CELAMY(1) **CELAMY(1)**

NAME

celamy – Celera assembly layout viewer and analyzer.

SYNOPSIS

celamy <specfile> . . .

DESCRIPTION

Celamy reads from one or more *specfiles* the specification of a layout of line segments and opens a window with a view of the entirety of the encoded layout. *Celamy* is a work in progress. Currently it can read spec files encoding all of the currently anticipated features as described below, but only displays line segments, marks, and link groups. It further is only capable of zoom, pan, selective display of segments and marks, and canvas picks.

In the future, one will be able to ask for statistics and displays of a given facet of the layout communicated in the form of a query, a preliminary specification of which is given below. Moreover, the features for clusters and overlap displays will all be added over time.

1. Specification File Syntax and Semantics:

A *celsim* input file consists of a sequence of lines where each line contains the definition of either a (a) category or style, (b) a category application loop, (c) a layout piece, (d) a link group, or (e) an overlap pair. Formally,

```
<spec.> <- ( <category> | <catloop> | <piece> | <link_group> | <overlap> )+
```

Both categories and pieces have a unique names assigned to them. These names are alphanumeric strings that must start with a digit. One need not be concerned with reference-before-definition issues, as all references to such name are resolved after all input files have been read. One should feel free to put items in any order convenient to the producer of the specification. In the event that one submits several files to *celamy* the result is the same as if the files had first been concatenated into a single larger file.

Every line except category application loops can end with an optional comment or label (see the syntax of each statement below). Such a comment starts at the first occurrence of a #-sign and continues to the end of the line. Leading and trailing white space is removed from the label and it is stored with the defined item and utilized in the display as described later when the behavior of the interface is detailed.

Category Definitions:

A category definition consists of its name followed without any intervening white space by a colon. Thereafter it consists of a sequence of graphic attributes specifying a color, thickness, or a drawing style of solid or dashed. Spaces may occur between these items, but no spaces should occur within any given item.

```
<category> <- <name>":" <attribute>+ ('#' <label>)!

    <attribute> <- "C[0-9A-Fa-f]*6" | "T"<positive int> | "D" | "S"
```

A color is specified by a 'C' followed by exactly 6 hexadecimal digits. Each pair of digits specifies an intensity level between 0 and 255 for the red, green, and blue components of the color, respectively. For example, 'CFF0000' is full intensity red, 'C00FF00' is full intensity green, and 'CFF00FF' is full intensity purple. A thickness is specified by a T followed by a positive integer. For example, T3 specifies that objects in the given category be drawn with a 3-pixel thick brush. Finally, specify a D if dashed line drawing is desired, and an S for solid lines.

There are three global registers that record the last specified color, thickness, and line style. If in a category definition one of these items is not specified, then the last recorded value is taken. For

example, if one says '0: CFF0000 T2 D' and the next category definition is '1: T3', then category 1 items will be red, dashed, and three-pixels thick. The three attribute registers are initally set to white, one-pixel thick, and solid, respectively.

Category Application Loop:

The full power of a category application loop does not become apparent until we discuss layout piece definitions below, but we give the initial idea here. Basically, we wish to realize an implicit method for categorizing a series of segments constituting a layout piece. A category application loop is a list of references to defined category types. A given category can occur on the list an arbitrrary number of times. A category application is always in effect, initially it is a list consisting of the single default category, which is white, solid, and 1-pixel thick. The current category application loop is reset whenever line with the following syntax is encountered:

```
<catloop> <- "ATT:" <name> +
```

The category loop is set to the list of categories given by the sequence of category names following the "ATT:" prefix. This category loop becomes the current one and is applied to all subsequent items until the next category loop definition occurs.

