Dynaforms and Dynaviews:

A Declarative Language for User Dialogs in Tcl/Tk

William H. Duquette
Jet Propulsion Laboratory, California Institute of Technology
William.H.Duquette@jpl.nasa.gov

Abstract

A dynaform is a specification of a dynamic data entry form: one whose content and layout can change based on the input of the user. Consider a GUI for entering rules for filing e-mail messages into folders: each rule can have many different forms, depending on the desired criteria. The user must first select the kind of criteria; each criterion has its own set of parameters, and the user's choices for those parameters might result in a further series of choices. The dynaform mechanism consists of a little language for specifying such a set of related inputs, infrastructure for processing that language, and the dynaview widget, which can display any desired dynaform and accept user input.

1. Background

The Athena Regional Stability Simulation is a model of political actors and the effects of their actions in a particular region of the world. The region is divided into neighborhoods, in which reside civilian groups. The actors have a variety of assets, including money, military and police forces, and means of communication, that they use to achieve their political ends. The extent to which they succeed depends on the attitudes of the civilians, which change in response to current events. The Athena user must enter a wide variety of data about many different simulation entities, and the complexity of the data to be entered has increased with each new version of the software. The dynaform mechanism is our latest approach to providing complex user-friendly easily maintainable data entry forms with a minimum of code.

2. The Problem

In the Athena application, an order is a specialized procedure used to handle user input. An order takes a dictionary of order parameters and values as input, validates them, and either throws a detailed error or performs the desired task. On success it returns an undo script; the application's undo/redo stack is in fact a stack of executed orders.

Some orders are simple, with one or two parameters that are always provided by the application; most are more complicated, and require significant input from the user. As a result, each order requires an order dialog. At present Athena contains 187 orders, and this number is always increasing. It is essential that implementation and maintenance of order dialogs be easy and straightforward.

2.1 Field Widgets

Tk provides a number of widget types useful for entering data, but their APIs often have subtle differences. Athena defines a standard field widget API, and uses Snit wrappers where necessary to make standard widget conform. The API was established early in Athena development, and has been remarkably stable since.

First, every field widget has at least the following options:

`-state` *`state`*
: Sets the widget's *state* to **normal** or **disabled** in the usual way.

`-changecmd` *`command`*
: Specifies a command prefix that will be called whenever the field's value changes for any reason (whether programmatically or due to user input). The *command* is called with one additional argument, the new value of the field.

Next, it has at least the following subcommands:

`field` `get`
: Retrieves the field widget's current value.

`field` `set` *`value`*
: Sets the field widget's *value*, calling the `-changecmd`.

Configuration is naturally field-type-specific, but once the field widgets are created and configured they all work just the same from the point of view of the form infrastructure.

2.2 Order Dialogs: The Mark I Solution

Our original solution to the order dialog problem was to associate enough metadata with each order definition to allow Athena to put together and pop up an order dialog. The first dialogs were quite simple: two parallel columns, with labels on the left and data entry fields on the right, implemented in a Tk grid. There were just a few field types, mostly text fields and combo box-based pull-downs; the metadata specified the list of valid items for the pull-downs.

These order dialogs often required additional code to provide dynamic behavior:

- Enabling or disabling fields based on prior entries in the dialog

- Configuring field widgets based on the state of the application (i.e., setting a pull-down to contain a list of currently defined simulation entities)
- Configuring field widgets based on the state of prior fields in the dialog.

Dynamic behavior was provided by means of a `-refreshcmd` callback associated with the dialog fields. Whenever the content of any field in the dialog changed, whether programmatically or due to user input, the dialog code called each field's `-refreshcmd` callback. The callback could enable or disable the field, or reconfigure it in a variety of ways. The difficulty was that each field was handled individually, when it was often necessary to coordinate changes to multiple fields.

Further, there were multiple reasons why a field might be refreshed, and the appropriate response differed. The callbacks needed to second guess the dialog logic in order to do the right thing. As a result, dialogs were finicky to get right, and when there was a problem it was hard to fix it without introducing problems elsewhere. Worse, the underlying dialog logic was convoluted and hard to follow.

