CASEVision / Tracker 2.0 Design Guide

Document Number 007-1664-020

CONTRIBUTORS

Written by Margaret-Anne Halse
Production by Gloria Ackley
Engineering contributions by Pete Orelup

CASEVision / Tracker 2.0 Design Guide
Document Number 007-1664-020

Contents

Figures

Tables

Introduction

This guide describes CASEVision™ / Tracker 2.0, a highly flexible tool from Silicon Graphics, Inc.®, that enables software organizations to create their own tracking systems for bugs and enhancement requests. It is intended for the designers of such systems. It is assumed that the designer has had exposure to one or more high-level programming languages and understands the concepts of database design and administration.

CASEVision / Tracker 2.0 runs on IRIX™ 5.1. For system and memory requirements, please refer to the CASEVision / Tracker 2.0 Release Notes.

About This Guide

This guide has the following chapters:

- Chapter 1, "Introduction to Tracker System Design," defines the Tracker terminology and gives an overview of the design cycle and tools.
- Chapter 2, "Using the Process Description Language (PDL)," describes the structure of the PDL and the contents of the PDL file, including help declarations.
- Chapter 3, "Using the Data Manipulation Language (DML)," gives an overview of the database structure and sets out the characteristics of DML statements.
- Chapter 4, "Tutorial—A Basic Tracking System," steps users through the process of creating and modifying a Tracker application.
- Chapter 5, "Installing RTS Applications," details the procedure for installing RTS applications.

- Chapter 6, “Advanced Design Techniques,” discusses advanced techniques such as using dates, customizing resources, using the *exec* functions, and importing data from external databases.
- Chapter 7, “Configuration Management,” outlines the *checkout* / *checkin* model and describes how to use the ClearCase / Tracker integration.
- Appendix A, “The policy_vars.sh File,” defines the environment variables used by Tracker applications and stored in this file.
- Appendix B, “RTS PDL Files with On-line Help,” gives the complete text of the four *.pdl* files, including the on-line help text.

Chapter 1

Introduction to Tracker System Design

CASEVision / Tracker 2.0 is a highly flexible tool that enables software organizations to create their own systems for tracking bugs and enhancement requests. There is no need to change your current method of making and tracking requests. Unlike commercial off-the-shelf systems, which are usually limited to a specific methodology and are difficult to modify, Tracker systems can be built from the ground up or from one of the starter systems provided.

The starter systems are *rtsquery*, a full-featured tracking system, and *sample1*, a more rudimentary system that can be used as a building block.

This chapter covers these topics:

- Tracker terminology
- Tracker design tools
- Tracker design cycle

Tracker Terminology

To understand Tracker system design, it is helpful to know a few of the terms used in Tracker. The term *request* refers to the type of data that you are tracking in the system, typically bug reports or requests for enhancement (RFE). A full report contains all the information associated with the request, such as submitter, owner, priorities, and dates.

Application Views

The program that a user runs to access a request is an *application*. A *view* is a window for inspecting or changing a request. An application generally has a *main view* with most of the request information and one or more *auxiliary views* (see definition below). The main view is special in several ways: it appears when you first start the application, whereas auxiliary views are windows that open from the Views menu in the main view. Exiting the main view exits the entire application, while exiting an auxiliary view merely closes that one window.

Figure 1-1 shows *rtsquery*, the starter application provided by Silicon Graphics Inc., which is a typical Tracker main view.