Layout Piece Definitions:

A piece definition consists of its name followed immediately by a colon, followed by a segment list, and terminated with an optional cluster designation and/or an optional row allocation constraint. A segment list consists of an increasing sequence of coordinates where one may intersperse a reference to a category between any two consecutive coordinates and one may optionally identify some of the interior coordinates as marks by placing an 'M' immediately in front of them (no intervening space please). Each category reference consists of the letter 'A' immediately followed by the category's name. All coordinates must be non-negative numbers. Formally we have the grammar:

```
<line> <- <name>":" <segmentlist> ("R"<#>)! ("C"<name>)! ('#' <label>)!

    <segmentlist> <- <coord> ( ("A"<name>)! "M"!<coord> )* ("A"<name>)! <coord>
```

Suppose a piece definition specifies the list $x_1,x_2,\ldots x_n$ of coordinates where $x_i \leq x_{i+1}$ and $0 \leq x_1$. Then *celamy* will allocate the interval $[x_1,x_n]$ to a row of its display so that this interval does not overlap the interval of any other piece also allocated to that row. The piece will be displayed as a series of segments [x1,x2],[x2,x3], … [xn-1,xn] where the category or style of each segment is determined either implicitly through the current category application loop, or explicitly through an attribute reference between the two delimiting coordinates. For example, the definition "1: 100 A3 200 A1 300 A2 500" defines a piece covering the interval [100,500] where the segment [100,200] will be displayed in the style of category 3, the segment [200,300] in style 1, and the segment [300,500] in style 2.

The preceding example illustrates the explicit specification of categories for each segment. Whenever an explicit category is not given between two consecutive coordinates for a segment, the category is fetched from the current category application loop as follows. Suppose that the loop is set to a_0, a_1, … a_{k-1}. Then the i^{th} fetch gets category $a_{(i-1) \bmod k}$. That is, each category in the list is taken in turn and when the list is exhausted, one starts over at the beginning of the list, hence the use of the term "loop". For example, if the most recent category application loop setting was "ATT: 2 1 3" then for the definition "3: 100 200 A0 300 400 500 A0 600 700", the category assignments to segments are: category 0 to [200,300] and [500,600], category 1 to [300,400], category 2 to [100,200] and [600,700], and category 3 to [400,500]. As an example more typical of a real application context, one might have the category application loop "ATT: 0 1" in force where 0 is a solid, 2-pixel thick, red line and 1 is a dashed, 1-pixel thick, green line. Thereafter, definitions of the form "<id>: x1 x2 x3 x4" will result in a piece that looks like two red "reads" (the intervals [x1,x2] and [x3,x4]) and a dashed "mate" connector between them (the interval [x2,x3]).

The final feature of layout pieces is that one may additionally specify any number of marks on the piece. By simply placing an 'M' immediately in front of an interior coordinate (i.e. neither the first nor

the last coord in the list), one is signifying that a tick mark should be drawn at that point in the line. The category/style of the tick mark is given by the explicity category immediately proceeding the coordinate, or the category that is implicitly placed there by the category application loop mechanism. For example, the definition: "23Line: 100 A3 M200 300 400 A3 M500 600" draws a layout piece equivalent to "100 300 400 600" as drawn by the category application loop with tick marks in style 3 at coordinates 200 and 500.

At the end of a piece definition one may optionally place a row allocation constraint of the form 'R#' or designate the piece's membership in a cluster with a clause of the form 'C<name>'. A row allocation constraint asserts that the given piece should be placed in the specified row of the display if at all possible. The rows in the display are number from 0 on up to however many are required, with row 0 at the top of the display and subsequent rows underneath it. If a row allocation constraint cannot be met than it is placed on the next available row. It is guaranteed that if two pieces overlap an both have row constraints, then the one with the lowered numbered row constraint will appear in a lower numbered row. One example of the use would be to place a facsimile of an entire genome in row 0 of the display, where each repeat element type is shown in a unique category style. Such a genome is so long that one cannot describe it as a single piece, but one can describe it as a series of pieces all constrained to be in row 0. Another example of the use of row constraints is to stratify the display. One might want for example all the pieces of a given category to appear in the uppermost rows and those of other pieces below them. Constraining all the strata pieces to row 0 will have the desired effect (even if some of them overlap). One can of course develop several strata by using constraints to the desired number of rows or strata.