2.3 Order Dialogs: The Mark II Solution

As a result, I rewrote the order dialog code from scratch, separating the main dialog logic from the form logic. I abstracted the form code out into a new `form` widget. The layout was still two parallel columns of labels and fields, but the order dialog code itself no longer needed to concern itself with the layout. The field-specific `-refreshcmd` callbacks were replaced by a single `-refreshcmd` callback that applied to the entire form. It was called with:

- The name of the order dialog, a Snit widget with a variety of introspection and configuration subcommands.
- A list of the names of the order parameters whose fields had changed
- A dictionary of the current values of all fields

The callback would then have to figure out the appropriate dialog configuration given the fields that had changed and the current field values, and update the dialog accordingly. It was now much easier to coordinate configuration changes to multiple fields, since all of the logic was in one callback. This scheme was a great improvement over its predecessor, but also proved to be unmanageable for complex dialogs. The complexity was all in one place—in the `-refreshcmd` callback—but refreshing still occurred for multiple reasons, and the callback code still needed to second guess the dialog logic in order to do the right thing. Writing robust code that caught all of the corner cases was difficult for any but the simplest patterns.

In the meantime Athena continued to grow, the number of orders continued to increase, and the individual orders continued to get more complex.

2.4 Lingering Issues

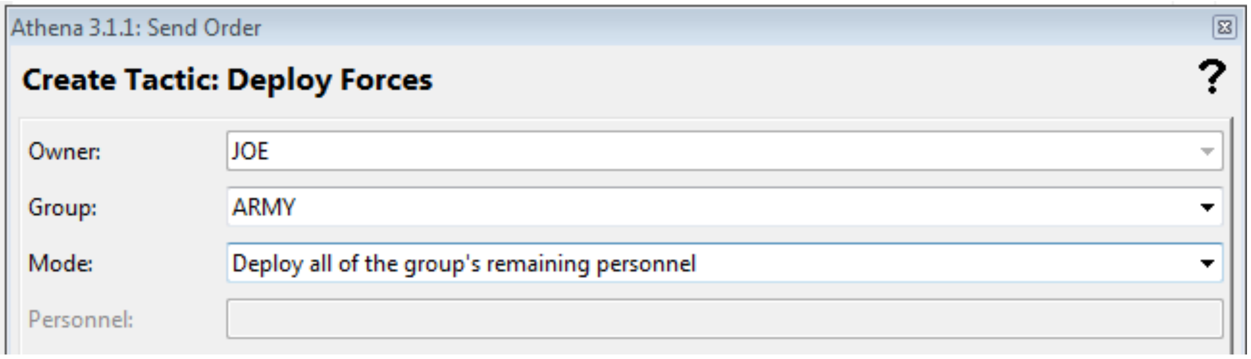
By this time we had a fairly sophisticated, maintainable set of code; but it was still unpleasantly limited.

- Dynamic behavior was easier to implement than before, but not truly simple.
- Form layout was limited to two parallel columns, and there was no easy way to change it.

3. An Example

As a simple example, the Athena analyst may create a DEPLOY tactic to deploy simulated troops into a neighborhood. He may choose the number of troops to deploy in two ways: he may deploy all remaining troops or he may deploy some specific number of troops; and in the latter case, he may deploy them with or without reinforcement. If he selects all remaining troops, he is done; but if he chooses to enter a specific number, he must enter the number and the reinforcement flag.

With the old system, the dialog looked like this:[1]


Athena 3.1.1: Send Order

Create Tactic: Deploy Forces

Owner:	JOE
Group:	ARMY
Mode:	Deploy all of the group's remaining personnel
Personnel:	

When the user chose to deploy all remaining personnel, the "Personnel:" field was disabled. When the user chose to deploy only some, the "Personnel:" field was enabled again, and the user could type in a number. Note that it was not possible to make the "Personnel:" field disappear when it was not wanted: the underlying "form" widget didn't support that.

[1] The reinforcement flag field is not shown, because it had not yet been added to the underlying order.

4. The Mark III Solution

The Mark I solution had the order dialog widget layout fields in a Tk grid based on the list of order parameters and a little metadata. Any relationships between fields were captured purely in external callbacks.

The Mark II solution made use of a Tk-grid-based form widget, and had the order dialog widget configure it based on the list of order parameters and a little metadata. The layout remained the same, and any relationships between fields were captured, again, in external callbacks.