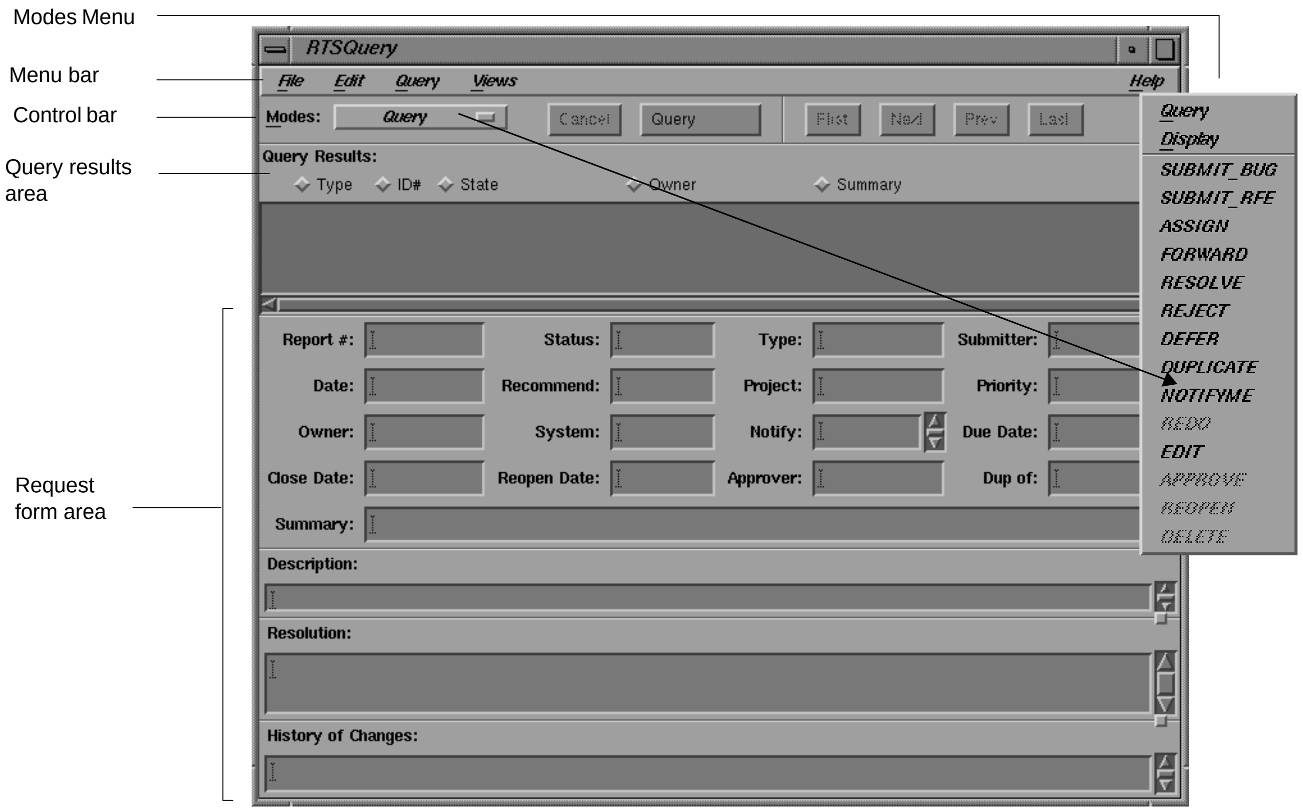
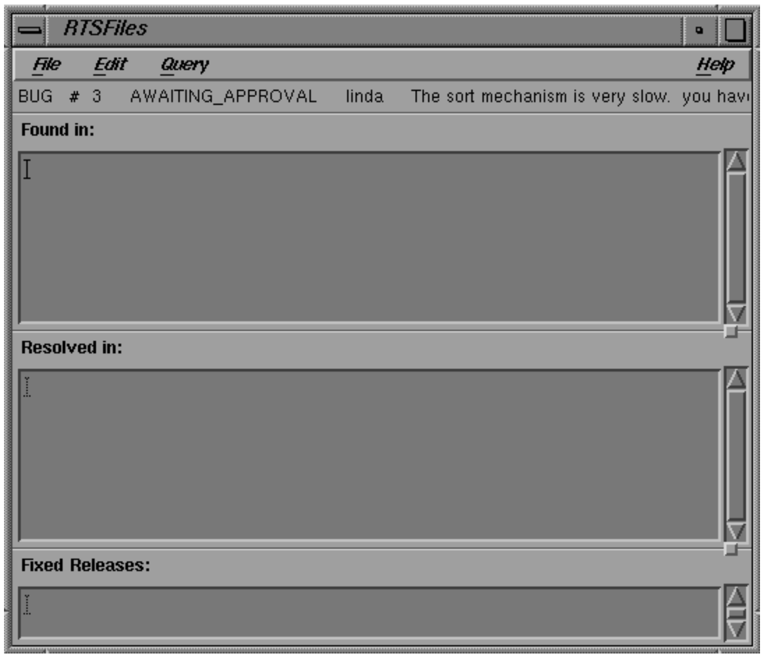


Figure 1-1 Major Parts of *rtsquery*, a Typical Tracker Window

The main sections of the window are labeled. *Report #*, *Description*, and *Type* in the request form area represent typical fields.

If you want different main views for different types of users, you can create *supplementary applications*. A supplementary application uses the same request database as the main application but has a different main view and potentially different auxiliary views. Supplementary applications can serve as filters for users doing specific tasks that don't require all of the fields (see definition below).

Auxiliary views supplement the main view and are used for special data or for users who require limited information or functionality. *RTSfiles*, for example in Figure 1-2, lets users enter the names of files in which a bug has been located into its three fields: *Found in:*, *Resolved in:*, and *Fixed Releases:*. The bug itself is listed at the top.



BUG # 3 AWAITING_APPROVAL linda The sort mechanism is very slow. you hav

Figure 1-2 *rtsfiles*, a Typical Tracker Auxiliary View Window

Display Areas

Within a window, there are *display areas,* which are groupings of related fields, enclosed by boxes. A *field* is a piece of information in a request. A field that has multiple lines may have a sash control. A *sash* is a small square "knob" near the right edge of a window that lets users change the size of a multi-line field by dragging it up or down. Fields are discussed in detail in "Field Declarations" and display areas are discussed in "View Declarations" in Chapter 2.