Each piece can optionally belong to a cluster. Each cluster is assumed to be given a unique name. Effectively, the cluster option permits one to *partition* the pieces into a set of disjoint clusters. The exact effect in terms of drawing is not fully delineable at this time, but the basic idea is that each cluster will be allocated in a set of contiguous rows so that the pieces will an identifiable grouping in the display.

Link Group Definitions:

A link group is defined by following the keyword 'LNK:' with a list of the piece names to be put together in a group, optionally followed by a category type.

```
<link_group> <- "LNK:" <name>+ ("A"<name>)! ("#" <label>)!
```

Using a bus-bar display style, *celamy* will draw a busbar that contains drops to each member of the link group. The bus-bar and drops are displayed in the style of the specified category (or the first style in the current category application loop if non is given.)

Overlap Definitions:

An overlap pair is defined by following the keyword 'OVLP:" with a pair of piece names, optionally followed by a category type.

```
<overlap> <- "OLP:" <name> <name> ("A"<name>)! ("#" <label>)!
```

The assumption is that the two intervals of the two pieces actually overlap. If such is the case, then *celamy* will attempt to allocate the pieces in consecutive rows and fill the space between the overlapping parts with the color and style of the specified category type.

Object Display:

Note that in total there are four types of displayable objects that can be defined in a celamy specification file: segments (of pieces), marks, link groups, and overlaps. *Celamy* creates a menu for each class of objects where each menu contains a list of the categories that have been applied to the particular object type. One may selectively turn on and off the display of each type of object in each category by either releasing the menu over the selected item or clicking the menu open and then clicking buttons on the menu on and/or off as desired. The first word of the comment label of each category is displayed in the menu.

One can pick objects in the drawing area. We describe the effect of each event type/object combination:

- Left-Button-Down / Piece. The boundary coordinates and length of the segment under (or nearly under) the cursor are displayed in the textbox. When the button is released, the information disappears. If the cursor is within 3 pixels of a mark, then the position of the mark is displayed. This takes precedence over displaying segment coordinates.
- Shift-Left-Button-Down / Piece: Same as above except marks are ignored.
- Control-Left-Button-Down / Piece: The coordinates and length of the selected piece are displayed.
- Right-Button-Down / Piece: The comment/label for the piece is displayed.
- Left-Button-Down / Link-Bus-Bar: All pieces of the link group are hilited in the text color and the number of pieces and span of these pieces is displayed in the text area.
- Right-Button-Down / Link-Bus-Bar: The comment/label for the link group is displayed.

One can zoom in on a particular portion of a display by dragging the mous over a segment of the horizontal axis. A drag consists of a button press, possible movement of the mouse, and a release. The axis coordinates at which the press and release occurred are captured and celamy zooms that segment of the axis to fill the entire horizontal axis. The drag must begin at or below the horizontal line of the axis drawing. If one simply clicks with no intervening mouse movement, then celamy zooms in by a factor of 2 centered on the coordinate where the click occurs.

2. Celamy Queries:

In the text area at the bottom of the celamy window, one can enter a query to gather statistical information about a layout. To activate the text area, move the cursor into the text area. One can type an edit a query with the customary wysiwyg mechanics. The syntax of a query is as follows:

```
    <query> <- <query> "[|&-]" <query>
             | <query> "?[ixc]" <query>
             |  <query> "[+-><]" <int≥0>
             | "!" <query>
             | <query> "_" <int≥1>
             | "(" <query> ")"
             | <name>
             | $<name>
             | "@" <obj_qry>

  <obj_qry> <- <objb_qry> "[|&-]" <obj_qry>
             | <obj_qry> "?[ixc]" <query>
             | "#" <obj_qry>
             | <obj_qry> "~[cn]?i?" "\""<regular_expr>"\""
             | <obj_qry> "[><]" <int≥0>
             | "!" <obj_qry>
             | "(" <obj_qry> ")"
             | $<name>
             | <name>
```

