A Description of the Implementation of Collaborative ReviewPlus

We describe the implementation issues of the Collaborative ReviewPlus project, mainly focus on event and logic with it. We shall see that the Master and Participant objects share a common Deterministic Finite Automaton (DFA) in a session, and so have the same logic with regard to the state transitions, and converge on the same sate on each event. We describe the issues as follows.

1. Unit and Unity

We have developed the Collaborative ReviewPlus applications – the Master and Participant collaboration objects – from the original ReviewPlus application, without changing the overall logic related to state transitions. So they have the same logic with regard to the state transitions on events.

The logic corresponds to the transition function δ of the DFA, or the extended transition function Δ. The logic is composed of many IDL routines – procedures and functions with unique names. We can think of the routines as the building blocks or units of the logic, and the logic as the unity of the routines. So, routine is the unit, and δ or Δ is the unity of the units.

On any event, only one or some routines are executing to do the transition; in other words, only one part or some parts of the unity are actually functioning. But we can indistinguishably say that δ or Δ is reacting on the event and transiting to the next state.

2. Divergence and Convergence

The Master and Participant collaboration objects are designed for different purposes, in different architectures and implementing mechanisms; they are divergent. At the same time, they have the same logic as to the state transitions on events, and get to the same state at the end of the process of each event; they are convergent.

Let us describe in more detail about the implementation.

On the Master client, each widget that fires event is associated with an event handler – either a procedure or a function – in the widget construction program, which is registered at the end of the construction with the IDL system routine “xmanager.pro”, which in turn is managing the life-cycle of the widget and listening for events for it. Whenever the widget is triggered by the user through the interface, the system automatically gathers the information for the event and fits in the event structure, and invokes the event handler with the structure as the only parameter.

We add the code for collaboration here at the beginning of each event handler to capture the event and get the information of it for every field of the event structure, convert them into flat strings, and serialize them into a delimited single string, along with names of the event structure and the event handler, and send this result string to NB message broker for broadcasting to participant clients.

NB broadcasts the string to the participant and saves it in a public variable which is one element of a synchronized linked list added in one of NB's interface class, and also updates its event flag variable which reflects the number of strings saved.

The Participant client is developed using a Polling Structure. It is a main loop that is constantly polling the public variables – testing the event flag to see if it is non-zero, if it is, then removing a string from the head of the linked list to do further process.

In the process, it parses the string on the delimiter to get all the field pieces, the event structure name (or widget name) and the event handler name, converts the field pieces to native type values of the event structure, constructs the event structure using these values according to the event structure name, and finally renders the display by calling the event handler routine with the event structure as parameter, according to the event handler name.

As to the interactive input value in an input field such as text field, on the Master side, the user input them physically; on the Participant side, after it gets the value, it sets it in the field programmatically.

From the description above, the objects of the Master and Participant clients diverge in the shape of codes, architectures, implementing mechanisms and purposes. They are in diversity under the goal of collaboration.

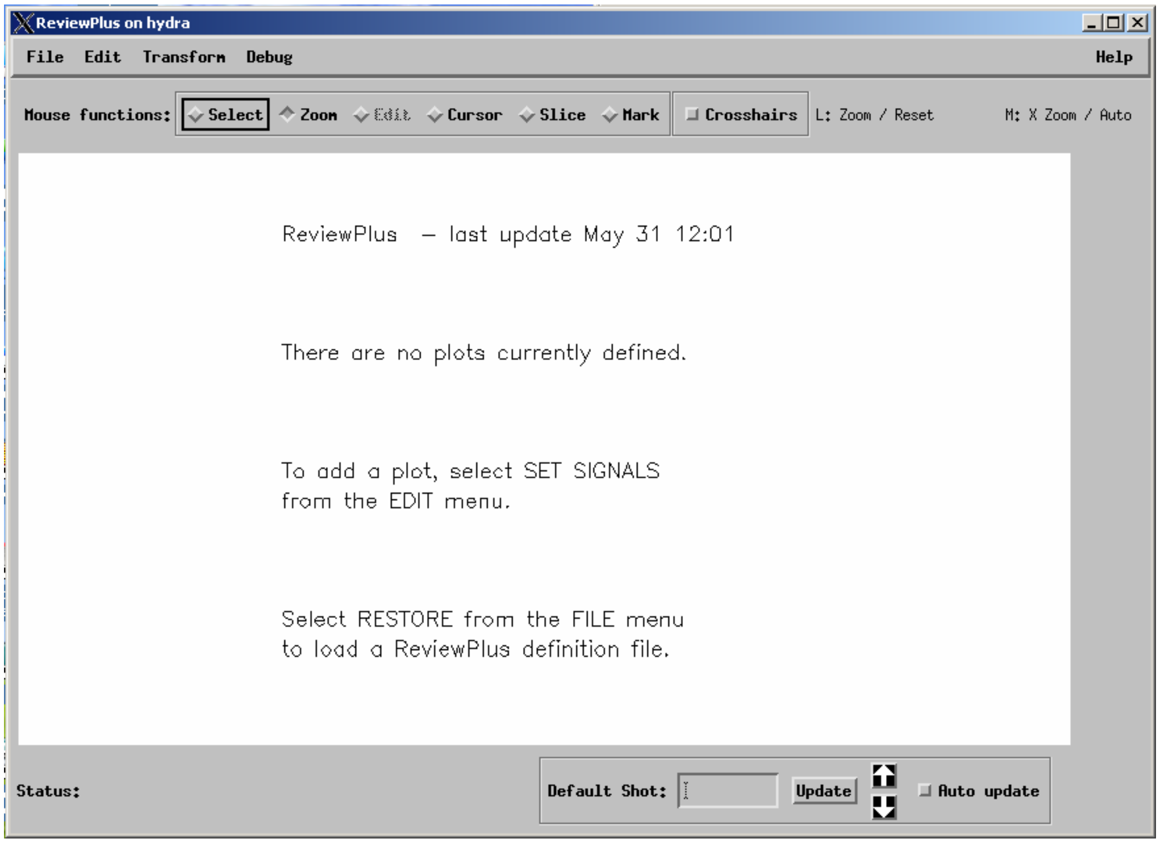
However, on each event, we have made them have the same input value in the input field if there is one, call the same routine of event handler with the event structure as parameter, and hence, at the end of the processes of the event, have the same output display; in other words, they converge on the same state of the DFA on each event, from the start state to the final accepting state, which is a well-defined session.

3. Collaboration on Event and Transition Function

We describe the collaboration between the objects of Master and Participant clients using pieces of code from the project, mainly focusing on event and the transition function. Instead of using exhausting enumeration of each and every widget cases in the interfaces, we give out the typical and interesting ones that sufficiently illustrate the idea of collaboration in terms of convergence on the same state of the DFA at the end of the process of each event, with the transition function doing the real job of state transitions.

Since the output display of both the Master and Participant object at each event are the same, we just show a single image capture in the demonstration at each step.

We begin with the invocation of the collaboration objects, as shown in Fig. 1. This corresponds to the start state q_0 of the DFA.



ReviewPlus — last update May 31 12:01

There are no plots currently defined.

To add a plot, select SET SIGNALS
from the EDIT menu.

Select RESTORE from the FILE menu
to load a ReviewPlus definition file.

Figure 1

From this interface of the Master object, if we click on the "Edit" item from the main menu, a sub-menu will appear, as shown in Fig. 2.

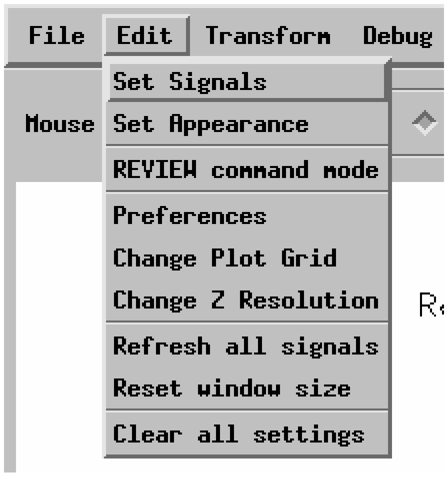


Figure 2

If we then click on the "Set Signals" item from the sub-menu, an event is fired. This is a button widget, and an event handler routine is defined for the event. We describe the pieces of code for both the Master and Participant objects in response of this cooperation as follows.

The Master Object Side

- Widget creation

```
  x = widget_button(mEdit, value='Set Signals', $
                    event_pro='ReviewPlus_SignalDialog_event')
```

From the code above we know that this button widget has `'Set Signals'` as its value shown on its appearance, and is associated with an event procedure named `'ReviewPlus_SignalDialog_event'`. When the button is clicked, the procedure is called by the IDL system.

- Definition of event structure for widget

Here is the definition of the event structure for widget button:

{WIDGET_BUTTON, ID:0L, TOP:0L, HANDLER:0L, SELECT:0L}

It has a name WIDGET_BUTTON and 4 fields – ID:0L, TOP:0L, HANDLER:0L, and SELECT:0L, each with a field name, a colon, and a type value. In this case all the values of the fields are of `long` type indicated by the suffix letter `L`.

SELECT: If the button is pressed, the value is 1; if it is released, the value is 0.

- Event handler

```
pro ReviewPlus_signaldialog_event,event
;;;;;;;; collaboration code added ;;;;;;;;
eventMessage = "ReviewPlus_signaldialog_event;"+"WIDGET_BUTTON;"+"ID;"$
  +string(event.ID)+";TOP;"+string(event.TOP)+";HANDLER;"$
  +string(event.HANDLER)+";SELECT;"+string(event.SELECT)

  COMMON BROKER, joChat2
  joChat2 -> writeMessage, eventMessage
;;;;;;;; end of collaboration code ;;;;;;;;

  widget_control,event.top,get_uvalue=info
  info.oReview->SignalDialog
end
```

From the code above we can see that the collaboration code captures the event and gets its field information from `event.ID`, `event.TOP`, `event.HANDLER`, etc., converts them into strings and serializes the strings into a semicolon delimited string, along with the event structure name `"WIDGET_BUTTON"` and the event handler name `"ReviewPlus_signaldialog_event"`. This result string is the event message, and is sent to the NB broker for broadcasting to Participants.