The Mark III solution is radically different: the order dialog uses a new form widget, the `dynaview` widget, which can display any *dynaform*. A dynaform is a description of the form's fields and layout, including significant relationships between the fields, as captured in a declarative Tcl-based language. A dynaform specification becomes part of each order's metadata, but the dynaform language and infrastructure are independent of the order and order dialog infrastructure. In short, the new infrastructure is useful not only for order dialogs but for other general purpose dialogs.

The Mark III equivalent of the order dialog show above implemented as follows as a dynaform:

```
rcc "Owner:" -for owner
text owner -context yes

rcc "Group:" -for g
enum g -listcmd {group ownedby $owner}

rcc "Mode:" -for mode
selector mode {
    case SOME "Deploy some of the group's personnel" {
        rcc "Personnel:" -for personnel
        text personnel

        rcc "Reinforce?" -for reinforce
        yesno reinforce -defvalue 0
    }

    case ALL "Deploy all of the group's remaining personnel" {}
}
. . .
```

The `text`, `enum`, `selector`, and `yesno` commands each create a field, giving it a name and any required configuration options. The `rcc` commands add labels and layout hints. We'll take them in turn.

4.1 Field Specifications

All field creation commands have the following signature:

```
    fieldtype name ?option value...?
```

There are a number of standard field options, and each field type may have its own specific set of options.

The `owner` field names the actor to whose strategy the new tactic will be added. It is defined as follows:

```
    text owner -context yes
```

It is a text entry field; the –context option indicates that it is a "context" field: one whose value is set by the application to provide context to those that follow. The value of a context field cannot be edited by the user. Tactics are always created from a browser widget that shows the strategy of one actor at a time, and so the owner field is field in by the browser when the dialog is popped up.

The `g` field is used to enter the name of the force group to be deployed. It is defined as follows:

```
    enum g -listcmd {group ownedby $owner}
```

The `enum` command creates an enumeration field, which is rendered as a combo box containing a list of force groups to deploy. Each actor has his own set of force groups, so the content of the list has to be limited to those belonging to the owner. In the Mark II code, this would have been done in a clumsy way by the `-refreshcmd`, which when called would have had to figure out which fields needed to be updated and how. Here, instead, we simply define a `-listcmd`, which returns the desired list. It is called whenever the infrastructure determines that any prior field's value has changed.

Note that the `-listcmd` references the variable `$owner`. Rather than being called in the global namespace, as callbacks usually are, all field configuration callback commands are called in a context in which the form's field values are available as variables. This makes it trivially easy to make a field's configuration depend on a prior field in the same dialog. In this case, the command simply returns all of the groups owned by the tactic's `owner`.

The `reinforce` field is a Boolean field created using the `yesno` field type. This is an example of a custom field type; it simply wraps the `enum` field type with the required configuration options to display "Yes" and "No" and map them to the integers 1 and 0. It was defined by the Athena application itself to streamline this common case.

4.2 Selector Fields

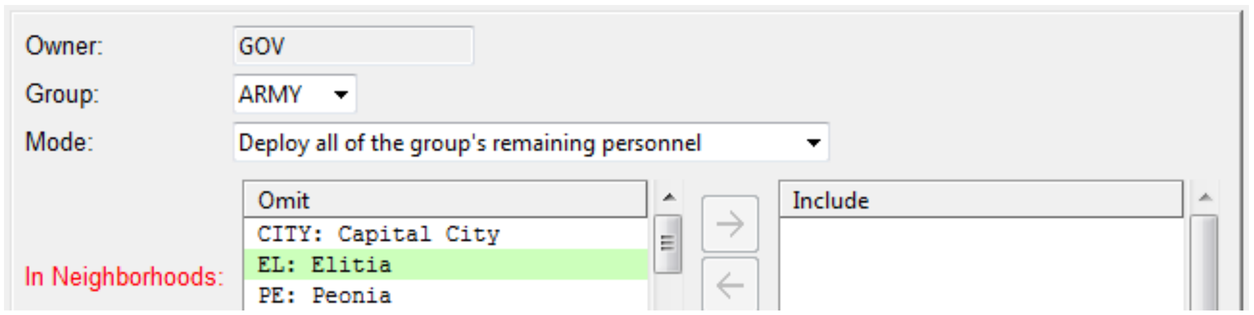
Next we come to the mode field, which specifies how the number of personnel to deploy is to be determined. It is defined as follows:

```
selector mode {
    case SOME "Deploy some of the group's personnel" {
        rcc "Personnel:" -for personnel
        text personnel

        rcc "Reinforce?" -for reinforce
        yesno reinforce -defvalue 0
    }

    case ALL "Deploy all of the group's remaining personnel" {}
}
```

The `selector` field type controls the form's displayed content in a special way. After any options, it takes a body script which contains `case` commands; each `case` command takes a body script which can contain arbitrary dynaform commands.

