Selected Point window. The code necessary for a full implementation of this example is included with `DsTool`'s source code in the directory `$DSTOOL/src/windows/trial`.

### 3.2.2 Designing the Window with GUIDE

Consistent naming conventions make the task of adding a new window easier. Give the new window a short, descriptive name. For purposes of exposition, the new window will be called `trial`. (In all the instructions which follow, you should replace the word `trial` with the name of the panel which you are creating.) Create a subdirectory `$DSTOOL_USER/trial` for the new code and resulting library.

We suggest the GUIDE program be used to construct the new interface window, not only because it simplifies the design process, but also because it actually produces most of the C language code needed to build the window. The parts which you must write or modify involve the handlers and any part of the window which must vary according to the dynamical system (eg., the number of fields may depend on the dimension of the phase space).

We will assume that a simple window consists entirely of a control panel with panel items. These panel items are composed of buttons, text fields, and settings. Here are the conventions to be followed while using GUIDE:

1. Name the GUIDE generated file `trial.G`.
2. Name the window `win` and the control panel `pan`.
3. Event handlers are NOT necessary for simple windows.
4. All text field items can use the same notify handler. Name it `trial_text_notify`.
5. All settings items can use the same notify handler. Name it `trial_settings_notify`.
6. Each button item should have its own notify handler. Name them using the form `trial_buttonname_notify`.
7. If there is to be a variable number of text fields or settings, then create only one of each type as a template.

Run `gxv` on the file `trial.G` by typing `gxv trial.G` in order to generate GUIDE's C code for the new window.

### 3.2.3 Customizing the Window Creation Code

We will now take the C code generated by GUIDE and modify it as necessary to fit into our windows scheme. If the window will have a variable number of items then we will term the window a *custom window*. Custom windows need to have the creation code written by GUIDE modified in some small ways; non-custom windows do not.

#### 3.2.3.1 Customizing trial_ui.h

If the window is custom, then copy `trial_ui.h` to `trial_cui.h`. (The "cui" stands for custom user interface!) Now edit `trial_cui.h` by putting an asterisk in from of any item name which will needs to be multiply generated. We will be making the item into an array of items. For example, if we want to have a whole list of `varb_value` items on the window:

```
typedef struct {
  Xv_opaque       win;
  Xv_opaque       pan;
  Xv_opaque       button1;
  Xv_opaque       setting1;
  Xv_opaque       varb_value;
} trial_win_objects;
```

becomes

```
typedef struct {
  Xv_opaque       win;
  Xv_opaque       pan;
  Xv_opaque       button1;
  Xv_opaque       setting1;
  Xv_opaque       *varb_value;
} trial_win_objects;
```

#### 3.2.3.2 Customizing trial_cui.c

If the window is custom then copy `trial_ui.c` to `trial_cui.c`. Now perform the following changes to `trial_cui.c`.

1. Change the include file from `trial_ui.h` to `trial_cui.h`.

2. From the procedure trial_win_objects_initialize(), remove the creation calls for any item which will be multiply generated. Then remove the creation calls for any settings objects which will have a variable number of choice items. For example, if we will have multiple `varb_value` items and a varying number of choices in the `setting1` settings item.

```
trial_win_objects *
  trial_win_objects_initialize(ip, owner)
trial_win_objects *ip;
Xv_opaque owner;
{
  if (!ip && !(ip = (trial_win_objects *)
                   calloc(1, sizeof (trial_win_objects))))
    return (trial_win_objects *) NULL;
  if (!ip->win)
    ip->win = trial_win_win_create(ip, owner);
  if (!ip->pan)
    ip->pan = trial_win_pan_create(ip, ip->win);
  if (!ip->button1)
```

```
    ip->button1 = trial_win_button1_create(ip, ip->pan);
  if (!ip->setting1)
    ip->setting1 = trial_win_setting1_create(ip, ip->pan);
  if (!ip->varb_value)
    ip->varb_value = trial_win_varb_value_create(ip, ip->pan);
  return ip;
}
```

