments and associates a given constraint relationship between them.

Such mechanisms can be set up on the *Constraint Matchers* panel in the following way. One can bind any capital letter, called a *variable*, to a regular expression. One may then specify a *binding template* that is a regular expression where some elements may be variables between square brackets, e.g. `[A]`. Finally, one specifies a *mating template* that is a string some of whose elements may be the special forms [*X*], [*X*+*i*], or [*X*+*i*!]. The variable X must have been used in the binding template, i is an integer constant, and the later two forms are permissible only if X's regular expression matches sequences of digits. A matcher consists of a template pair and a constraint relationship defined previously on the Constraint panel.

When it is time to infer which constraints apply to which pairs of fragments (see Section 9), the mechanism works as follows. The binding template of a matcher is applied to a fragment's label to see if it matches. If it does, then the exact substring of the label matched by each variable in the binding template is determined. A mate label is then generated by replacing every variable reference in the mating template with the corresponding substring from the binding match. In the case of the [*X*+*i*] and [*X*+*i*!] forms, the integer i is added to the value of the number denoted by the substring, and in the later case, leading 0's are preserved. Once the mate label has been determined, FAKtory searches for a fragment whose label coincides and if found, it then designates this fragment and the one whose label matched the binding template and associates the matcher's constraint to them.

For example, suppose `A` is bound to `[a-zA-Z]*` (matches any sequence of letters) and `N` is bound to `[0-9]*` (matches any sequence of digits). First, consider the template pair: `[A].f` and `[A].r`. The binding template matches `Aname.f` and the corresponding mate label is `Aname.r`. Next, consider the template pair: `[A].[N]` and `[A].[N+1]`. The binding template matches `Dname.100` and the corresponding mate label is `Dname.101`. Also, the binding template matches `Dname.001` and produces mate `Dname.2`. If one wishes to keep the leading 0's in the mate label then one should use `[A].[N+1!]` as the mating template, in which case the mate would be `Dname.002`.

While FAKtory's matching mechanism is not a universal panacea for the problem of automatically inferring constraint relationships, it is powerful enough to capture a wide range of existing data sets, and certainly more than adequate for future projects where labeling conventions can be chosen to permit its application.

9. The Layout Edit Panel: Manipulating and Viewing Assemblies

FAKtory maintains a master *layout cache* of possible assemblies which may be viewed and individually manipulated in the *Layout Edit* panel. Like the Fragments and Finishing panels, this is one of FAKtory's three main *working* panels where one actually processes data, as opposed to configuring things in order to process data. We refer to assemblies as *layouts* in this panel, as the intent here is to manipulate and study the assemblies at the level of how the fragments have been arranged, rather than at the level of the individual bases in them which is the domain of the Finisher panel (see Section 10). Each fragment is always represented in this panel as a line or stick whose proportionate length reflects that of the fragment's sequence, and whose position in the *stick layout* depicting the assembly shows how it interacts with all the other fragments. The focus of this panel is to allow a user to experiment and interact with the space of all possible assemblies of the data by adding and deleting constraints, manually locking, splitting, or joining contigs, and adding new fragments to existing assemblies. FAKtory is unique in its concept of allowing a user to manipulate and experiment with a number of distinct assembly objects.

One or more assembly objects are added to FAKtory's layout cache whenever a user advances a set of fragments through the final *Assembly* stage of the fragment pipeline. The exact process is as follows. First, any configured constraint matchers (see Section 8) are applied to the given set of fragments and the specified constraints attached to specific pairs of fragments. Then the FAKII assembly kernel is invoked on this set of fragments, producing up to k-best assemblies where k can be specified in the *Assembly Preferences* panel that contains a number of other useful configuration options for controlling the assembly. These assemblies are then added to the layout cache. Note carefully, that only the subset of fragments advanced through the Assembly stage are passed to the kernel for assembly, which is not necessarily every fragment in FAKtory's database. Furthermore, every such advance produces new, distinct assembly objects to be added to FAKtory's layout cache.

The top portion of the Layout Edit panel is designed for viewing the set of layouts currently on the layout cache. It consists of a scrollable display of these assemblies that can present them in a number of visual styles. First, each assembly can be presented as either a stick layout, a coverage plot, or a 3'/5' coverage plot, with or without summary statistics for each contig. One may zoom in and out on these presentations in either dimension of the graphic display. One may optionally view the parameters that were used to produce the assembly, the number of contigs in it, and the number of fragments that did not overlap with any others. Finally, one may also compare one assembly to all the others in the list by clicking on the assembly and then on the `Compare` button. Each contig of the selected assembly is distinctly colored, and the fragments in every other assembly are colored according to the color they have in the selected assembly. This simple mechanism allows one to easily see rearrangements in the data between solutions.

Conceptually layouts consist of an assembly *and a set of constraints*. As described above, layouts can be generated and added to the layout list by advancing fragments through the Assembly stage of the pipeline. One may also delete layouts from the layout cache. A third and powerful option, is where one edits the set of constraints for a layout and then *reassembles* it, possibly with some new fragment sequences if desired. Precisely speaking, the reassembly of a layout consists of taking (1) the fragments in the layout, (2) the current set of constraints for the layout, and (3) possibly some new fragments and concomitant new constraints inferred by matchers, submitting these to the FAKII assembly kernel, and adding the resulting assembly objects to the layout cache. This reassembly mechanism can be utilized in a number of ways as illustrated below.