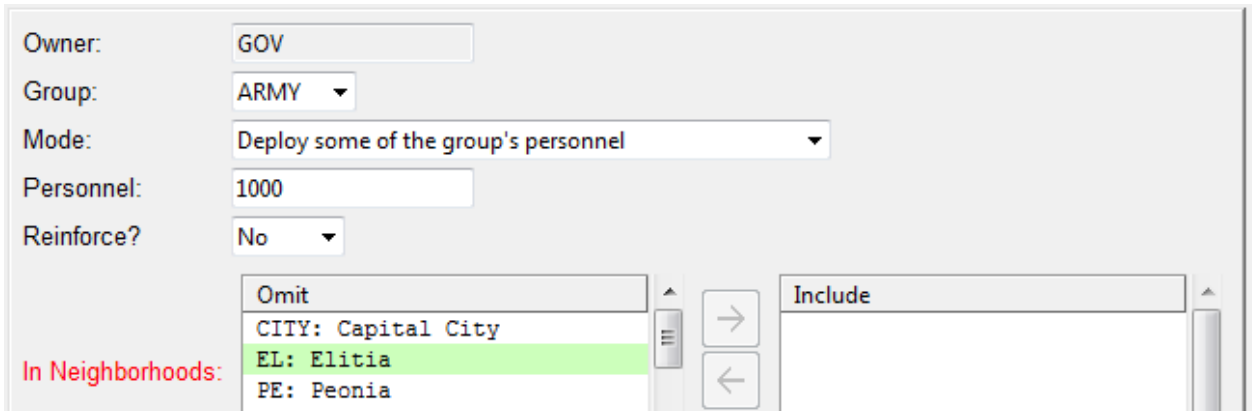
The `mode` field will be displayed as a combo box with one item for each of the cases in the selector body. The field's value will be **SOME** or **ALL**, and the combo box will display the associated text, e.g., "Deploy some of the group's personnel". When the user selects a case, the fields defined in that case's body will be displayed. For this dynaform, the dialog will display the `personnel` and `reinforce` fields only when the `mode` is **SOME**.

For example, the dialog looks like this when the mode is **ALL**:



Owner:	GOV
Group:	ARMY
Mode:	Deploy all of the group's remaining personnel
In Neighborhoods:	

but like this when the mode is **SOME**:



Owner:	GOV
Group:	ARMY
Mode:	Deploy some of the group's personnel
Personnel:	1000
Reinforce?	No
In Neighborhoods:	

Selector fields can be nested arbitrarily deeply.

4.3 Dynamic Behavior

The Mark I and Mark II versions of the order dialog infrastructure allowed for three kinds of dynamic behavior in response to changes in the content of the fields:

- Changing the configuration of a field, e.g., setting the list of valid items for a pull-down field.
- Loading record data into the dialog when a key field's value changes.
- Changing the layout in some way

Dynaforms support these same behaviors, but in a significantly different way. In the Mark I and Mark II code it was necessary to write `-refreshcmd` callbacks; these callbacks had to contain significant logic to determine what just happened and the current state of the dialog, and make the required changes. This required second-guessing the underlying dialog code, and unless the programmer had a pretty good notion of how that code worked, it was hopeless. Even if the callbacks did mostly what was wanted it was hard to catch all of the corner cases.

With dynaforms, each of the three behaviors listed above is handled in an appropriate way.

Fields that require dynamic configuration do so by a callback that returns the required configuration information given the values of the prior fields in the dialog and the state of the application. The `enum` field, for example, has a `-listcmd` option; the command must return the current list of valid items, which is passed directly to the underlying field widget. The dynaform infrastructure ensures that the callback will be called whenever needed; and the callback need not concern itself with the dialog or second-guess its behavior or state. The command usually can just do a simple data retrieval, without any complex logic.