might become

```
trial_win_objects *
  trial_win_objects_initialize(ip, owner)
trial_win_objects *ip;
Xv_opaque owner;
{
  if (!ip && !(ip = (trial_win_objects *)
                    calloc(1, sizeof (trial_win_objects))))
    return (trial_win_objects *) NULL;
  if (!ip->win)
    ip->win = trial_win_win_create(ip, owner);
  if (!ip->pan)
    ip->pan = trial_win_pan_create(ip, ip->win);
  if (!ip->button1)
    ip->button1 = trial_win_button1_create(ip, ip->pan);
  return ip;
}
```

3. Edit the creation procedures for the custom text field objects so that the functions will accept an integer argument. This argument should be used in the code in order to correctly place the item on the window. As an example,

```
Xv_opaque
  trial_win_varb_value_create(ip, owner)
caddr_t ip;
Xv_opaque owner;
{
        .
                PANEL_VALUE_X, 135,
                PANEL_VALUE_Y, 64,
        .
}
```

could become:

```
Xv_opaque
  trial_win_varb_value_create(ip, owner, i)
caddr_t ip;
Xv_opaque owner;
int i;
{
```

```
        .
                PANEL_VALUE_X, 135,
                PANEL_VALUE_Y, 64 + 20 * i,
        .
}
```

4. Edit the creation procedures for the custom settings items so that the functions will accept an integer argument. This argument should be used in the code in order to set the number of choices. As an trial,

```
Xv_opaque
  trial_win_setting1_create(ip, owner)
caddr_t ip;
Xv_opaque owner;
{
        .
                PANEL_CHOICE_NROWS, 1,
        .
}
```

would become

```
Xv_opaque
  trial_win_setting1_create(ip, owner, n)
caddr_t ip;
Xv_opaque owner;
int n;
{
        .
                PANEL_CHOICE_NROWS, n,
        .
}
```

### 3.2.4 Writing the Panel Handlers

Create a file named `trial_hnd.c` to contain the notify handlers for this window. You will need to include some header files — the following should be sufficient (look in the GUIDE generated file `trial_stubs.c` for a complete set for your panel):

```
#include <xview/xview.h>
#include <xview/panel.h>
#include "trial_ui.h"              /* or #include "trial_cui.h" */
```

The notify handlers can be identical to the sample ones given below. This is because we use two routines, namely trial_read_window() and trial_data_refresh() which will read the data from the items in the window and, respectively, refresh the items in the window. These two routines are discussed in Sections 3.2.5.5 and 3.2.5.6.

#### 3.2.4.1 A Settings Notify Handler

This routine may be used as the notify handler for the settings items.

```
/*
 * trial_settings_notify()
 *
 * notify callback function for settings type panel item.
 */
void
  trial_settings_notify(item, value, event)
Panel_item item;
int value;
Event *event;
{
  trial_read_window();
  trial_data_refresh();
}
```

#### 3.2.4.2 A Text Field Notify Handler

This routine may be used as the notify handler for the text field items.

```
/*
 * trial_text_notify()
 *
 * notify callback function for text field type panel item.
 */
Panel_setting
  trial_text_notify(item, event)
Panel_item item;
Event *event;
{
  trial_read_window();
  trial_data_refresh();
  return panel_text_notify(item, event);
}
```

#### 3.2.4.3 A Button Notify Handler

Each button item will need it's own notify handler, because clicking on different buttons should result in taking different actions! Essentially, they must first make sure that any information in the postmaster that they will use or modify is up to date. This requires calling the read window routines for the windows from which the data is read. For our example, this will be the Trial window and the Selected window, since we will be writing to the selected window. Then we perform the computations which we desire. Finally, we must call the data refresh routines for all the windows for which we called the read routines.