Transitions

A *transition* is an operation performed through a view that changes the request either in terms of status or data. Transitions are accessed from the *Modes menu* in the view's control bar (see Figure 1-1). Note that the selections "Query" and "Display" also appear in the Modes menu but are used for inspection rather than request modification.

A transition can have associated rules and actions. A *rule* is a required condition necessary for the transition to take place. For example, *rtsquery* requires a description entry for submitting requests. An *action* is an operation performed as part of the transition. For example, when a request is submitted, the submit date changes to the current date.

Transitions are discussed in detail in "Transition Declarations" in Chapter 2.

Tracker Design Tools

A Tracker request tracking system has a graphical user interface (GUI) and a database of requests. Tracker provides these tools for designing a system:

- process description language (PDL)
- data manipulation language (DML)
- Tracker application generator (*tvgen*)
- Tracker X resources file (*app-defaults*)

The PDL File

At the heart of a Tracker request tracking system is the *PDL file*, a file coded in the process description language. A single PDL file can define a complete request tracking system, including the data to be recorded, the operations allowed on that data, rules controlling those operations, automatic actions performed by the system, and the application from which users can access the request database. The PDL is discussed in Chapter 2, “Using the Process Description Language (PDL).”

The Data Manipulation Language

The *data manipulation language* (DML) is an interface to the Tracker database. It is similar to SQL and other fourth-generation database query languages. All queries and database transactions are made through the DML. The system administrator controls access to the database through the DML, and can set permissions for individuals who need to make modifications to the database using DML. Figure 1-3 illustrates database access in Tracker.

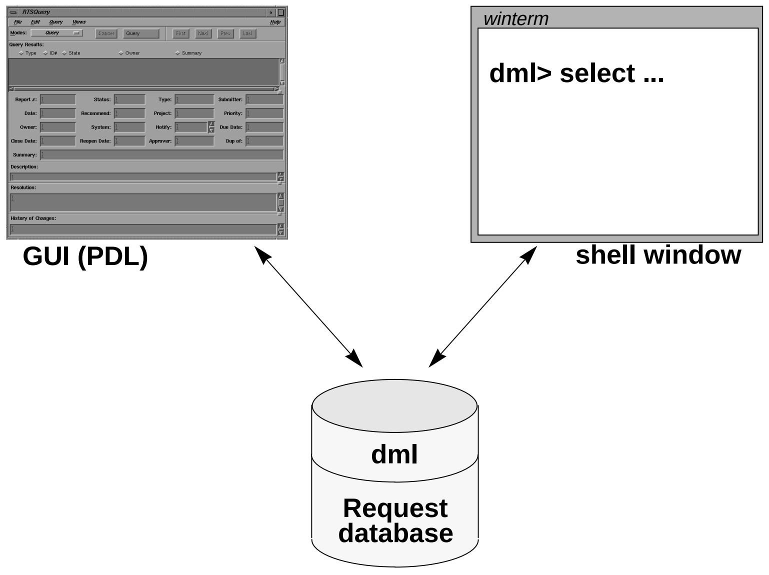


Figure 1-3 Overview of Tracker Database Access

Most users access the database from a Tracker application window (generated from a PDL file). You can also access the DML directly from a shell for more complex queries on the database, report generation, and batch database modifications. The DML is hidden from users in the GUI. Database access is discussed in Chapter 3, “Using the Data Manipulation Language (DML).”

The *tvgen* Program

The *tvgen* program generates Tracker application files from PDL files, as shown in Figure 1-4. It produces a script containing user-editable views, a Tracker database with supporting files and directories, an *app-defaults* file containing user interface resources, and help files. Field and transition definitions are stored in the database. After you run *tvgen*, you can make changes to any of the new files.

Finally, each user must run *tvinstall* to create links to the application files. Installation using *tvgen* and *tvinstall* is described in Chapter 4, “Tutorial—A Basic Tracking System.”