The Participant Object Side

- Parsing of event message

```
           result = STRSPLIT(uval, ';', COUNT=count, /EXTRACT,
/PRESERVE_NULL)

           which_event = result[0]
           which_widget = result[1]
```

The next event message string for the Participant Object to process is saved in variable `uval`. The IDL system function `STRSPLIT` is called to parse it with `';'` as the delimiter. All the pieces of information around the delimiter are extracted and saved in the array `result` with null string preserved as a piece, and the total number of them is saved in variable `count`. The event handler name is in `result[0]` or `which_event`, and the event structure name (or widget name) is in `result[1]` or `which_widget`. The rest of the pieces are all for the fields of the event structure and are saved in the rest elements of the array starting with `result[2]`.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_name = 'ID'
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_name = 'TOP'
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_name = 'HANDLER'
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'SELECT') THEN BEGIN
      select_name = 'SELECT'
      select_value = long(result[i+1])
                          :

   ENDIF
ENDFOR
```

The code above converts the information (in string) of the fields of the button event structure to its IDL native types; each pair of the strings, i.e. those stored in `result[i]`and `result[i+1]`, decide the field's value and the type of the value, with the former indicating the name and type of the value (due to the unique association of a name with a type, the name alone can also indicate a type, e.g. `ID` is a `long` type), and the latter the value in string.

In this case, all the values of the fields are of `long` type, and therefore the strings are converted to IDL type `long`.

- Construction of event structure

```
        IF (which_widget EQ 'WIDGET_BUTTON') THEN $
           event_structure = {WIDGET_BUTTON,id:id_value,$
           top:top_value,handler:handler_value,select:select_value}$
        ELSE IF ...
```

The code above constructs the widget button event structure using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routine of event handler

```
...
ELSE IF (which_event EQ 'ReviewPlus_signaldialog_event') THEN BEGIN
           ReviewPlus_signaldialog_event, event_structure
     ENDIF ELSE IF ...
```

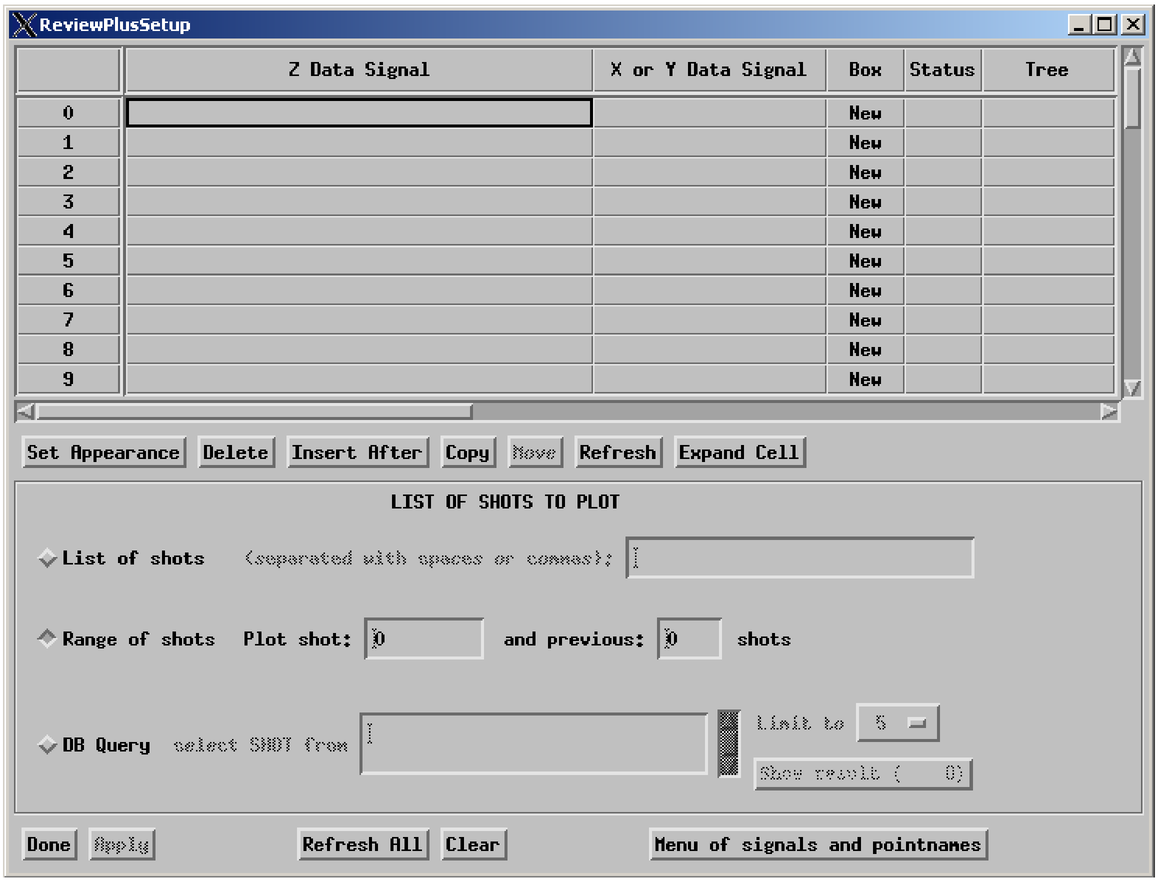
The code above calls the routine of the event handler `ReviewPlus_signaldialog_event` with the constructed event structure `event_structure` as the only parameter.

Step Summary

In the process on the event, both the Master and Participant objects call the same routine – the event handler `ReviewPlus_signaldialog_event -` which is a unit of the transition function δ, with the event structure as the only parameter. The event message acts as the messenger, the information source, and the coordinator.

With $\delta(q_0, a_0) = q_1$, the Master and Participant objects converge on the same state q_1 of the DFA on event message a_0 at the end of the process of the event, and therefore they have the same output display, as in Fig. 3 and Fig. 4, which are two parts of a big interface.

Note that, inside the event handler, other routines can be called in any sequence and order, which we don't have to worry about and just think of it as the encapsulation and abstraction of the event handler.



	Z Data Signal	X or Y Data Signal	Box	Status	Tree
0			New		
1			New		
2			New		
3			New		
4			New		
5			New		
6			New		
7			New		
8			New		
9			New		

LIST OF SHOTS TO PLOT

List of shots (separated with spaces or commas):

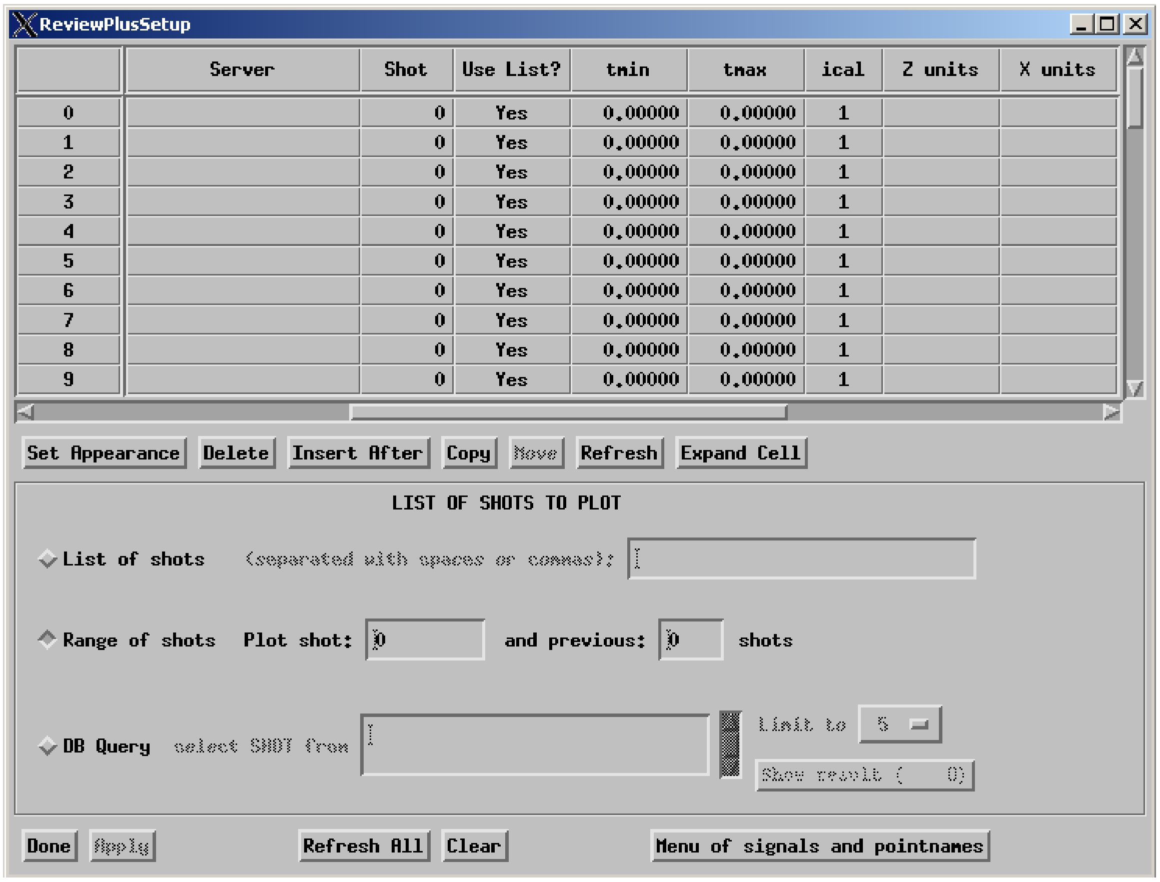
Range of shots Plot shot: 0 and previous: 0 shots

DB Query select SHOT from

Limit to 5

Show result (0)

Figure 3



	Server	Shot	Use List?	tmin	tmax	ical	Z units	X units
0		0	Yes	0.00000	0.00000	1		
1		0	Yes	0.00000	0.00000	1		
2		0	Yes	0.00000	0.00000	1		
3		0	Yes	0.00000	0.00000	1		
4		0	Yes	0.00000	0.00000	1		
5		0	Yes	0.00000	0.00000	1		
6		0	Yes	0.00000	0.00000	1		
7		0	Yes	0.00000	0.00000	1		
8		0	Yes	0.00000	0.00000	1		
9		0	Yes	0.00000	0.00000	1		

LIST OF SHOTS TO PLOT

◇ List of shots (separated with spaces or commas):

◇ Range of shots Plot shot: 0 and previous: 0 shots

◇ DB Query select SHOT from Limit to 5 Show result (0)

Figure 4

At the upper part of this interface is a widget table. Different types of events can happen to it, such as *Insert Single Character, Insert Multiple Characters, Delete Text, Select Text, Select Cell, Change Row Height, Change Column Width*, etc., each with a different definition of event structure. Some of the events are of time sequence priority, for example, in order to enter a single or multiple characters, we must first select the cell, and there must be some content in the cell before we can delete or select the text. However, the process for each event is similar. So not loosing generality, we can choose to just describe its *Insert Single Character* event.

Suppose we enter some data in row 0 of the widget table. Let us input “ip” in the cell under column “Z Data Signal” in Fig. 3, and “104276” in the cell under column “Shot” in Fig. 4. In this case, each character entered including the line feed and carriage return will cause an event, and therefore a state transition in the DFA. We just describe one of them in representation because the rest are the same in process.

The Master Object Side

- Widget creation