The atoms of a query are names or words chosen from those in the segment, mark, and link menus, or the names given to saved queries preceded by a dollar-sign (see “Query Manager” below). In parsing a query the precedence order is _, <, >, +, or ~,then @ or !, then ?i, ?x,or ?c, then &, then -, and finally |. When precedence is tied, grouping is left to right. *Use parenthesis when in doubt !*

Object Queries:

There are two distinct facets of a layout over which one can query: objects and coverage plots. Any query or portion of a query to which the @ operator is applied is an object query. The result of such a query is the set of line segments and/or link items satisfying the query. For example, @ (Red | Green > 500), returns the set of all objects that are either (1) Red or (2) Green and over 500 units long. Note carefully that the universe of all objects is links and line segments, so saying @ ! Red for example, where Red is say a line segment type, returns *all links* and all line segments that are not

Red. The logical operators and (&), or (|), minus (-), and not (!) have their usual interpretations. The operators greater than int (> <int>) and less than int (< <int>) select those objects on the l.h.s. that are strictly greater or less than the given length, respectively. The matches operator (~) selects all those objects whose line name (if a line segment) or label matches the regular expression on the right hand side. The letters "c", "n", and "i" immediately after the ~ qualify the matches as follows: "c" – try matching only the label, "n" – try matching only the line name, and "i" – case does not matter. Without either the "c" or "n" qualifier, celamy tries matching both the line name and label. See the description of regular expressions given later in this section. The prefix "#" operator produces the set of line segments that are linked by a link item in its operand. Finally, the "?i", "?x", and "?c" operators select those objects from the set on the left, that are contained in, intersect with, or contain, respectively, an interval of the coverage plot on the right.

Coverage Queries:

Any query operators not within an @ subexpression operate on a coverage plot. A coverage plot is a rectilinear plot whose axis is the same as that of the layout celamy is displaying. At each coordinate the coverage plot has an integer coverage depth. An interval of a coverage plot is a maximal contiguous segment for which the coverage depth is non-zero. An interval of depth d of a coverage plot is a maximal contiguous segment for which the coverage depth is d or more. For a set of line segment and link objects, the coverage depth at each coordinate is equal to the number of objects spanning that point. A line segment spans the coordinates from its start point to its finish point, inclusively, and a link object spans the coordinates from the start point of its leftmost line to the finish point of its rightmost line, inclusively. So in this case, a coverage plot corresponds to exactly what one would expect for an assembly of line segments. An object expression is converted to a coverage plot whenever the next operator to be applied to it is not part of the object expression.

Logical combinations of coverage plots permit one to inquire about the intervals of the layout that satisfy the boolean query. For example '`Yellow - (Red | Green)`' would be an inquiry about the intervals of the layout that are covered by yellow but not red or green segments. Moreover, the operators also combine the coverage depths in a meaningful way. Let c(a,x) be the coverage depth of subexpression a at some point x on the axis. Then inductively:

- `c(a&b,x) = min(c(a,x),c(b,x))`
- `c(a|b,x) = max(c(a,x),c(b,x))`
- `c(!a,x) = if c(a,x)>0 then 0 else 1`
- `c(a-b,x) = c(a&!b,x)`
- `c(a_i,x) = floor(c(a)/i,x)`
- `c(a+i,x) = max(c(a,y):y ∈ [x-i,x+i])`
- `c(a-i,x) = min(c(a,y):y ∈ [x-i,x+i])`
- `c(a>i,x) = max( d : ∃y ( x ∈ [y,y+i] and ∀z ∈ [y,y+i], c(a,z) ≥ d))`
- `c(a<i,x) = if "length of interval containing x" < i then c(a) else 0`