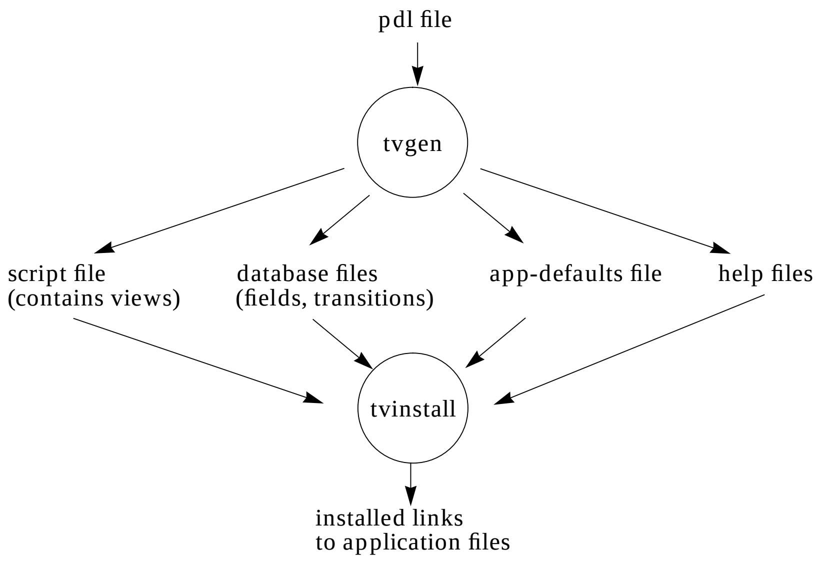


Figure 1-4 Generating and Installing Tracker Applications

The *app-defaults* File

The *app-defaults* file lets you fine-tune many aspects of the appearance and behavior of the applications. Tracker provides a number of predefined features that can be selected through the PDL. You can make other adjustments to the *app-defaults* file as well before distributing the complete applications to the users. For more information, see “Customizing Resources” in Chapter 6, “Advanced Design Techniques.”

Tracker Design Cycle

The designer of the request tracking system typically performs the following steps to build a request tracking system for an organization:

1. Identify all the information to be tracked, the users, groups, and any special needs they have.
2. Define the desired request tracking and approval process. In particular, define the different states of a request as it goes through the process. (A state transition or data flow diagram may be useful.)

   The state transition diagram for requests in the *rtsquery* system is shown in Figure 1-5.
3. Determine whether to use one of the starter systems supplied by Silicon Graphics or build a new system from scratch.

   Note: As with other programming methods, sometimes it is easier to modify an existing system than to create a new one.
4. Develop a PDL file to match the desired process.
5. Compile the PDL file and specify the request database by using the *tvgen* utility.

   tvgen checks the syntax, creates an empty database if necessary, loads the PDL into the request database, takes care of on-line help, and stores the application in a subdirectory called *tools* for mounting or copying to other systems.

   The application’s name is based on the name of the main view defined in the PDL file, shifted to all lower case (the convention for IRIX commands). For more information, see the *tvgen* man page.

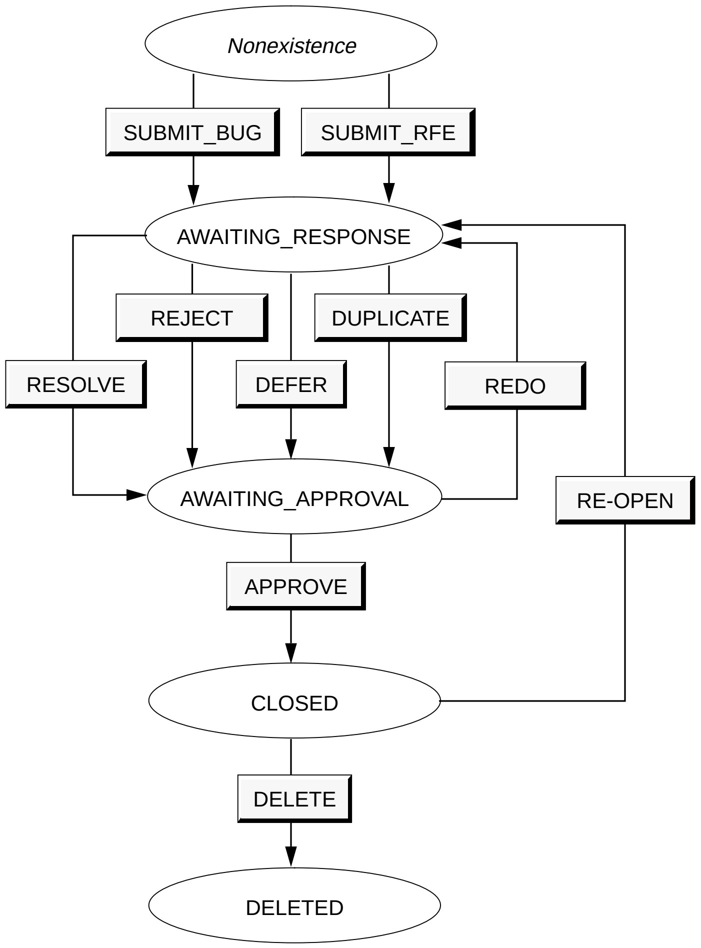


Figure 1-5 State Transition Diagram for a Request in *rtsquery*

6. Provide further customization if desired.

   Typical customizations at this point include resource settings in the *app-defaults* file (named *tools/Tracker.adinstall*) or addition of any external scripts used by applications. For more information, see Chapter 6, “Advanced Design Techniques.”