```
oColumns = objarr(17)
```

```
  oColumns[0] = obj_new("GATableColumnEdit", label="Z Data Signal",
width=300)
...

  oColumns[6] = obj_new('GATableColumnEdit', label="Shot", width=60,
alignment=2)
...

  oTable = obj_new("GATable", self.wTLB, oColumns, Y_SCROLL_SIZE=10,
X_SCROLL_SIZE=5, /RESIZEABLE_COLUMNS)
...

  info = self->_GetColumnInfo()
  self.wID = widget_table(self.wParent, $
                          column_labels=info.label, $
                          /edit, /all_events, $
                          ...
                          event_pro='GATable_event', uvalue=self)
```

From the code above we can see that this table widget is associated with an event procedure named `'GATable_event'`, and whenever an event is occurred, the procedure is called by the IDL system; that it is editable, indicated by the keyword `/edit`; that an event is fired whenever the content of the text field of a cell has changed, indicated by the keyword `/all_events`. This table consists of 17 columns; each is an object of a module, or class. The listed one that is of interest to us now is `GATableColumnEdit`. The table itself `oTable` is an aggregated object consisting of the column objects; during its instantiation `obj_new("GATable", ...)`, it calls `widget_table(...)`to build up the widget.

- Definition of event structure for widget

Here is the definition of the *Insert Single Character* event structure for widget table:

{WIDGET_TABLE_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L, CH:0B, X:0L, Y:0L}

It has a name WIDGET_TABLE_CH and 8 fields. The field CH:0B is of `byte` type indicated by the suffix letter B, the field TYPE:0 is of `int` type indicated by a value only, and the others are of `long` type.

TYPE: The event type from the widget table event types.
CH: The ASCII value of the inserted character.
OFFSET: The zero-based character position after insertion.
X: The zero-based column address of the cell in the widget table.
Y: The zero-based row address of the cell in the widget table.

- Event handler

```
pro GATable_event, ev
;;;;;;;; collaboration code added ;;;;;;;;
   COMMON BROKER, joChat2
   case (ev.TYPE) of
      0 : BEGIN
            eventMessage =
              "GATable_event;"+"WIDGET_TABLE_CH;"+"ID;"+string(ev.ID)$
            +";TOP;"+string(ev.TOP)+";HANDLER;"+string(ev.HANDLER)$
            +";TYPE;"+string(ev.TYPE)+";OFFSET;"+string(ev.OFFSET)$
            +";CH;"+string(ev.CH)+";X;"+string(ev.X)+";Y;"+string(ev.Y)

            joChat2 -> writeMessage, eventMessage
          END
      ...
   endcase
;;;;;;;; end of collaboration code ;;;;;;;;

  widget_control,ev.id,get_uvalue=oSelf
  oSelf->_HandleEvent,ev
end
```

From the code above we can see that the collaboration code captures the event and gets its field information from `ev.ID`, `ev.TOP`, `ev.HANDLER`, etc., converts them into strings and serializes the strings into a semicolon delimited string, along with the event structure name `"WIDGET_TABLE_CH"` and the event handler name `"GATable_event"`. This result string is the event message, and is sent to the NB broker for broadcasting to Participants.

This event handler handles different types of widget table events such as *Insert Single Character, Insert Multiple Characters, Delete Text, Select Cell,* etc., depending on the event structure's field value `ev.TYPE` to decide the current case. We just list the one in question *Insert Single Character* above.

The Participant Object Side

- Parsing of event message

Same as previously described.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TYPE') THEN BEGIN
      type_value = fix(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'OFFSET') THEN BEGIN
      offset_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'CH') THEN BEGIN
      ch_value_array = byte(result[i+1])
      ch_value = ch_value_array[0]
```

```
              ENDIF ELSE IF (result[i] EQ 'X') THEN BEGIN
                 x_value = long(result[i+1])
              ENDIF ELSE IF (result[i] EQ 'Y') THEN BEGIN
                 y_value = long(result[i+1])
                                     :

              ENDIF
           ENDFOR
```

The code above converts the information (in string) of the fields of the *Insert Single Character* event structure of widget table to its IDL native types; each pair of the strings, i.e. those stored in `result[i]`and `result[i+1]`, decide the field's value and the type of the value, with the former indicating the name and type of the value (due to the unique association of a name with a type, the name alone can also indicate a type, e.g. `ID` is a `long` type), and the latter the value in string.

In this case, the field `TYPE` is of `int` type and the string is converted to IDL type `fix`, the field `CH` is of `byte` type and the string is converted to IDL type `byte` and then the first element of the byte array is pulled out for the character, and the other fields are of `long` type, and therefore the strings are converted to IDL type `long`.

- Construction of event structure

```
           IF (which_widget EQ 'WIDGET_TABLE_CH') THEN $
               event_structure = {WIDGET_TABLE_CH,id:id_value,$
                                  top:top_value,handler:handler_value,$
                                  type:type_value,offset:offset_value,$
                                  ch:ch_value,x:x_value,y:y_value}
```

The code above constructs the *Insert Single Character* event structure of widget table using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routine of event handler

```
           ...
           ELSE IF (which_event EQ 'GATable_event') THEN BEGIN
              GATable_event, event_structure
           ENDIF ELSE IF ...
```

The code above calls the routine of the event handler `GATable_event` with the constructed event structure `event_structure` as the only parameter.

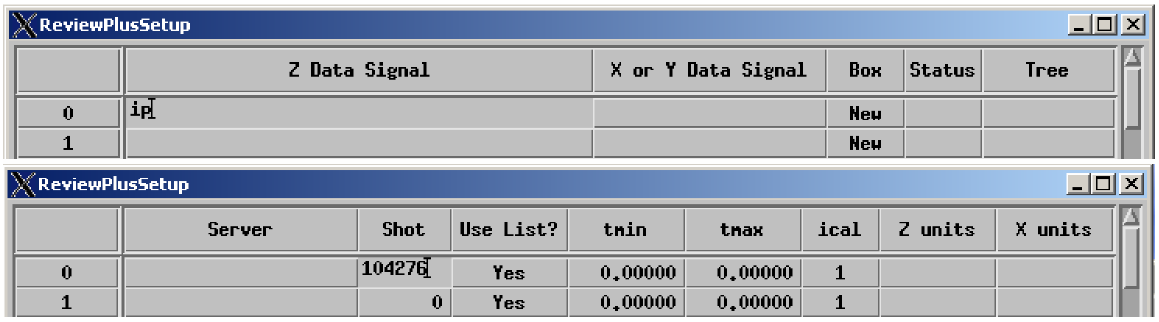
The event handler `GATable_event` calls the following procedure indirectly to insert the new character into the content of the current cell of the widget table programmatically.

```
pro GATableColumnEdit::_HandleInsertCharacter, cell, ev
    ...
    self->_AddToBuffer, string(ev.ch), ev.offset
    cell = {column:ev.x, row:ev.y}
    s = self.oTable->GetBuffer()
    self.oTable->SetCellValue, cell, s
    ...
end
```

Step Summary

In the process on the event, both the Master and Participant objects call the same routine – the event handler `GATable_event` – which is a unit of the transition function δ, with the event structure as the only parameter. The event message acts as the messenger, the information source, and the coordinator.

With $\delta(q_i, a_i) = r$, the Master and Participant objects converge on the same state r of the DFA on event message a_i at the end of the process of the event, and therefore they have the same output display – in this case, whenever a character is entered on the Master side, it is immediately reflected on the Participant side, so that the entire detailed entering process is showing on both sides. We show the final display about this in Fig. 5.



ReviewPlusSetup

	Z Data Signal	X or Y Data Signal	Box	Status	Tree
0	ip		New		
1			New		

ReviewPlusSetup

	Server	Shot	Use List?	tmin	tmax	ical	Z units	X units
0		104276	Yes	0.00000	0.00000	1		
1		0	Yes	0.00000	0.00000	1		

Figure 5

At this point, if we click on the widget button "Done" at the lower part of the interface, as shown in Fig 4, we get the result as shown in Fig. 6. We can omit the description of this event process because we have done button event previously, except in this case the routine of event handler `ReviewPlusSetup_done_event` is called on both sides. So, they converge to the same state r of DFA.