The lower portion of the Layout Edit panel is the layout editing area. One can select a layout from the master layout cache in the top portion and load it into the editing area. Thereupon one may *split* some contigs apart, and *join* others along ''weak'' overlaps between their end fragments. Both of these operations are achieved by (an implicit) reassembly. One may also *lock* sets of fragments together, which adds constraints to the layout so that it is guaranteed that these fragments will always be in the same relative positions with respect to one another if ever reassembled. One can visualize the constraints that are in force for the layout in the editing area, and can either add or delete constraints individually for this layout. The resulting layout may then be put back into the layout cache, either replacing its former instance or as a new layout object. Often, however, what one will do after locking fragments and editing constraints, is to hit the `ReAssemble` button to see how the fragments assemble under the modified constraints. The results appear in the layout cache where they may be discarded or kept according to the user's evaluation of the results. Such an explicitly requested reassembly will also involve any fragments in the current selection of the Fragment panel, permitting one to incorporate new data into the assembly, if desired.

In summary, our idea is to create a ''scratch pad'' where one can experiment and interact with layouts in a very general way. One can remove or add some constraints, and then reassemble to see what they get. If the result looks right, the user can keep it, and if not, discard it. We finish with a couple of examples that hopefully convey the flexibility of the assembly/reassembly cycle provided by the Layout Editor. Late in a project, a few reactions are resequenced or some gaps are filled by PCR. One can lock the entire the current solution and reassemble it with the new data, in which case the new fragments will be incorporated without disturbing the existing assembly. In the final example, one may have several dual-end sequencing constraints that are actually incorrect due to lane tracking problems. One can resolve the problems they cause by deleting suspect constraints and seeing how the solution changes without them.

10. The Finishing Panel: Optimizing and Keeping Track of Edits

Once a user has an assembly of the fragments that they are confident is correct, the final step in producing a finished sequence is to examine the individual bases in the assembly and resolve discrepencies due to errors in the sequencing reactions. To do so in FAKtory, one must designate one of the layouts in the master layout cache as being *finishable*. Only one layout in the list may be designated finishable at any given time. When open on the screen, the *Finishing* panel displays and operates upon the currently finishable assembly in the layout cache. The display contains a text window on a section of the multi-alignment of a contig of the candidate, a stick layout of the candidate with a box around the part currently in the text window, and if available, a scrollable set of waveforms for the fragments in the text window. This is very similar to the Staden Gap4 model, a model that has received high marks, is widely accepted, and is very natural.

Apart from this basic similarity, our finisher does differ in the details of its operation. First, the waveform area is a vertically scrollable canvas of *all* the traces relevant at the currently selected column of the multi-alignment. As expected, traces are always aligned with the currently selected column. FAKtory uses its own internal compression method for storing waveforms so that a typical 600 base pair trace requires only 7kb of space. FAKtory has an automated scanning mechanism that advances a user to the next ‘‘problem’’ column for editing, where the notion of a problem column is configurable in the *Finisher Preferences* panel. As such an editing sweep takes place, FAKtory records the regions of the finishable assembly that have been so scanned. Such intervals are called *edited regions* and are retained as an integral part of the finishable solution. A user will never be directed to such sections again by the automated editing scanner. Edited regions are highlighted on the stick layout of the candidate so a user can get a quick idea of their progress on the finishing task.

We paid particular attention to developing a protocol for an editing sweep that attempts to minimize the number of key strokes involved. We did so because it is clear that good ergonomics are essential to making this tedious task move as quickly as possible. Hitting a tab advances one to the next problem column flagged by the editing scanner and places the cursor on the consensus row of the column. Typing a letter with the cursor on the consensus character, turns every character in the column into that letter, i.e., the user is asserting that the letter is the correct call for the column. Hitting a carriage return moves the cursor from the consensus sequence row of the multi-alignment to the corresponding symbol of the first sequence. Subsequent carriage returns move the cursor to successive sequences allowing one to cycle through the sequences, making individual changes by simply typing a symbol. Hitting a tab in the middle of such a column tour moves one to the next problematic column. Backspace undoes the last edit. Shift-tab jumps to the previous problem. Delete removes the current column, space inserts a column of dashes, and so on.

When a user has finished an editing session, quitting the Finisher panel saves all edits as part of the assembly. One can also quit the session and discard all work done if desired. Finally, one can at any time ask FAKtory to reflect the edits made to each fragment, back to its record in the FAKtory fragment database. Such a request will affect all future assembly generation steps as the edited sequences will be used. Moreover, these changes need not replace the current sequences, but instead may result in a new *version* of the sequences. FAKtory supports a linear history of revisions of each fragment’s sequence that may be controlled from the Finishing panel. Revisions may be removed at will, save for the original sequence which can never be overwritten.

One last and interesting design problem arose because of the fact that a FAKtory project can contain many layouts. One may select an assembly as being finishable, embark on finishing it, and after making significant progress, discover that this layout is not correctly assembled. Alternatively, a user may discover they need to reassemble the layout with some additional data. To handle this, FAKtory allows one to designate another assembly as being finishable. However, this is not a light-weight process but involves the transfer of the edits and editing regions recorded for the current finishable assembly to its successor. Regions where the data is assembled differently or where new data is present, and their immediate surroundings must be marked as unedited as the user has not examined the effect of the reassembly. In other words, FAKtory does everything it can to preserve the prior editing work, but inevitably some regions will need re-examination. Because the step can result in a loss of work, the user is advised to carefully contemplate the consequences of changing which layout is the finishable one.

11. References

[MJAM99] Miller, S., Jain, M., Anson, E., and Myers, E., "Interface for the FAKII Fragment Assembly Kernel". Technical Report 99-01, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721, 1999.