Intuitively, the query '`a_i`' effectively is asking for regions where the coverage is `i` or more. The "+" and "–" operators can be thought of having the effect of expanding or contracting every interval of each depth of the coverage plot by the given amount. If in the case of "–" an interval at some depth becomes negative, then it disappears. The > and < operators keep only those intervals that are greater or smaller than a given length. There is assymetry in these two definitions as it makes no sense to, for example, keep a very small interval at depth 5 if the interval (at depth 1) that it rests on top of is too big. So for this operator, we chose to keep the coverage profile of an interval of positive coverage if the length of the interval is less than the given threshold. We suggest you try a few examples and observe the coverage plots that result.

Finally, the "?i", "?x", and "?c" operators on coverage plots select the intervals at each depth of the plot on the l.h.s. that are contained in, intersect with, or contain, respectively, an interval (of depth 1) of the plot on the r.h.s. As for the ">" and "<" operators, the definition of "in" must be assymetric for the same reasons. Thus an interval at some depth is retained only if its parent interval at depth 1 is

contained in an interval of the plot on the r.h.s.

Regular Expressions:

Celamy regular expression are exactly the same as those recognized by egrep plus an extension designed to facilitate matching numeric ranges. The basic egrep matches the following patterns, where in the term 'character' excludes the new-line symbol:

- A \ followed by a single character other than newline matches that character.
- The character ^ matches the beginning of a string (i.e., name or comment).
- The character $ matches the end of a line (atom name/comment).
- A . (period) matches any character.
- A single character not otherwise endowed with special meaning matches that character.
- A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in 'a-z0-9'. A] may occur only as the first character of the string. A literal – must be placed where it can't be mistakend for a range indicator.
- A regular expression followed by an * (asterisk) matches a sequence of 0 or more matches of the regular expression. A regular expression followed by a + (plus) matches a sequence of 1 or more matches of the regular expression. A regular expression followed by a ? (question mark) matches a sequence of 0 or 1 matches of the regular expression.
- Two regular expressions concatenated match a match of the first followed by a match of the second.
- Two regular expressions separated by a | match either a match to the first or a match to the second.
- A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [], then *, +, or ?, then concatenation, and finally |.

Celamy regular expressions have three additional features. First, the symbol '#' is a short hand for the regular expression "0|[1-9][0-9]*", i.e., that matches any positive integer constant. For example the pattern "CA#" matches "CA1", "CA13", and "CA203" but not "CA033". To match the latter use the pattern "CA0*#". Second, the symbol '@' is a short hand for "[_A-Za-z][_A-Za-z0-9]*", i.e., any C-identifier. Probably not too useful in the context of Celamy, but comes with the # symbol by way of another application domain for which Celamy regular expression package was designed. Finaly, and of most importance, is support for a "number class", denoted "'{' {integer_1} . {integer_2} '}'". The first integer must not be greater than the second. The number classes matches any sequence of digits denoting a number in the range of the class. For example "{3-56}" matches "3", "4", ... "9", "10", "11", ... and "56". The integers may be padded on the left with zeros in which case the padding is considered to be part of the matching string. For example, "{003-056}" matches "003", "004", ... "009", "010", "011", ... and "056".

Action of Queries:

To activate the query and see the results, press the 'Query' button to the left of the text area. A valid object query results in (1) the production of a window display a histogram of the lengths of all the objects satisfying the query, (2) the production of a window displaying a coverage plot of all the objects satisfying the query, and (3) every selected item is recolored the default hilite color, white. A valid coverage query results in (1) a histogram of the lengths of all the intervals in the coverage plot, and (2) the coverage depth along the layout for the given query. Currently the histogram result is fully functional and the coverage window is complete save that there is still no way to control the scale of the vertical axes. The coverage window is sized in the horizontal dimension to exactly match the canvas of the main window and any scrolling or zooming in the main canvas will be immediately reflected in the coverage window. One may have histogram and coverage windows from

several queries open simultaneously. A given result window only disappears when it is explicitly closed.