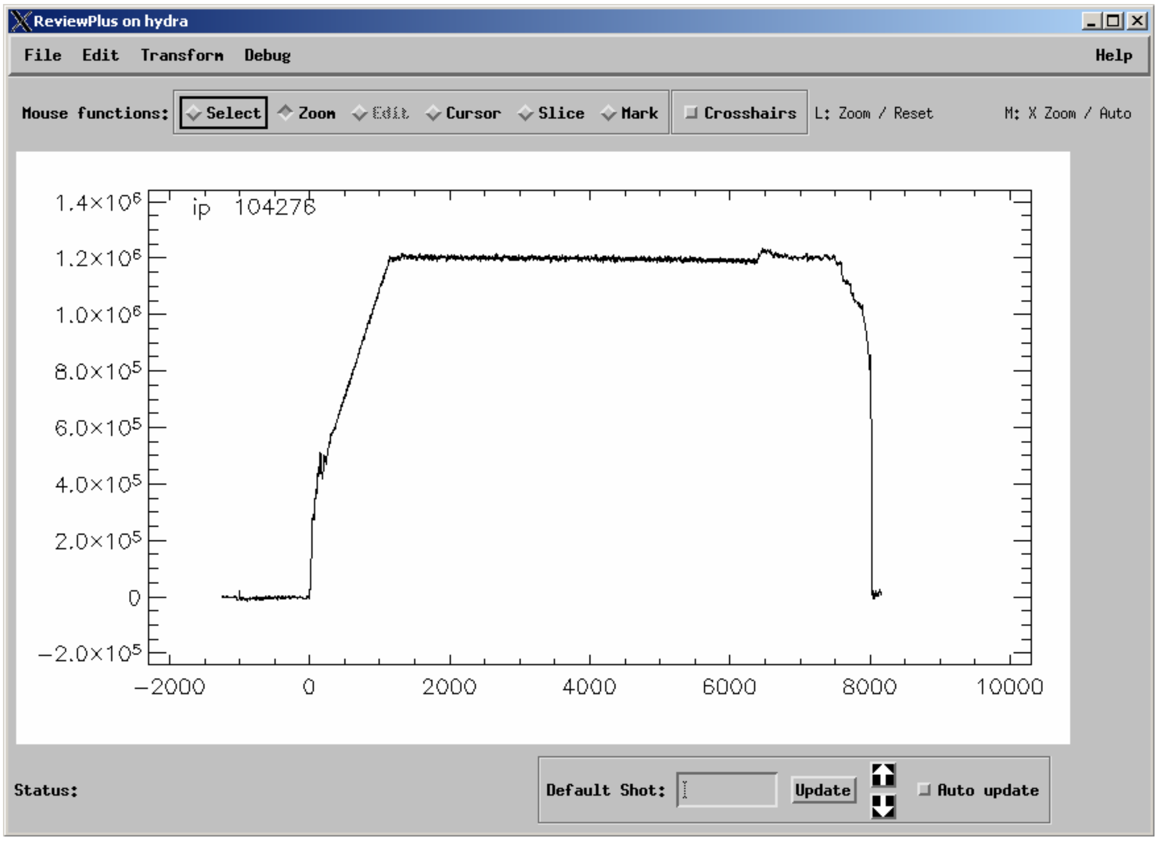
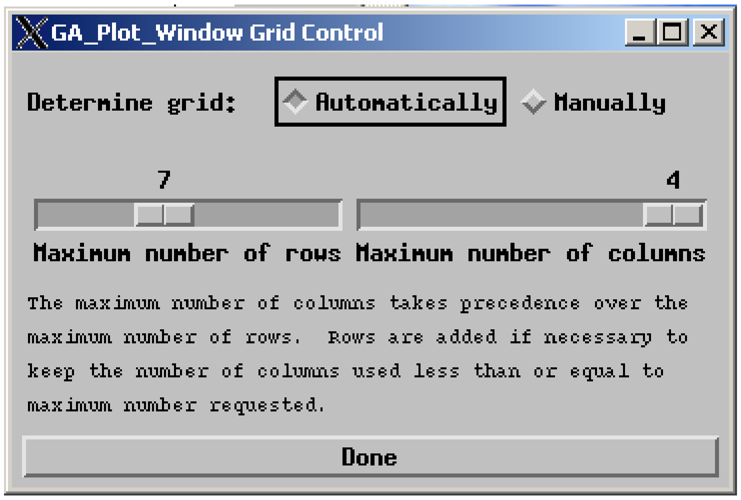


Figure 6

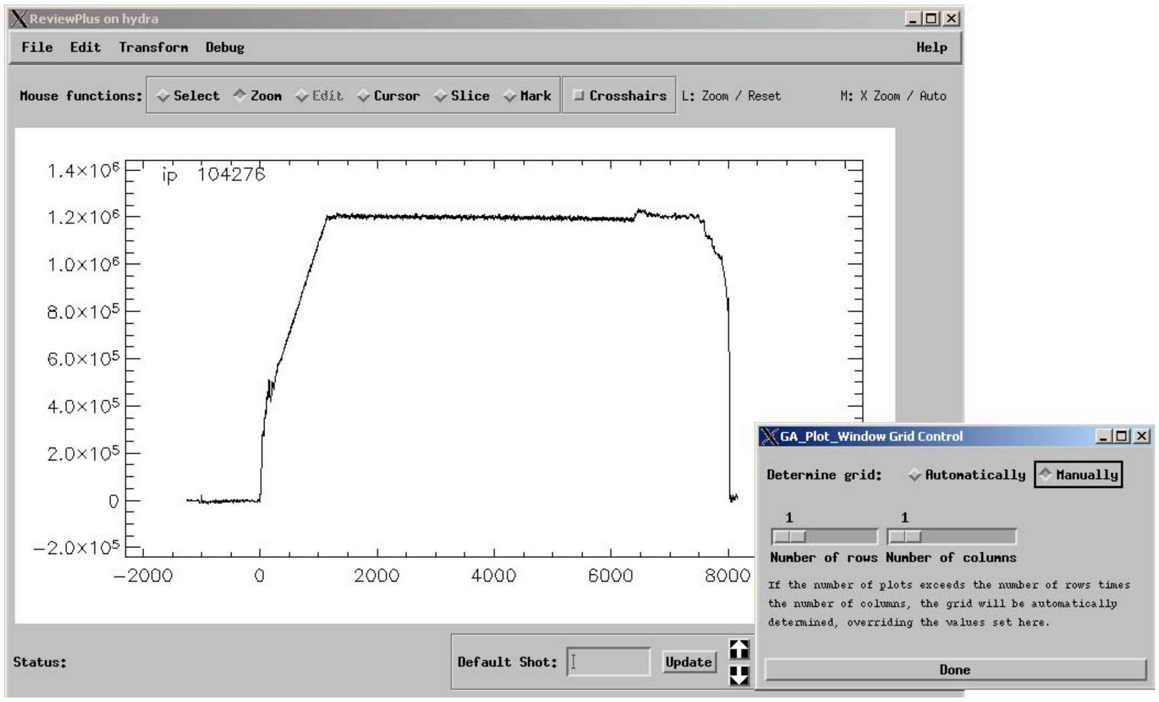
If we click on the “Change Plot Grid” item from the sub-menu of Fig. 2, an event is fired. This is a button widget, and the Master and Participant objects will converge to a same state r of DFA and have the same display, as follows in Fig. 7.



The maximum number of columns takes precedence over the maximum number of rows. Rows are added if necessary to keep the number of columns used less than or equal to maximum number requested.

Figure 7

Most of the widgets in this interface are widget sliders. We shall describe one of them, and the rest will be clear. Let us click on the exclusive button "Manually" in Fig. 7, and we get the display as follows in Fig. 8.



Determine grid: Automatically [✓ Manually]

1 Number of rows
1 Number of columns

If the number of plots exceeds the number of rows times the number of columns, the grid will be automatically determined, overriding the values set here.

Figure 8

We shall drag the widget slider "Number of rows", change its value from 1 to 2, and see how this action will also affect the graphics output in the main interface.

The Master Object Side

- Widget creation

```
...
baseMan = widget_base(bboard, /column, map=0,$
                event_pro='ga_plot_window_griddialog_man_event')

r = widget_base(baseMan, /row)
slrow = widget_slider(r, title="Number of rows", min=1, max=16)
...
```

From the code above we can see that this widget slider is associated with an event procedure named `'ga_plot_window_griddialog_man_event'` through its ancestor widget in the widget hierarchy. The two base widgets determine the layout of the grid for component widgets. Whenever an event is occurred to the widget slider, it bubbles up through the hierarchy to the widget base with the handler, and the event procedure is called with the event by the IDL system. The widget slider has the title `"Number of rows"` and the range of values with `min=1, max=16`.

- Definition of event structure for widget

Here is the definition of the event structure for widget slider:

{WIDGET_SLIDER, ID:0L, TOP:0L, HANDLER:0L, VALUE:0L, DRAG:0}

It has a name WIDGET_SLIDER and 5 fields. The field DRAG:0 is of `int` type indicated by a value only, and the others are of `long` type.

VALUE: The new value of the widget slider.
DRAG: The value that indicates whether the event is generated during or at the end of a slider drag, with the value 1 or 0.