7. Let users access the new application.

   The *tools* subdirectory, created in the database directory by *tvgen*, should be copied or NFS-mounted onto each user’s system (it does not matter where). Then, run the script *tvinstall* on the user’s system to complete the installation.

When the system is fully installed, the user can start the application generated by *tvgen* by entering its name on the command line.

Using Starter Systems

Note that the Tracker starter systems do not come already installed. Before you can use them, you must install *sample1* and / or *rtsquery* by following the instructions in Chapters 4 (for *sample*1) or 5 (for *rtsquery*).

Chapter 2

Using the Process Description Language (PDL)

The field declarations, transition definitions, and graphical user interface for Tracker applications are specified by using a special process description language (PDL). This chapter describes the PDL and consists of:

- PDL file
- Field declarations
- Transition declarations
- View declarations
- Help declarations

The PDL File

A *PDL file* (a file coded in the process description language) defines a complete request tracking system, including the data to be recorded, the operations allowed on that data, rules controlling those operations, automatic actions performed by the system, and the application from which users can access the request database.

After you have gathered the information necessary for defining your process, you can put together your PDL file.

Application Components

Specifically, Tracker allows you to define the following for a system:

- fields in the request
- transitions, in terms of:
  - name
  - prior state and new state
  - rules
  - actions
- views (the windows used to access request information). Some typical display areas in view windows are:
  - control bar
  - query results area
  - request form area, the rows of request information to be displayed, including labels and fields
- help information accessible through the on-line help system

PDL Structure

The declarations and top-level expressions in the PDL resemble those in a block-structured language such as C. The blocks are delimited by braces ({}). Declarations are separated by semicolons (;).

White space, including newlines, is ignored. Comments are allowed, using either the C form (/* *comment* */) or the C++ form (//*comment*).

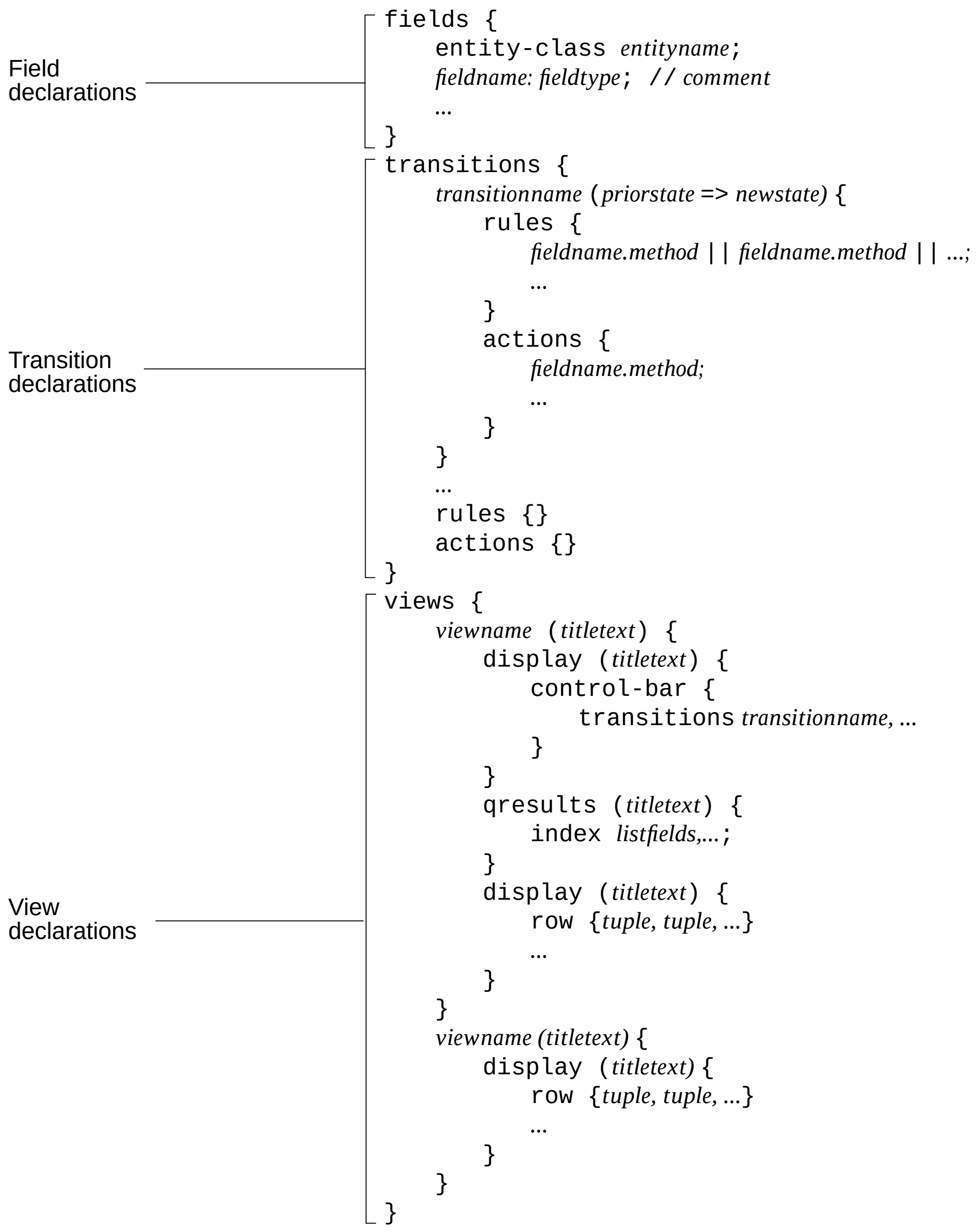
You can use the *#include*, *#define*, and *#ifdef* C preprocessor constructs.