However, they may all have the following form:

```
/*
 * trial_button1_notify()
```

```
 *
 * notify callback function for `button1'.
 */
void
  trial_button1_notify(item, event)
Panel_item item;
Event *event;
{
  trial_read_window();
  selected_read_window();
  trial_button1_go();
  trial_data_refresh();
  sel_data_refresh();
}
```

### 3.2.5 Writing the Standard Panel Procs

Six standard window routines accompany each panel. The first two trial_read_data() and trial_data_refresh() were mentioned above. A routine named trial_open() is used to create and open the panel. A routine named trial_close() closes the panel. A routine named trial_field_manager() is used to handle the custom fields on the panel. And last, a routine named trial_init() is used to handle the data structure which accompanies the panel. These routines should be placed in a file named `trial.c`. We now discuss each of these routines in turn.

#### 3.2.5.1 trial_open()

This is the routine which is called when the window's name is selected from the panels menu on the command window. It should create the window if necessary and then display the window. It has changed considerably in Version 2.0. The following code should be adequate in Version 1.x.

```
/* trial.c
 *
 * standard routines for trial window
 *
 */
#include <stdio.h>
#include <xview/xview.h>

#include <pm_main.h>
#include <ui_init.h>
#include <constants.h>
#include "trial_cui.h"

static trial_win_objects *trial_ip = NULL;

/*
 * trial_open()
 *
 * displays the trial window, creating it if necessary.
```

```
 */
trial_open()
{
  if (trial_ip == NULL)
    {
      /* create the window, make the owner the dstool cmd panel */
      trial_ip = trial_win_objects_initialize(NULL, ui_windows->cmd_win->win);
      trial_field_manager();
    }
  /* show the window */
  xv_set(trial_ip->win, WIN_SHOW, TRUE, WIN_FRONT, NULL);
}
```

while Version 2.0 requires:

```
/* trial.c
 *
 * standard routines for trial window
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <xview/xview.h>

#include <pm.h>
#include <ui_init.h>
#include <constants.h>
#include "trial_cui.h"

static trial_win_objects *trial_ip = NULL;

/*
 * trial_open(use_default,left,top,width,height)
 *
 * displays the trial window, creating it if necessary.
 */
trial_open(use_default,left,top,width,height)
int   use_default;    /* either SET_WIN_CONFIG  or  DEFAULT_WIN_CONFIG */
int   left;           /* position (on screen) of left edge of window.  */
int   top;            /* position (on screen) of top edge of window.   */
int   width;          /* width of window;  width <= 0 means use default width */
int   height;         /* height of window; height <= 0 means use default width */
{
  Rect
    *rect;

  if (trial_ip == NULL)
    {
      /* create the window, make the owner the dstool cmd panel */
      trial_ip = trial_win_objects_initialize(NULL, cmd_ip->win);

      /* The following 5 lines are for Version 2.0 only. */
      /* Add this window to the "Win_Names" postmaster object so that
```

```
 routines such as saveload know of its existence. */
      register_win_and_type("Trial",trial_ip->win,POPUP_WINDOW);
      /* Initialize any postmaster fields special to this window */
      /*    trial_install(); */

      trial_field_manager();
    }

  /* These lines allow the configuration of the window to be
     restored in Version 2.0 */
  if(use_default == SET_WIN_CONFIG)
    {
      rect = (Rect *)calloc(1,sizeof(Rect));
      frame_get_rect(trial_ip->win,rect);
      rect->r_left = (short) left;
      rect->r_top = (short) top;
      if(width>0) rect->r_width = (short) width;
      if(height>0) rect->r_height = (short) height;
      frame_set_rect(trial_ip->win,rect);
      free(rect);
    }


  /* show the window */
  mark_window_open("Trial");
  xv_set(trial_ip->win, FRAME_CMD_PUSHPIN_IN, TRUE, NULL);
  xv_set(trial_ip->win, WIN_SHOW, TRUE, WIN_FRONT, NULL);
}
```