- Event handler

```
pro ga_plot_window_griddialog_man_event,ev
;;;;;;;; collaboration code added ;;;;;;;;
   tag = tag_names(ev,/str)
   if (tag eq 'WIDGET_SLIDER') then begin
      eventMessage = "ga_plot_window_griddialog_man_event;"$
                    +"WIDGET_SLIDER;"+"ID;"+string(ev.ID)$
                    +";TOP;"+string(ev.TOP)+";HANDLER;"$
                    +string(ev.HANDLER)+";VALUE3;"+string(ev.VALUE)$
                    +";DRAG;"+string(ev.DRAG)
      COMMON BROKER, joChat2
      joChat2 -> writeMessage, eventMessage
   endif
;;;;;;;; end of collaboration code ;;;;;;;;

  widget_control,ev.top,get_uvalue=u
  widget_control,u.slrow,get_value=row
  widget_control,u.slcol,get_value=col
  u.self->UserSetGrid,[col,row],/draw
end
```

From the code above we can see that the collaboration code captures the event and gets its field information from `ev.ID`, `ev.TOP`, `ev.HANDLER`, etc., converts them into strings and serializes the strings into a semicolon delimited string, along with the event structure name `"WIDGET_SLIDER"` and the event handler name `"ga_plot_window_griddialog_man_event"`. This result string is the event message, and is sent to the NB broker for broadcasting to Participants.

The Participant Object Side

- Parsing of event message

Same as previously described.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'VALUE3') THEN BEGIN
      value3_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'DRAG') THEN BEGIN
      drag_value = fix(result[i+1])
                        :

   ENDIF
ENDFOR
```

The code above converts the information (in string) of the fields of the event structure of widget slider to its IDL native types; each pair of the strings, i.e. those stored in `result[i]`and `result[i+1]`, decide the field's value and the type of the value, with the former indicating the name and type of the value (due to the unique association of a name with a type, the name alone can also indicate a type, e.g. `ID` is a `long` type), and the latter the value in string.

In this case, the field `DRAG` is of `int` type and the string is converted to IDL type `fix`, and the other fields are of `long` type, and therefore the strings are converted to IDL type `long`. The field `VALUE` is of `long` type in this event structure, but in some other structures, the field `VALUE` is used for `int` or `float`. To resolve this conflict in programming, we use `VALUE3` to exclusively indicate that it is of `long` type.

- Construction of event structure

```
...
ELSE IF (which_widget EQ 'WIDGET_SLIDER') THEN $
   event_structure = {WIDGET_SLIDER,id:id_value,$
                      top:top_value,handler:handler_value,$
                      value:value3_value,drag:drag_value}$
ELSE IF ...
```

The code above constructs the event structure of widget slider using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routine of event handler

```
...
ELSE IF (which_event EQ 'ga_plot_window_griddialog_man_event')$
     THEN BEGIN
          ReviewPlus_widget_slider_event, event_structure
          ga_plot_window_griddialog_man_event, event_structure
ENDIF ELSE IF ...
```

The code above first calls the routine of `ReviewPlus_widget_slider_event` with the constructed event structure `event_structure` to set up the value of the widget slider programmatically as that of the Master, and it then calls the routine of the event handler `ga_plot_window_griddialog_man_event` with the event structure to have the function as that of the Master.

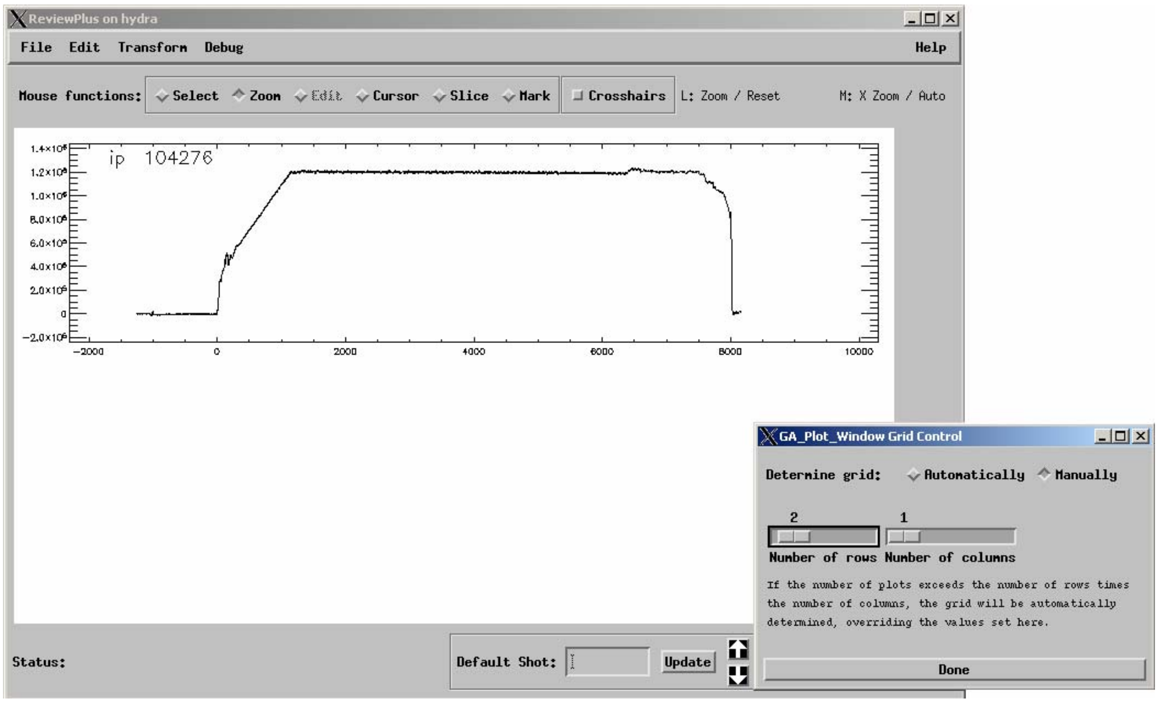
The routine of `ReviewPlus_widget_slider_event` is listed below.

```
pro ReviewPlus_widget_slider_event,event
  widget_control,event.id,set_value=event.value
end
```

Step Summary

In the process on the event, both the Master and Participant objects call the same routine – the event handler `ga_plot_window_griddialog_man_event –` which is a unit of the transition function δ, with the event structure as the only parameter. On the Participant, an extra routine `ReviewPlus_widget_slider_event` is called to set up the value of the slider in order to simulate the slider drag on the Master, so that the sliders on both the Master and Participant have the same value and appearance. We can think of this as another unit of the transition function δ, and hence in this process of the event, two units are called to get to the same state.

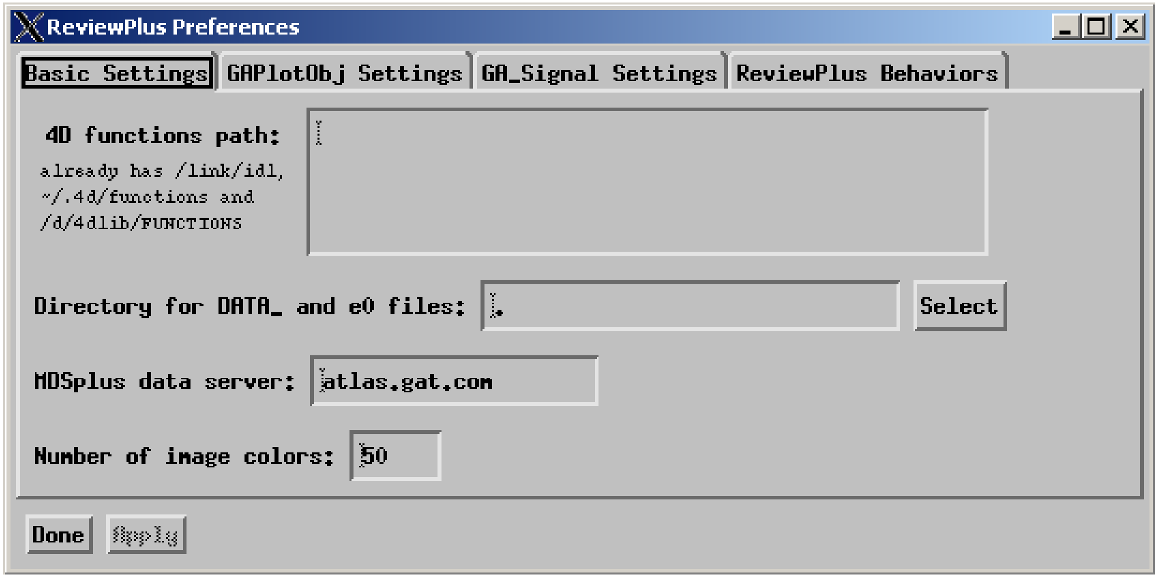
With $\delta(q_i, a_i) = r$, the Master and Participant objects converge on the same state r of the DFA on event message a_i at the end of the process of the event, and therefore they have the same output display – in this case, the same appearance of the widget slider and the look of the output in the main interface, as shown in Fig. 9.


ReviewPlus on hydra
File Edit Transform Debug Help
Mouse functions: Select Zoom Edit Cursor Slice Mark Crosshairs L: Zoom / Reset M: X Zoom / Auto
ip 104276
Status:
Default Shot: Update
GA_Plot_Window Grid Control
Determine grid: Automatically Manually
2 1
Number of rows Number of columns
Done

If the number of plots exceeds the number of rows times the number of columns, the grid will be automatically determined, overriding the values set here.

Figure 9

If we click on the “Preferences” item from the sub-menu of Fig. 2, an event is fired. This is a button widget, and the Master and Participant objects will converge to a same state r of DFA and have the same display, as follows in Fig. 10.



4D functions path:

already has /link/idl, ~/.4d/functions and /d/4dlib/FUNCTIONS

Directory for DATA_ and e0 files: .

MDSplus data server: atlas.gat.com

Number of image colors: 50

Figure 10

This interface mainly contains a new type of widget – tab widget. Tab widget usually contains multiple different pages, each with a tab for labeling. Only one page can be displayed at a time in the widget's display area by selecting the appropriate tab. The titles of the tabs above are "*Basic Settings*", "*GAPlotObj Settings*", "*GA_Signal Settings*" and "*ReviewPlus Behaviors*".

Let us select the tab with the title "*GAPlotObj Settings*" and describe the process of this event on the collaboration objects.

This action actually triggers two events – one related to the widget tab, and the other related to a widget base in the upper level of the widget hierarchy.

A function event handler is associated with the tab widget. A function event handler not only processes the event happened to its associated widget, but also returns the event, possibly with modifications, to the parent widget in the higher level of the widget hierarchy. The event keeps bubbling up through the hierarchy until it is consumed or swallowed by an associated event handler, or if not, by the IDL system at last.

The function event handler in charge of the widget tab processes the event, and then it returns it as new event to the event handler associated with the widget base for further process to change the result of the output.

The Master Object Side

- Widget creation