Key fields are handled by a `-loadcmd` callback; this is an option that can be used with any field type. The callback is called whenever the field's content changes in any way; and its job is to return a dictionary of field names and values to load into the dialog. Again, the command usually just does a simply data retrieval. All of the logic is handled by the infrastructure.

Finally, it is no longer necessary to write custom logic to control the dynamic layout of the dialog. Instead, such dynamic behavior is stated declaratively in the dynaform specification script. No callbacks are involved.

In short, dynaforms get the encapsulation layer in the right place. There are a few things a form might need to know, and callback options are provided so that the dynaform can ask for them; but the application programmer need only provide information. The dynaform's logic is internal, where it belongs.

4.4 Layout Hints and Layout Engines

The remaining commands in the dynaform language create label text and provide layout hints; the `rcc` command does both. For example, the label for the `personnel` field is created as follows:

```
    rcc "Personnel:" -for personnel
```

This command assumes that a table layout is being used. It creates a new row and column, places the label "Personnel:" in that column, and then creates a new column for any subsequent content. The `-for` option relates it explicitly to the `personnel` field; when the field's value is invalid, the label will be drawn in red.

The `rcc` command is referred to as a layout "hint" because the dynaform language doesn't presuppose any particular mechanism of laying out the fields; rather, it is aimed at defining the fields and linking them to each other and to the required application data. The hints are there for the use of the GUI when it displays the dynaform, which it is free to do in any way it likes. In particular, the dynaform language does not itself assume the use of Tk; it could in principle be used with Gtk or (on the server side) with HTML or JavaScript displayed in a browser.

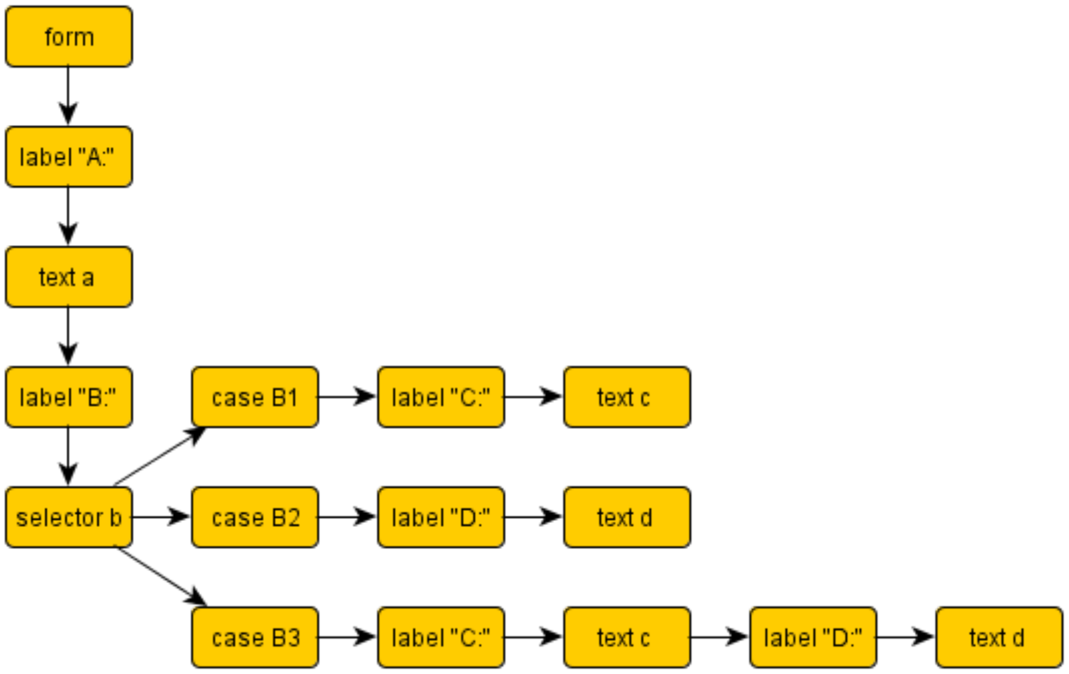
4.5 The Item Tree

Consider the following dynaform.

```
label "A:" –for a
text a

label "B:" –for b
selector b {
    case B1 "B1:" {
        label "C:" –for c
        text c
    }

    case B2 "B2:" {
        label "D:" –for d
        text d
    }

    case B3 "B3:" {
        label "C:" –for c
        text c

        label "D:" –for d
        text d
    }
}
```

First there is field `a`, which is always displayed. Next, there is selector `b`, which is always displayed. The selector has three cases; the first displays field `c`, the second field `d`, and the third both. Each field has an associated label. Each command in the form defines an *item*, and these items form a tree, as shown in the following diagram.