#### 3.2.5.2 trial_close()

This routine, required only in Version 2.0, allows the postmaster to keep track of the open status of windows, and also supports batch and TCL closing of windows.

```
int
trial_close()
{
    mark_window_closed("Trial");
    if(trial_ip) {
xv_set(trial_ip->win, FRAME_CMD_PUSHPIN_IN, FALSE, NULL);
        xv_set(trial_ip->win, XV_SHOW, FALSE, NULL);
    }
}
```

#### 3.2.5.3 trial_field_manager()

This routine will be called whenever a dynamical system is loaded and immediately after the panel is first created. It should make sure that all the items in the window reflect the properties of the current dynamical system. It should also call the trial_init() routine in order to make sure that the data structure for the window is current for the selected dynamical system. If the window is not a custom window, meaning that it does not need to change when the dynamical system changes, then the procedure trial_init() may take it's place and the field manager need not exist. Here is a sample Version 2.0 field manager to accompany the custom window we have been developing.

```
/*
 * trial_field_manager()
 *
 * manager for the custom panel items
 */
trial_field_manager()
{
  int i,n_varb,status=0;
  char name[MAX_LEN_VARB_NAME];
  static int total_fields=0;


  /* exit now if the window has not been created */
  if (trial_ip == NULL || status != 0) return(status);

  /* initialize data structure for the trial window for the new dynamical system */
  status = trial_init();

  /* destroy the custom items */
  if (total_fields > 0)
    {
      for (i=0; i<total_fields; i++)
        {
          xv_destroy(trial_ip->varb_value[i]);
        }
      cfree((char *) trial_ip->varb_value);
      xv_destroy(trial_ip->setting1);
    }

  /* find out the number of variables */
  n_varb = *((int *) pm( GET, "Model.Varb_Dim", NULL ));
  total_fields = n_varb;

  /* allocate memory for the array of text fields */
  trial_ip->varb_value = (Xv_opaque *) calloc(n_varb, sizeof(Xv_opaque));
  if (trial_ip->varb_value == NULL)
    {
      total_fields = 0;
      system_mess_proc(1,"trial_field_manager: Memory allocation error.");
      return(-1);
    }

  /* create the window text fields, and write the label into them */
  for(i=0; i<n_varb; i++)
    {
      trial_ip->varb_value[i] = trial_win_varb_value_create(trial_ip,trial_ip->pan,i);
      pm(GET, "Model.Varb_Names", i, name, NULL);
      xv_set(trial_ip->varb_value[i], PANEL_LABEL_STRING, name, NULL);
    }
```

```
  /* set up the settings item choices */
  trial_ip->setting1 = trial_win_setting1_create(trial_ip, trial_ip->pan, n_varb);
  for (i=0; i<n_varb; i++)
    {
      pm(GET, "Model.Varb_Names", i, name, NULL);
      xv_set(trial_ip->setting1, PANEL_CHOICE_STRING, i, name, NULL);
    }

  /* make window tall enough  */
  window_fit(trial_ip->pan);
  window_fit(trial_ip->win);

  /* write data into the fields */
  trial_data_refresh();

  return(status);
}
```

#### 3.2.5.4 trial_init()

This procedure is designed to be called by trial_field_manager(). The code here should prepare and initialize a data structure to hold the information read from and written to the new window. This means that for every text field or settings item on the window, there must be an element of the data structure which will contain the value of the window item. In the simple example which we have been developing the data structure simply consists of an array of doubles to hold the values of the `varb_value` text fields and an integer to indicate the selected choice from the settings panel.

```
double *trial_varb_value = NULL;
int trial_choice;

int
  trial_init()
{
  int i,n_varb;


  n_varb =  *((int *) pm( GET, "Model.Varb_Dim", NULL ));


  /* free up previous storage, if any */
  if (trial_varb_value != NULL) cfree((char *) trial_varb_value);

  /* create storage for operations, and set initial values */
  trial_varb_value = (double *) calloc(n_varb,sizeof(double));
  if (trial_varb_value == NULL)
    {
      system_mess_proc(1,"trial_init: Memory Allocation failed.");
      return(-1);
    }

  /* initialize the values for the text fields */
```