```
...
wBase = widget_base(title="ReviewPlus Preferences", /column,$
               /floating, group_leader=self.wTLB, tlb_frame_attr=3)

wTabs = WIDGET_TAB(wBase, event_func='ReviewPlus_widget_tab_event')

wBasic = WIDGET_BASE(wTabs, TITLE='Basic Settings', /COLUMN)
self.oWindow->PreferencesDialog,wTabs,TITLE='GAPlotObj Settings'
wSigs = WIDGET_BASE(wTabs, TITLE='GA_Signal Settings', /COLUMN, $
                                                      /ALIGN_LEFT)
wBehaviors = WIDGET_BASE(wTabs, TITLE='ReviewPlus Behaviors',$
                                                        /COLUMN)
...
```

From the code above we can see that this tab widget is associated with an event function named `'ReviewPlus_widget_tab_event'`, and has 4 tabs with the title names "*Basic Settings*", "*GAPlotObj Settings*", "*GA_Signal Settings*" and "*ReviewPlus Behaviors*", respectively; that the tabs are its children, and it is the child of the base widget `wBase` in the widget hierarchy.

- Definition of event structure for widget

Here is the definition of the event structure for widget tab:

{WIDGET_TAB, ID:0L, TOP:0L, HANDLER:0L, TAB:0L}

It has a name WIDGET_TAB and 4 fields. All the fields are of `long` type.

TAB: The zero-based index of the selected tab in the tab widget.

- Event handlers

```
function ReviewPlus_widget_tab_event,event
   tag = tag_names(event,/str)
   if (tag eq 'WIDGET_TAB') then begin
      eventMessage = "ReviewPlus_widget_tab_event;"+"WIDGET_TAB;"$
                    +"ID;"+string(event.ID)+";TOP;"+string(event.TOP)$
                    +";HANDLER;"+string(event.HANDLER)$
                    +";TAB;"+string(event.TAB)
      COMMON BROKER, joChat2
      joChat2 -> writeMessage, eventMessage
   endif

   return, event
end
```

From the code above we can see that the collaboration code captures the event and gets its field information from `event.ID, event.TOP, event.HANDLER,` etc., converts them into strings and serializes the strings into a semicolon delimited string, along with the event structure name `"WIDGET_TAB"` and the event handler name `"ReviewPlus_widget_tab_event"`. This result string is the event message, and is sent to the NB broker for broadcasting to Participants.

The main purpose of this event function is to capture and get the event message when the Master clicks on the tab "*GAPlotObj Settings*". Later on the event message acts as the messenger and the information source, and coordinates the Participant to change to the same tab programmatically.

The second purpose of this event function is to return the event, without modification, to its parent widget, which is a base widget with the widget ID `wBase` and associated with an event handler named `reviewplus_preferences_apply_event`, as listed below.

```
    xmanager, 'ReviewPlus_preferencesdialog', wBase, $
        event_handler='reviewplus_preferences_apply_event', /no_block
```

The event acts as new event to this event handler, and is further processed and consumed by it.

```
pro ReviewPlus_preferences_apply_event,event
;;;;;;;; collaboration code added ;;;;;;;;
  tag = tag_names(event,/str)
  if ...
     ...
  endif else if (tag eq 'WIDGET_TAB') then begin
    eventMessage = "ReviewPlus_preferences_apply_event;"+"WIDGET_TAB;"$
```

```
                +"ID;"+string(event.ID)+";TOP;"+string(event.TOP)$
                +";HANDLER;"+string(event.HANDLER)$
                +";TAB;"+string(event.TAB)
  endif else ...

  COMMON BROKER, joChat2
  joChat2 -> writeMessage, eventMessage
;;;;;;;; end of collaboration code ;;;;;;;;

...
;;; other statements and commands
...

end
```

From the code above we can see that the collaboration code captures the event and gets its field information from `event.ID`, `event.TOP`, `event.HANDLER`, etc., converts them into strings and serializes the strings into a semicolon delimited string, along with the event structure name `"WIDGET_TAB"` and the event handler name `"ReviewPlus_preferences_apply_event"`. This result string is the event message, and is sent to the NB broker for broadcasting to Participants.

The Participant Object Side

- Parsing of event message

Same as previously described.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TAB') THEN BEGIN
      tab_value = long(result[i+1])
                         :

   ENDIF
ENDFOR
```

The code above converts the information (in string) of the fields of the event structure of widget tab to its IDL native types; each pair of the strings, i.e. those stored in `result[i]`and `result[i+1]`, decide the field's value and the type of the value, with the former indicating the name and type of the value (due to the unique association of a name with a type, the name alone can also indicate a type, e.g. `ID` is a `long` type), and the latter the value in string.

In this case, all the fields are of `long` type, and therefore the strings are converted to IDL type `long`.

- Construction of event structure

```
          ...
          ELSE IF (which_widget EQ 'WIDGET_TAB') THEN $
             event_structure = {WIDGET_TAB,id:id_value,top:top_value,$
                                handler:handler_value,tab:tab_value}$
          ELSE IF ...
```

The code above constructs the event structure of widget tab using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routines of event handlers

```
   ...
   ELSE IF (which_event EQ 'ReviewPlus_widget_tab_event') THEN BEGIN
             ReviewPlus_widget_tab_event, event_structure
   ENDIF ELSE IF (which_event EQ 'ReviewPlus_preferences_apply_event')$
         THEN BEGIN
             ReviewPlus_preferences_apply_event, event_structure
   ENDIF ELSE IF ...
```

The Master captured and sent out the event message `'ReviewPlus_widget_tab_event…'` and `'ReviewPlus_preferences_apply_event…'` in that order. The Participant should receive them in the same order (there is other mechanism to guarantee this order).

Therefore, the code above (within a loop) first calls the routine of `ReviewPlus_widget_tab_event` with the constructed event structure `event_structure` to change to the tab of the tab widget programmatically as that of the Master, and it then calls the routine of the event handler `ReviewPlus_preferences_apply_event` with the event structure to have the function as that of the Master. This is to reflect to the routine any changes in the previous page with the tab we have switched from.

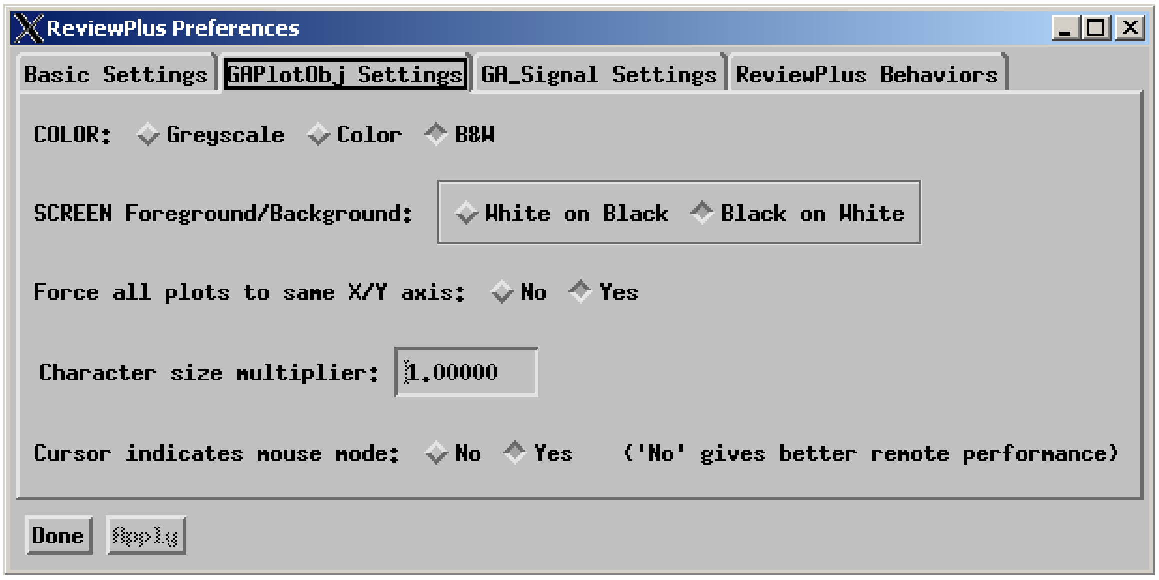
The routine of `ReviewPlus_widget_tab_event` is listed below.

```
Pro ReviewPlus_widget_tab_event,event
  widget_control,event.id,set_tab_current=event.tab,/show
end
```

Step Summary

In the processes on the events, both the Master and Participant objects call the same routines in the same order – the event handlers `ReviewPlus_widget_tab_event` and `ReviewPlus_preferences_apply_event`, which are two units of the transition function δ, with the event structure as the only parameter. As we have noticed, the event handler `ReviewPlus_widget_tab_event` is of different shape, purpose and function on the Master and Participant; on the Master, it is a function, and it is for capturing and sending out the event message and returning the event to upper widget, while on the Participant, it is a procedure, and it is for changing to the page of the desired tab. Nevertheless, it has the goal to coordinate both sides to have the same function and appearance at the end of the process of the event, in other words, to converge to the same state r of DFA.

With $\delta(q_i, a_i) = r$, the Master and Participant objects converge on the same states r of the DFA on event messages a_i at the end of the processes of the events, and therefore they have the same output display – in this case, the same appearance of the widget tab and the output in the main interface. We list the appearance of the widget tab in Fig. 11.