If this were a real form, it would be associated with an order with fields `a`, `b`, `c`, and `d`, and the processing of fields `c` and `d` would depend on the value of field `b`. Note that there are fields named `c` and `d` on several different branches. This is perfectly OK. Laying out the dynaform involves walking this tree, and picking the branches corresponding to the selector values. A field name can appear on any number of branches, just so long as it does not appear twice on the path from the root to any leaf node.



The path from the root to a leaf given the current selector values is called the *current path*.

4.6 Creating a Dynaform

Dynaforms are created using the `dynaform define` command:

```
dynaform define MYFORM {
    # form specification commands
}
```

Most dynaforms in Athena are associated with orders; the form specification is entered as part of the order's metadata and `dynaform define` is called automatically.

5. The Dynaview Widget

Dynaforms are displayed by the `dynaview` widget; the dynaform to display is determined by the `-formtype` option, which names any defined dynaform. The manner in which the form is displayed is determined by the selected layout algorithm. Given the algorithm, the widget follows the current path from the root to a leaf, using or ignoring hints and laying out labels and fields. How the hints are used depends on entirely on the chosen layout algorithm.

5.1 Layout Algorithms

At present, the dynaview widget supports three different layout algorithms: **ribbon**, **2column**, and **ncolumn**. The default is **ncolumn**, but a form can select a different algorithm using the layout command, e.g.,

```
    layout ribbon
```

The **ribbon** algorithm lays out labels and fields in one row, horizontally, ignoring other layout hints.

The **2column** algorithm lays out labels and fields in two parallel columns, like the Mark I and Mark II order dialog code.

The **ncolumn** algorithm allows labels and fields to be laid out in table cells, as with a Tk grid or HTML table. The layout hints indicate row and column breaks.

The real point to allowing multiple layout algorithms is "future-proofing". The current layout scheme is working for us, but the architecture allows us to define additional schemes for particular purposes without changing any existing dynaforms.

The underlying geometry manager in each case is a Tkhtml3 widget; thus, the layout algorithm is generally producing HTML text. Tkhtml3 has a number of advantages over a Tk grid; it is easy to include rich text in the form, and lines of label text and field widgets wrap more attractively. However, the dependence of the API on Tkhtml3 is minimal, and future layout algorithms could use a different underlying widget.

5.2 Widget API

The `dynaview` widget has the following options:

`-formtype` *`name`*
Sets the name of the dynaform to display.

`-state` *`state`*
Sets the widget's *state* to **normal** or **disabled** in the usual way.

`-changecmd` *`command`*

Specifies a command prefix that will be called whenever the form's value changes for any reason (whether programmatically or due to user input). The *command* is called with one additional argument, the new value of the form.

`-currentcmd` *`command`*

Specifies a command prefix that will be called whenever the one of the form's fields receives the focus. The *command* is called with one additional argument, the name of the field that received the focus.

Next, it has the following subcommands (among others):

`form` `get`

Retrieves the form's current value: a dictionary of field names and values.

`form` `set` *`dict`*

Sets the values of the form's field widgets given the field names and values in the *dict*, and calls the `-changecmd`.

`form` `current`

Returns the name of the field that has the input focus (or the first field, if no field has the input focus).

`form` `invalid` *`fields...`*

Marks the named fields invalid. The effect is determined by the layout algorithm; for **ncolumn**, for example, the associated fields are colored red.

`form` `clear`

Clears all non-context fields, and fills in default values.

`form` `refresh`

Refreshes all fields from first to last; all fields will now be configured in accordance with the current state of the application.

Note that the `dynaview` widget adheres to the field widget API described in section 2.1. This is often useful when a field's value isn't a simple scalar value. Instead, a custom field type can be defined based on the `dynaview` widget and an appropriate dynaform.