A PDL file has three major sections (which must be declared in this order):

1. Field declarations, where the fields in the request are named and where their types are declared.
2. Transition declarations, which define the states through which requests can pass and the rules for controlling the process.
3. View declarations, where the GUI applications to interact with the request database are defined.

The format for a PDL file is shown in Figure 2-1. Help declarations, that is, the information on help cards in the on-line help system, can be made for most items in a PDL file. Since they take up a lot of space, the help declarations are not shown in Figure 2-1. They are described in “Field Declarations” later in this chapter.

Note: The PDL files for supplementary applications contain view declarations only, since they use the same field and transition declarations as their main view PDL files.



```
fields {
    entity-class entityname;
    fieldname: fieldtype;  // comment
    ...
}
transitions {
    transitionname (priorstate => newstate) {
        rules {
            fieldname.method || fieldname.method || ...;
            ...
        }
        actions {
            fieldname.method;
            ...
        }
    }
    ...
    rules {}
    actions {}
}
views {
    viewname (titletext) {
        display (titletext) {
            control-bar {
                transitions transitionname, ...
            }
        }
        qresults (titletext) {
            index listfields,...;
        }
        display (titletext) {
            row {tuple, tuple, ...}
            ...
        }
    }
    viewname (titletext) {
        display (titletext) {
            row {tuple, tuple, ...}
            ...
        }
    }
}
```

Figure 2-1 Format for a PDL File

Field Declarations

Fields are defined by type at the beginning of the PDL file. You can declare fields that display in the user interface as well as fields for internal purposes such as scratch variables. Note that you declare only fields in this section. You specify default values for fields (and other manipulations) in the view declarations section.

Field Types

The field types available in Tracker are shown in Table 2-1. Those types that display predefined values through the "Values" item in the field menus are indicated. The "Values" menu is displayed when the user clicks the right mouse button on a field.

Table 2-1 Tracker Field Types

Type Name	Comments
boolean	Holds boolean data (true / false). Displayed as either `True` or `False`. Accepts entries of `True, False, T`, or `F`. The "Values" menu item displays "True" and "False."
`date`	Tracker accepts a wide range of formats for dates and times. To see the full range, refer to "Using Dates" in Chapter 6, "Advanced Design Techniques."
file	Holds a file name. The "Values" menu item displays a pop-up file selection dialog. If a file name is entered through the GUI, the file must exist. The file type has special implications for ClearCase® users (see "Using file with ClearCase" below).
int	Holds a 32-bit signed integer.
journal	Holds multiple lines of text. Allows the system administrator to maintain a history of the changes made to any request. Instead of overwriting the current information, changes are appended to the existing history, for as long as the request is active. See "Using the journal Field" below.
list-of	Holds a list of any of the other scalar field types.

Table 2-1 (continued) Tracker Field Types

Type Name	Comments
long-text	Holds multiple lines of text. Intended for fields containing explanations. There is no “Values” menu item but there is an “Edit...” item for using an external editor.
one-of	Holds an enumeration value, that is, one of a set of predefined values. Values are separated by commas and can be legal identifiers or quoted, single-line text. Its use is described in detail below. The “Values” menu item displays either a sub-menu or a pop-up selection dialog, depending on the number of values.
short-text	Used for single-line text. Intended for short entries.

Using *file* with ClearCase

When entering a file in a field of type *file*, Tracker responds as follows:

- for a plain file: it verifies that the file exists.
- for a file in a ClearCase Versioned Object Base (VOB): Tracker verifies that the file exists, determines which version is selected by the current ClearCase view, and records the version information in the *file* field.