ReviewPlus Preferences

Basic Settings | GAPlotObj Settings | GA_Signal Settings | ReviewPlus Behaviors

COLOR: Greyscale Color [✓ B&W]

SCREEN Foreground/Background: White on Black [✓ Black on White]

Force all plots to same X/Y axis: No [✓ Yes]

Character size multiplier: 1.00000

Cursor indicates mouse mode: No [✓ Yes]   ('No' gives better remote performance)

Figure 11

Now, in this interface, if we click on the button "*White on Black*" from the exclusive button group beside the title "*SCREEN Foreground/Background*", it will be set and at the same time the other one "*Black on White*" will be unset, and the colors of the foreground and the background of the output will be exchanged.

As the step above, this action triggers several events in sequence, and it is related to event functions and the bubbling up of event through the widget hierarchy. We describe it below.

The Master Object Side

- Widget creation

```
pro ga_plot_window::PreferencesDialog,parent, title=title
   ...
```

```
tlb = widget_base(parent, /column, frame=1, title=title, $
                  pro_set_value='ga_plot_window_preferences_set',$
                  event_func='ga_plot_window_pref_applyfunc_event')
...
baseReverse = widget_base(tlb, /row,$
                event_func='ReviewPlus_cw_bgroup_index_func_event')

x = widget_label(baseReverse, value="SCREEN Foreground/Background: ")
butsRev = cw_bgroup(baseReverse,['White on Black', 'Black on White'],
                    /exclusive, /row, /frame, /no_release)
...
```

From the code above we can see that this compound widget CW_BGROUP is associated with an event function named `'ReviewPlus_cw_bgroup_index_func_event'` through its parent widget `widget_base` with ID `baseReverse`; that the group of buttons in its base are exclusive indicated by the keyword `/exclusive` and are laid horizontally indicated by `/row`. The parent of widget `baseReverse` is `widget_base` with ID `tlb`, and the parent of widget `tlb` is the widget with ID `parent`. If we list again some piece of the code we have described in the last step as follows,

```
...
wBase = widget_base(title="ReviewPlus Preferences", /column,$
              /floating, group_leader=self.wTLB, tlb_frame_attr=3)

wTabs = WIDGET_TAB(wBase, event_func='ReviewPlus_widget_tab_event')

...
self.oWindow->PreferencesDialog,wTabs,TITLE='GAPlotObj Settings'
...
```

We can see that ID `parent` and ID `wTabs` are equivalent algebraically in execution, and the parent of widget `wTabs` is the base widget with ID `wBase`.

For clarity, we list the related widgets in question along the path from the upper to lower levels of the widget hierarchy showing part of the relationships of the family, with each member having the format of *widget ID*, *widget type*, and *associated event handler*.

```
    wBase, widget_base, pro ReviewPlus_preferences_apply_event,event
                                       ↑
      wTabs, WIDGET_TAB, function ReviewPlus_widget_tab_event,event
                                       ↑
 tlb, widget_base, function ga_plot_window_pref_applyfunc_event,event
                                       ↑
                   baseReverse, widget_base, function
             ReviewPlus_cw_bgroup_index_func_event,event
                                       ↑
                             butsRev, cw_bgroup,
```

- Definition of event structure for widget

Here is the definition of the event structure for CW_BGROUP widget:

{ID:0L, TOP:0L, HANDLER:0L, SELECT:0L, VALUE:0}

It has 5 fields, but unlike most of the widgets, it doesn't have a structure name with it. The field VALUE is of `int` type, and the rest are of `long` type.

SELECT: The value passed through from the button event with 1 indicating the button is set and 0 otherwise.

VALUE: The value of the button. It can be "INDEX" (the index of the button in the base), "ID" (the widget ID of the button), "NAME" (the name of the button), or "BUTTON_UVALUE" (the user value for the button).

- Event handlers

Whenever an event happens to CW_BGROUP widget `butsRev`, e.g. a button is set or unset in the base, it automatically bubbles up to the parent of `butsRev`, which is `baseReverse`, because `butsRev` doesn't have its own event handler. The `baseReverse` has an event function as follows to deal with the event.

```
function ReviewPlus_cw_bgroup_index_func_event,event
   tag = tag_names(event,/str)
   if (tag ne 'WIDGET_BASE') then begin
      eventMessage = "ReviewPlus_cw_bgroup_index_event;"$
                   +"CW_BGROUP_INDEX;"+"ID;"+string(event.ID)$
                   +";TOP;"+string(event.TOP)+";HANDLER;"$
                   +string(event.HANDLER)$
                   +";SELECT;"+string(event.SELECT)$
                   +";VALUE_CW_BGROUP;"+string(event.VALUE)
      COMMON BROKER, joChat2
      joChat2 -> writeMessage, eventMessage
   endif

   return, event
end
```

From the code above we can see that the collaboration code captures the event and gets its field information from `event.ID, event.TOP, event.HANDLER,` etc., converts them into strings and serializes the strings into a semicolon delimited string, along with the event structure information string `"CW_BGROUP_INDEX"` (meaning it is a CW_BGROUP widget and created using `INDEX` to refer to a button in the base) and the event handler name `"ReviewPlus_cw_bgroup_index_event"` to be called on the Participant. This result string is the event message, and is sent to the NB broker for broadcasting to Participants.

The main purpose of this event function is to capture and get the event message when the Master clicks on a button from the button group in the base. Later on the event message acts as the messenger and the information source, and coordinates the Participant to set/unset the same button programmatically.

The second purpose of this event function is to return the event, without modification, to its parent widget, which is a base widget with the widget ID `tlb` and associated with an event handler named `ga_plot_window_pref_applyfunc_event`, as listed below.

```
function ga_plot_window_pref_applyfunc_event,event
  return,event
end
```

This function does nothing but returns the event to the widget in the upper level of the hierarchy. It contributes nothing to the state transitions of the DFA as long as the Participant object is concerned. So we would just leave it alone.

From here, the event continues to bubble up to the parent of `tlb`, which is `WIDGET_TAB` with the widget ID `wTabs` and associated with the function event handler named `ReviewPlus_widget_tab_event`. We have described previously the process and action happened here in the last step. We list again the part of interest to this step as follows.

```
function ReviewPlus_widget_tab_event,event
   tag = tag_names(event,/str)
   if (tag eq 'WIDGET_TAB') then begin
      eventMessage = ...
      ...
   endif

   return, event
end
```

Because CW_BGROUP widget is not a `WIDGET_TAB`, the mechanism of conditional test guarantees that no event message will be generated and sent out to NB, but the `event` will be returned untouched to the parent widget.

Once more, the event continues to bubble up to the parent of `wTabs`, which is `widget_base` with the widget ID `wBase` and associated with the procedure event handler named `ReviewPlus_preferences_apply_event`. The event gets processed and finally gets swallowed here. We list this procedure once again as follows only with the relevant part of it.

```
pro ReviewPlus_preferences_apply_event,event
;;;;;;;; collaboration code added ;;;;;;;;
    ...
    endif else begin
      eventMessage = "ReviewPlus_preferences_apply_event;"$
                   +"CW_BGROUP_INDEX;"+"ID;"+string(event.ID)$
                   +";TOP;"+string(event.TOP)+";HANDLER;"$
                   +string(event.HANDLER)$
                   +";SELECT;"+string(event.SELECT)$
                   +";VALUE_CW_BGROUP;"+string(event.VALUE)
    endelse
    ...

  COMMON BROKER, joChat2
  joChat2 -> writeMessage, eventMessage
;;;;;;;; end of collaboration code ;;;;;;;;

...
;;; other statements and commands
...

end
```

From the code above we can see that the collaboration code captures the event and gets its field information from `event.ID`, `event.TOP`, `event.HANDLER`, etc., converts them into strings and serializes the strings into a semicolon delimited string, along with the event structure information string `"CW_BGROUP_INDEX"` (meaning it is a CW_BGROUP widget and created using `INDEX` to refer to a button in the base) and the event handler name `"ReviewPlus_preferences_apply_event"` to be called on the Participant. This result string is the event message, and is sent to the NB broker for broadcasting to Participants.

The Participant Object Side

- Parsing of event message

Same as previously described.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'SELECT') THEN BEGIN
      select_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'VALUE_CW_BGROUP') THEN BEGIN
      cw_bgroup_value = result[i+1]
                          :

   ENDIF
ENDFOR
```

The code above converts the information (in string) of the fields of the event structure of CW_BGROUP widget to its IDL native types, except the `VALUE` field. Because it can be "INDEX" (the index of the button in the base), "ID" (the widget ID of the button), "NAME" (the name of the button), or "BUTTON_UVALUE" (the user value for the button), depending on how the widget is created, we need further information from the event message to decide its type. For example, if we get the event structure information string `"CW_BGROUP_INDEX"`, then we know it is an index and so its type is of `int`. So, for the moment, we just save it in a variable `cw_bgroup_value` and wait for further treatment in the construction of event structure.

- Construction of event structure

```
    IF (which_widget EQ 'CW_BGROUP_INDEX') THEN BEGIN
       bgroup_value = fix(cw_bgroup_value)
       event_structure = {CW_BGROUP_INDEX,id:id_value,$
                          top:top_value, handler:handler_value,$
                          select:select_value, value:bgroup_value}
    ENDIF
```

The code above constructs the event structure of CW_BGROUP widget with the `value` field of `INDEX` type, using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`. The type of the `value` field is finally decided with the information indicated in `'CW_BGROUP_INDEX'`, and the value is converted from string to integer via `fix()`.

- Invocation of the routines of event handlers