5.3 The dynabox Command

The dynabox command pops up a modal dialog containing a dynaform, with "OK" and "Cancel" buttons. On "OK" it returns the value of the dynaform (i.e., `[$form get]`); on "Cancel" it returns the empty string. It is roughly similar to the standard tk_messageBox command, but displays an arbitrary dynaform. Thus, dynaforms can be used for general data entry, not simply for order dialogs. This is another advantage of dynaforms over the Mark I and Mark II solutions.

6. Dynaform Language

This section gives a rough overview of the dynaform language. We divide the commands in the language into two sets: content and layout.

6.1 Content Commands

The content commands are as follows:

`fieldType name ?option value...?`

Adds a field of the given type with the given *name* to the form. There are a number of predefined field types, including `text` and `enum`, as described above, and the application programmer can add more. The options are used to configure the field. The following options can be used with any field type.

`-context flag`

If *flag* is true, the field is a read-only "context" field. It provides context to the user, and its value might be used by field configuration callbacks.

`-defvalue value`

Specifies a default value for the field; the value is placed in the field when the form is cleared.

`-invisible flag`

If *flag* is true, the field is invisible, i.e., is not displayed in the dialog. This allows the application to provide context data for use by field configuration callbacks without making it visible to the user.

`-loadcmd command`

This command is called when the field's value changes; its purpose is to load data into the dialog when a key field's value changes. It is given the field's current value and other configuration data, and returns a dictionary containing field names and values. Any field whose name matches a dictionary key is updated with the value from the dictionary.

`-tip` *`text`*

Specifies tool-tip text for the field. This option is a layout hint, and how the text is used depends on the layout algorithm.

`label` *`text`* `?-for` *`field`*`?`

Adds the *text* to the form; it can contain rich text in the form of HTML markup. If the `-for` option is included then the text is a label for the named *field*, and may be highlighted (i.e., colored red) when the field's content is in error.

`selector` *`name`* `?`*`options...`*`?` *`selscript`*

The selector command adds a special pull-down field that controls the content of the dialog. It has the given *name* and takes all of the standard field options. The *selscript* contains one or more `case` commands; the user selects the case from the pull-down.

`case` *`case label script`*

This command can only appear in a `selector` field's *selscript*, where it adds another case to the selector. The *case* is a symbolic constant used as the `selector` field's value when the case is selected, and the *label* is corresponding text that appears in the pull-down. The *script* contains any arbitrary dynaform commands (fields, labels, and layout hints); the form items in the *script* are displayed when the case is selected by the user.

`when` *`expr tscript`* `?else` *`fscript`*`?`

This command is like a selector case in that it controls the items that will be displayed; however, it is not a field. Instead, *expr* is a Boolean expression that may reference upstream fields by name (as well as arbitrary commands). If the expression is true, then the items in the *tscript* will be laid out; otherwise those in the *fscript* (if any) will be laid out.

6.2 Layout Commands

`layout` *`algorithm`* `?`*`options...`*`?`

Specifies the layout algorithm to use. Currently, the choices are **ncolumn** (the default), **2column**, and **ribbon**. The choice of algorithm determines how the layout hints are used. Most existing hint commands are used only by the **ncolumn** algorithm.

`br`

Adds a line break to the form.

`c ?`*`label`*`? ?`*`options...`*`?`

For **ncolumn** layouts, starts a new column. If *label* is given, a label is added as the first item in the column.

`-for` *`field`*
Relates the label to a field, as for the `label` command.

`-span` *`n`*
The new column will span *n* table columns; defaults to 1.

`-width` *`width`*
The desired column width, in HTML length units, e.g., "3in" or "100pxi".

`cc` *`label`* `?`*`options...`*`?`
For **ncolumn** layouts, starts a new column, places the *label* text in it, and starts a second column. The options are the same as for the `c` command.

`para`
Adds a paragraph break to the form.

`rc ?`*`label`*`? ?`*`options...`*`?`
For **ncolumn** layouts, starts a new row and column. The arguments are as for the `c` command.