A couple of tricks: If you desire a coverage histogram and plot of an object query XXX, simply saying, XXX _ 1 will force the query to be considered a coverage query without alterring the result. If you want a coverage plot that just shows the intervals of non-zero depth, then apply negate (!) twice.

Status: Queries now function over both line and link items, but still not marks.

Celamy Tools:

The "Tools" menu of the celamy main window has options to allow a user to (a) set the dimensions of the canvas in pixels from the key board, (b) set the portion of the layout in view from the key board, and (c) save, reuse, and control queries.

View Range Setting:

Selecting the "View…" option brings up a window with two text boxes in which one can type the range of coordinates one would like to see in the canvas. Tabbing moves the cursor from one box to the other and hitting a carriage return is equivalent to pressing the "Set" button, so one's hands never need leave the keyboard. The desired range is displayed in the viewer with a little bit of padding on each side so the selected endpoints are interior to the view. Whenever one changes the view on the main window, the numbers in the text windows change to reflect the mouse-selected range.

Canvas Sizing:

The "Canvas…" option brings up a window very much like that for "View…" with the same ergonomic behavior. In this case the two numbers set the width and height of the canvase area in pixels. This is useful if one wishes to do screen capture and wants a consistent window size.

Query Manager:

The "Query…" option brings up the query manager window wherein one can save and re-execute queries after optionally setting buttons that control their action. Initially the window is empty save with a frame of control buttons at the top and a row of column labels for the entries that will be added, one per row.

To add a query to the manager window, type the query in the query text box on the main display window. Then press the “Add” button in the query manager. A row of widgets and text will be appended to the list of queries being saved. From left to right, there is (a) a radio button which is set when the given query is the one to which a manager control button is to apply, (b) a colored button showing the hilite color that will be used for object hiliting (initially white and changeable by clicking the button), (c) three small option buttons controlling whether a histogram (“H?”), coverage plot “P?”, or show hiliting (“S?”) will be produced when this query is re-executed, (d) a textbox wherein the query can be given a name, and (e) the text of the query (uneditable).

To delete a query from the manager’s list, simply select the query by pressing on its radio button at the far left and then press the “Delete” control button. To re-execute a query, select its radio buttion and then press “Query”. Histogram and Coverage Plot windows will appear according to the display option settings, and the objects of an object query will be hilited in the selected color if the Show option is selected. Hiliting induced by a given query on the main display window can be turned off by pressing the “Clear” button while the query is selected.

The name given to a query in the manager window can be referred to in a subsequent query by placing a $-sign before the name in the new query. One should be cautioned that the query manager saves a data structure representing the result of a query when it is added, and doesn not re-execute a query when the “Query” button is clicked. Thus query names may be changed without affecting the correctness of any previously saved query.

Pressing the “Save” button, causes celamy to save the current query manager contents to the file

> ".celquery" at the users home directory. Whenever celamy is invoked it automatically loads the contents of the query manager from this file, if it can find it. The one pitfall in the current implementation is that celamy re-executes the queries at the time of this load in order to recreate the results of the queries. Thus if one of these queries A refers to a query B whose name was changed after A was saved to the manager, then an error will result. The correction is for celamy to read and write the results of the query and this may be corrected in future version.

AUTHORS

Gene Myers:

```
Created April 22, '99

Updated May 7, '99  (added description of comments/labels)

Updated May 16, '99  (replaced int ID's with names and enhanced pick)

Updated May 27, '99  (added first level of query capability)

Updated June 20, '99 (added view & canvas set & segment search)

Updated August 3, '99 (added patch to query mechanism, links and @)

Updated January 1, '00 (expanded @ to object queries and added to
                        operator repetoire)

Updated January 16, '00 (Added query manager)

Updated January 23, '00 (query names are operational and query
                         contents can be saved between sessions)
```