```
  ...
  IF (which_event EQ 'ReviewPlus_cw_bgroup_index_event') THEN BEGIN
        ReviewPlus_cw_bgroup_index_event, event_structure
        ...
  ...
  IF (which_event EQ 'ReviewPlus_preferences_apply_event') THEN BEGIN
        ReviewPlus_preferences_apply_event, event_structure
        ...
```

The Master captured and sent out the event message `'ReviewPlus_cw_bgroup_index_event…'` and `'ReviewPlus_preferences_apply_event…'` in that order. The Participant should receive them in the same order (there is other mechanism to guarantee this order).

Therefore, the code above (within a loop) first calls the routine of `ReviewPlus_cw_bgroup_index_event` with the constructed event structure `event_structure` to set/unset the button of the CW_BGROUP widget programmatically as that of the Master, and it then calls the routine of the event handler `ReviewPlus_preferences_apply_event` with the event structure to have the function as that of the Master. This is to exchange the colors of the foreground and the background of the output display within the main interface.

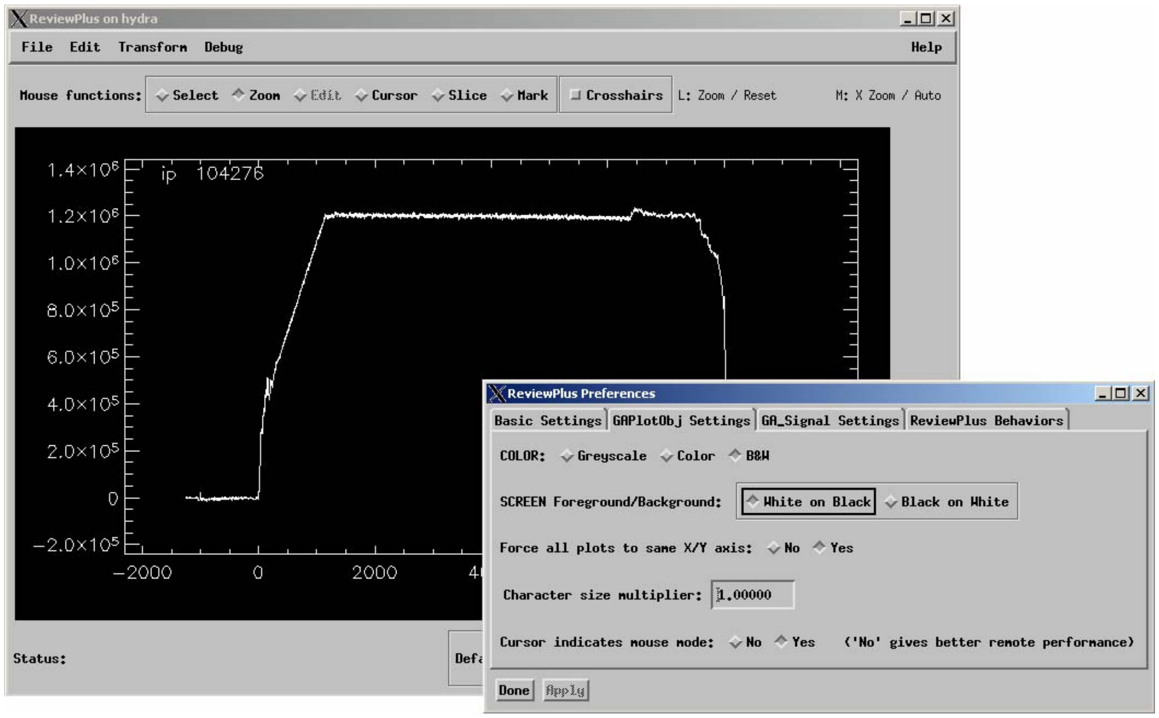
The routine of `ReviewPlus_cw_bgroup_index_event` is listed below.

```
pro ReviewPlus_cw_bgroup_index_event,event
  widget_control,event.id,set_value=event.value
end
```

Step Summary

In the processes on the events, both the Master and Participant objects have the same functionality of the routines by executing them in the same order – the event handlers `ReviewPlus_cw_bgroup_index_event` and `ReviewPlus_preferences_apply_event`, which are two units of the transition function δ, with the event structure as the only parameter. As we have noticed, the event handler `ReviewPlus_cw_bgroup_index_event` is of different shape, purpose and function on the Master and Participant; on the Master, it is a function, and it is for capturing and sending out the event message and returning the event to upper widget, while on the Participant, it is a procedure, and it is for setting/unsetting the button of the CW_BGROUP widget programmatically as that of the Master. Nevertheless, it has the goal to coordinate both sides to have the same function and appearance at the end of the process of the event, in other words, to converge to the same state r of DFA. The `ReviewPlus_preferences_apply_event` in this case exchanges the colors of the fore- and background of the output display.

With $\delta(q_i, a_i) = r$, the Master and Participant objects converge on the same states r of the DFA on event messages a_i at the end of the processes of the events, and therefore they have the same output display – in this case, the same appearance of the CW_BGROUP widget and the output in the main interface. We list the appearance of them in Fig. 12.



COLOR: Greyscale Color [✓ B&W]

SCREEN Foreground/Background: [✓ White on Black] Black on White

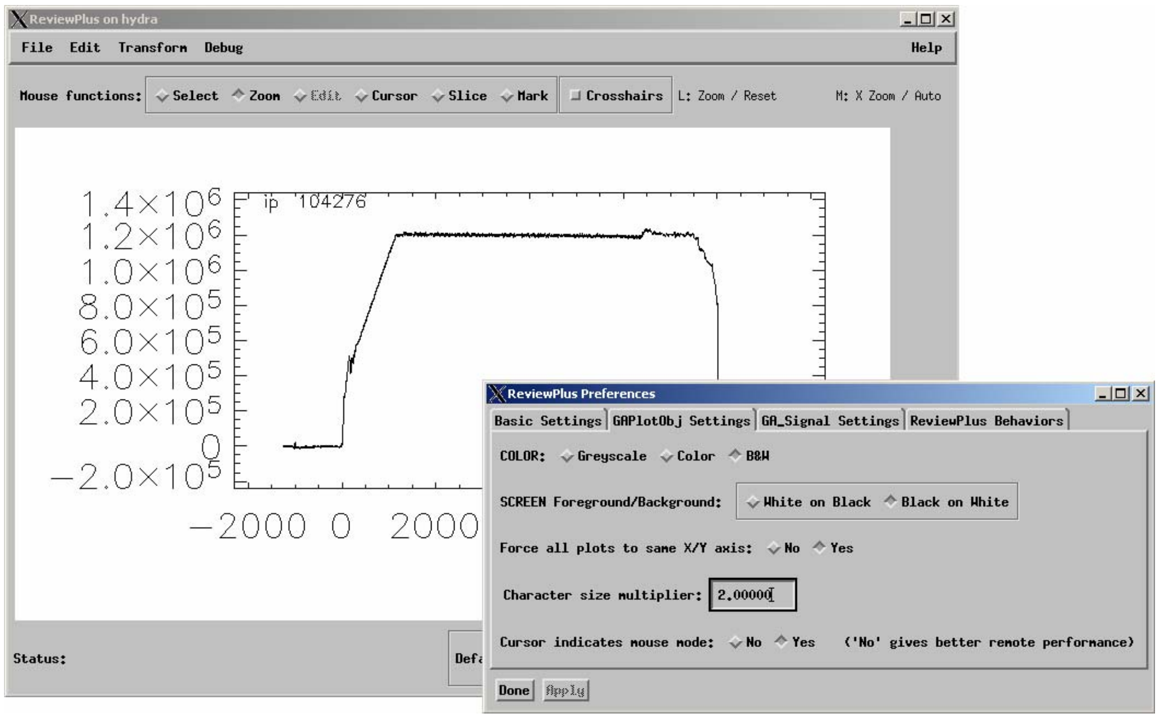
Force all plots to same X/Y axis: No [✓ Yes]

Character size multiplier: 1.00000

Cursor indicates mouse mode: No [✓ Yes] ('No' gives better remote performance)

Figure 12

Now, in the interface of Fig. 11, if we click on the CW_FIELD widget beside the title "*Character size multiplier*", input "2" and hit the carriage return, the output on both the Master and Participant objects will be displayed as in Fig. 13.


ReviewPlus on hydra
File Edit Transform Debug Help
Mouse functions: Select Zoom Edit Cursor Slice Mark Crosshairs L: Zoom / Reset M: X Zoom / Auto
ip 104276
Status:
ReviewPlus Preferences
Basic Settings GAPlotObj Settings GA_Signal Settings ReviewPlus Behaviors
Done Apply

COLOR: Greyscale Color B&W

SCREEN Foreground/Background: White on Black Black on White

Force all plots to same X/Y axis: No Yes

Character size multiplier: 2.00000

Cursor indicates mouse mode: No Yes ('No' gives better remote performance)

Figure 13

The process of this step is very similar as that of the last step, so we would not bother to describe it. The only difference is that, this is a CW_FIELD widget, and we would like to list its event structure and specialties below.

Here is the definition of the event structure for CW_FIELD widget:

{ID:0L, TOP:0L, HANDLER:0L, VALUE:'', TYPE:0, UPDATE:0}

It has 6 fields, but unlike most of the widgets, it doesn't have a structure name with it. The field VALUE is of `string` type, and the TYPE and UPDATE are of `int` type.

VALUE: The value of the field.
TYPE: The type of the data in the field, with 0=string, 1=floating point, 2=integer, and 3=long integer.
UPDATE: 0 if the field has not been updated, 1 if it has.

Widget CW_FIELD is not defined with keyword `event_pro` or `event_func` and therefore it has to be associated with event handler through its parent or ancestor widget, normally base widget.

As before, this step converges to the same states r of DFA and so has the same output as in Fig. 13.

There are a lot more interfaces and widgets in ReviewPlus, such as WIDGET_DRAW, WIDGET_DROPLIST, WIDGET_LIST, WIDGET_TEXT, CW_FORM, etc. The descriptions of their processes of events are similar to those we have done so far. So we would like to get off the stage and leave the imaginations to the reader.