The menu for file fields includes two items useful for obtaining information about ClearCase files, available if the field has a valid value and the file referred to is a VOB file. They are “Describe” and “History,” both accessible from the “Actions” sub-menu. They use ToolTalk™ to communicate with ClearCase. “Describe” executes the cleartool `describe -long` and “History” executes the cleartool `lshistory` command on the file referred to by the field. Both commands display their results in a tty window, created as needed, which is separate from the Tracker application. Subsequent command results appear there as well.

Using *one-of* Fields

The default *one-of* list is a closed set from which a user must select one of the pre-defined values shown in the “Values” menu or dialog box. You can specify an open *one-of* set by entering an ellipsis (. . .) immediately (no comma) after the last value. Use open sets to specify the values used most often and to allow entry of non-predetermined values.

A *one-of* field may contain any single-line text value. In PDL, *one-of* field values may appear either as identifiers or as single-line quoted strings. A legal identifier must begin with an alpha character (upper or lower case). It may include letters, the underscore (_), and digits, in any order. You may not begin a legal identifier with $, since the $ prefix marks predefined fields or environment variables.

A single-line quoted string value may use the full character set. You can mix legal identifiers with quoted literal strings in a *one-of* field definition. For example;

```
x: one-of
    a, b, c, ’123’, ’sarah’, ’4.0.1’ ...
```

The *short-text* values let you pull in values from external programs by using the *exec* commands (see “Using the exec Functions” in Chapter 6, “Advanced Design Techniques”). It also lets you cross-assign values between fields. You can use the *setValue* method to set the *one-of* field value to *short-text*. For example, suppose that the *Submitter* field is a *one-of* field with the values being a list of engineers in a particular group. You can set the value to default to the current user of the application ($USER), which is a *short-text* field.

If a *one-of* field has 25 values or less (default), Tracker creates a cascading submenu off its “Values” menu item in the GUI. If there are more than 25 values, Tracker builds a selection dialog box that presents the user with a scrolled list from which to choose a value.

To change the default setting for the number of fields required for a scrolled list, you must set the *maxAssistValues* resource in the *app_defaults* file (see “Personal Tracker Resources” in Chapter 6, “Advanced Design Techniques.”

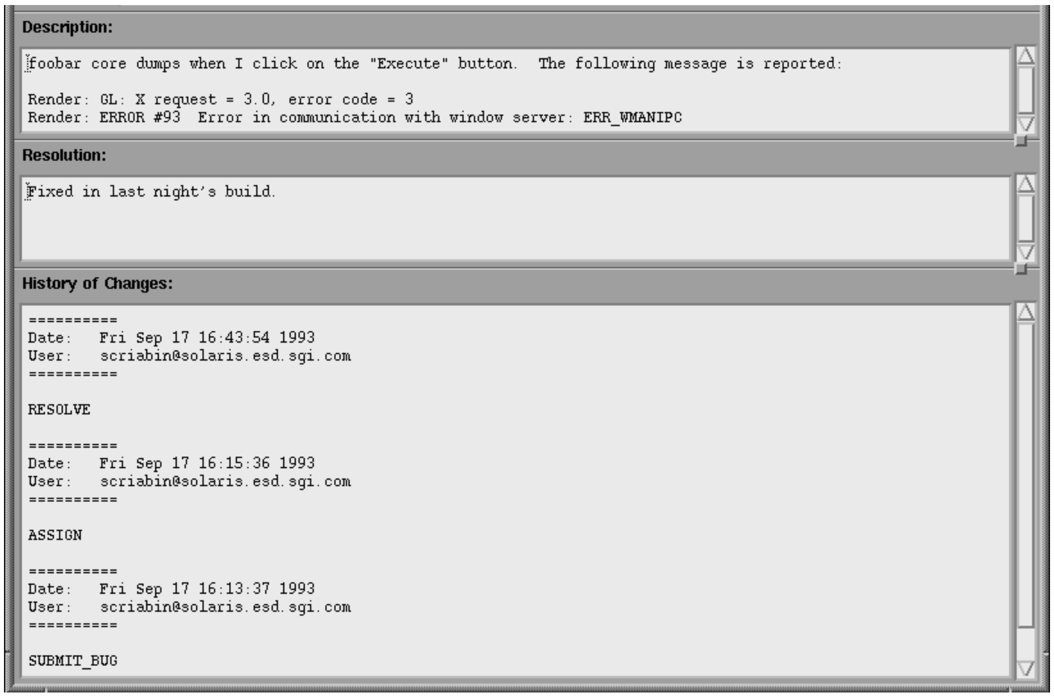
Duplicated Values for *one-of* Fields

There are two limitations on using *one-of* fields:

- The same value cannot appear in more than one closed enumeration in a *one-of* field. You can, however, share an entire enumeration between two fields by listing both field names in the same declaration or by using *like* (see below).
- You may not use the same identifier for a *one-of* field value and a state identifier. If you do, Tracker produces an error message indicating that the state identifier is a duplicate name. Transition states are discussed in more detail in “Transition Declarations.”