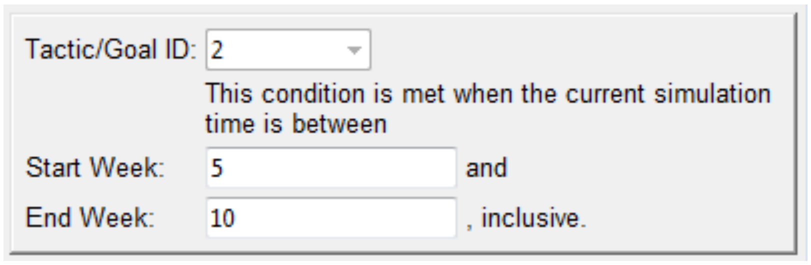
`rcc` *`label`* `?`*`options...`*`?`
For **ncolumn** layouts, starts a new row and column, places the *label* text in it, and starts a second column. The options are the same as for the `cc` command.

7. Examples

This section gives a number of additional examples of dynaforms and the resulting dialogs.

7.1 Dynaform with Help Text

The following dynaform is designed to contain help text so as to be immediately understandable to the user. It is used to enter inputs for a Boolean condition called "DURING":



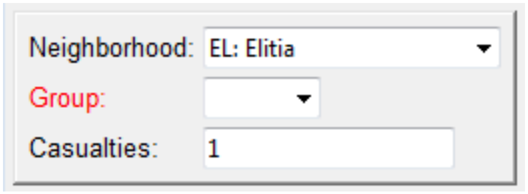
Tactic/Goal ID: 2
This condition is met when the current simulation time is between
Start Week: 5 and
End Week: 10 , inclusive.

The form specification is as follows; `condcc` is a custom field type.

```
rcc "Tactic/Goal ID:" -for cc_id
condcc cc_id

rcc "" -width 3in
label {
     This condition is met when the current simulation time
     is between
}

rcc "Start Week:" -for t1
text t1
label "and"

rcc "End Week:" -for t2
text t2
label ", inclusive."
```

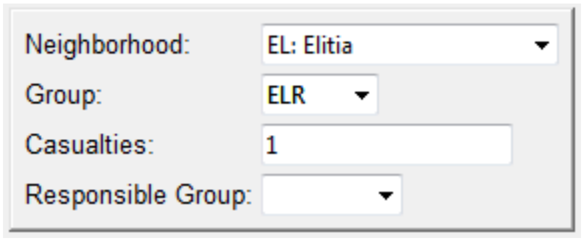
7.2 Dynaform with "When" Conditions

The following dynaform grows depending on the user input. It is used to enter some number of casualties to a group in the simulation.



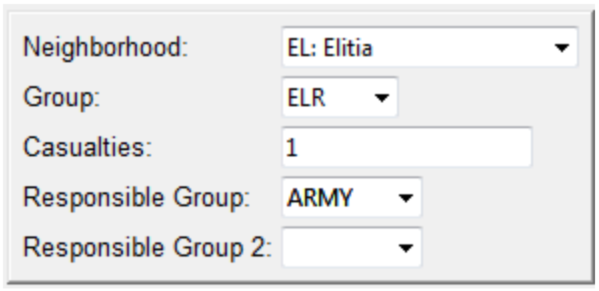
Neighborhood:	EL: Elitia
Group:	
Casualties:	1

First the user must select a group. If it's a civilian group, then Athena wants to know what force group is responsible for the casualities (if any). A "Responsible Group" field appears.



Neighborhood:	EL: Elitia
Group:	ELR
Casualties:	1
Responsible Group:	

For historical reasons, Athena allows there to be up to two responsible force groups. If a name is chosen for the first, a second "Responsible Group" field appears.



Neighborhood:	EL: Elitia
Group:	ELR
Casualties:	1
Responsible Group:	ARMY
Responsible Group 2:	

The dynaform specification is as follows; `nbhood` and `frcgroup` are custom field types.

```
rcc "Neighborhood:" -for n
nbhood n

rcc "Group:" -for f
enum f -listcmd {::aam GroupsInN $n}

rcc "Casualties:" -for casualties
text casualties -defvalue 1

when {$f in [::civgroup names]} {
    rcc "Responsible Group:" -for g1
    frcgroup g1

    when {$g1 ne ""} {
        rcc "Responsible Group 2:" -for g2
        enum g2 -listcmd {::aam AllButG1 $g1}
    }
}
```

8. References

[1] Duquette, William, Snit Object Framework, found in Tcllib, http://tcllib.sourceforge.net/doc/snit.html.

9. Acknowledgements

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during the development of the Athena Stability & Recovery Operations Simulation (Athena) for the TRADOC G2 Intelligence Support Activity (TRISA) at Fort Leavenworth, Kansas.