Using the *journal* Field

Figure 2-2 shows the history of the actions taken on a request in a Tracker application. The actions SUBMIT_BUG, ASSIGN, and RESOLVE are recorded, along with the date and the submitter’s name. Note that you cannot customize the format of the entryheader when you design your application; Tracker creates the headings automatically.



```
foobar core dumps when I click on the "Execute" button.  The following message is reported:

Render: GL: X request = 3.0, error code = 3
Render: ERROR #93  Error in communication with window server: ERR_WMANIPC
```

```
Fixed in last night's build.
```

```
==========
Date:   Fri Sep 17 16:43:54 1993
User:   scriabin@solaris.esd.sgi.com
==========

RESOLVE

==========
Date:   Fri Sep 17 16:15:36 1993
User:   scriabin@solaris.esd.sgi.com
==========

ASSIGN

==========
Date:   Fri Sep 17 16:13:37 1993
User:   scriabin@solaris.esd.sgi.com
==========

SUBMIT_BUG
```

Figure 2-2 Entries in the Journal Field

You can write anything you like to the body of the journal entry using *setValue()*. The RTS example PDL file stores only the transition name in the journal entry body, using:

```
history.setValue($TRANSITION.text);
```

This is an example of a more complex journal entry:

```
tmpShortText.setValue(
execFilter('echo "Transition from state $STATE_old
            to $STATE"'));
history.setValue(tmpShortText.value);
```

Figure 2-3 shows the field declaration for a journal field.

Declaring Fields with *like*

The key word *like* lets you share a field type between two fields and apply different help text.

For example;

```
fieldname1: fieldtype1
   help {helpcardspec1};
fieldname2: like fieldname1
   help {helpcardspec2};
```

In the example, *fieldname2* has the same type as *fieldname1*, that is, *fieldtype1*. It uses a different help card specification, *helpcardspec2*.

Declaring Entities

At the beginning of the field declaration section, the following entry can appear:

```
entity-class entityname;
```

Tracker fields in a PDL file are grouped into an entity for direct access from DML. To specify a custom name for an entity, you must declare it here; otherwise, the default name *tracker_request* is used.

The concept of entities in the database is covered in more detail in Chapter 3, “Using the Data Manipulation Language (DML).”

Entity Identification

Tracker automatically assigns an entity identification number ($ENTITY_ID) to each new request as it is submitted to the database. Because the ID number is assigned prior to the execution of any actions, its value can be used in an action to set the value of other variables. For example, where `report_number` is of type `int`:

```
actions {
        report_number.setValue($ENTITY_ID.value);
        ...
        }
```

Fields Declared by Tracker

The following fields are predeclared and controlled solely by Tracker:

- *$STATE* holds the current state of the request.
- *$TRANSITION* is set to the name of the current transition.
- $ENTITY_ID is a unique integer assigned to each request when it is entered into the system.

Environment variables are also available as predeclared fields. They can be changed by the PDL and used in transition rules and actions through the external methods (see “External Methods.” later in this chapter).

Field Declaration Example

The code segment in Figure 2-3 shows the field declarations for the PDL file used to generate the *rtsquery* application.

Each declaration shows the field on the left and its type on the right. Notice how the *short-text* type is used for short entries and how *long-text* is used for the multiple-line fields. Fields with the *one-of* designator are followed by the defined set of values.

```
fields {
   report_number:     int;
   submitter:         short-text;
   submit_date:       date;
   recommendation:    one-of
          DEFERRAL, REJECTION, RESOLUTION, DUPLICATION;
   type:              one-of
                      BUG, RFE;
   priority:          one-of
                      LOW, MEDIUM, HIGH;
   owner:             short-text;
   project:           one-of
#include "projects.h" // This include file contains the
                      // list of projects. Edit it to
                      // change the list of known projects.
   ;
   system:            one-of
                      SYSTEM_1, SYSTEM_2, SYSTEM_3;
   found_in:          list-of short-text;
   summary:           short-text;
   description:       long-text;
   is_duplicate_of:   int;   //pr-num
   interested_parties:list-of short-text;
   due_date:          date;
   close_date:        date;
   reopen_date:       date;
   resolved_in:       list-of file;
   resolution_description:long-text;
   fixed_releases:    list-of short-text;
   approver:          short-text;
   history:           journal
// These fields are not visible to the users
   czar:              short-text;
   bboard:            short-text;
   notify_list:       list-of short-text;
}
```

Figure 2-3 Field declarations for the PDL file

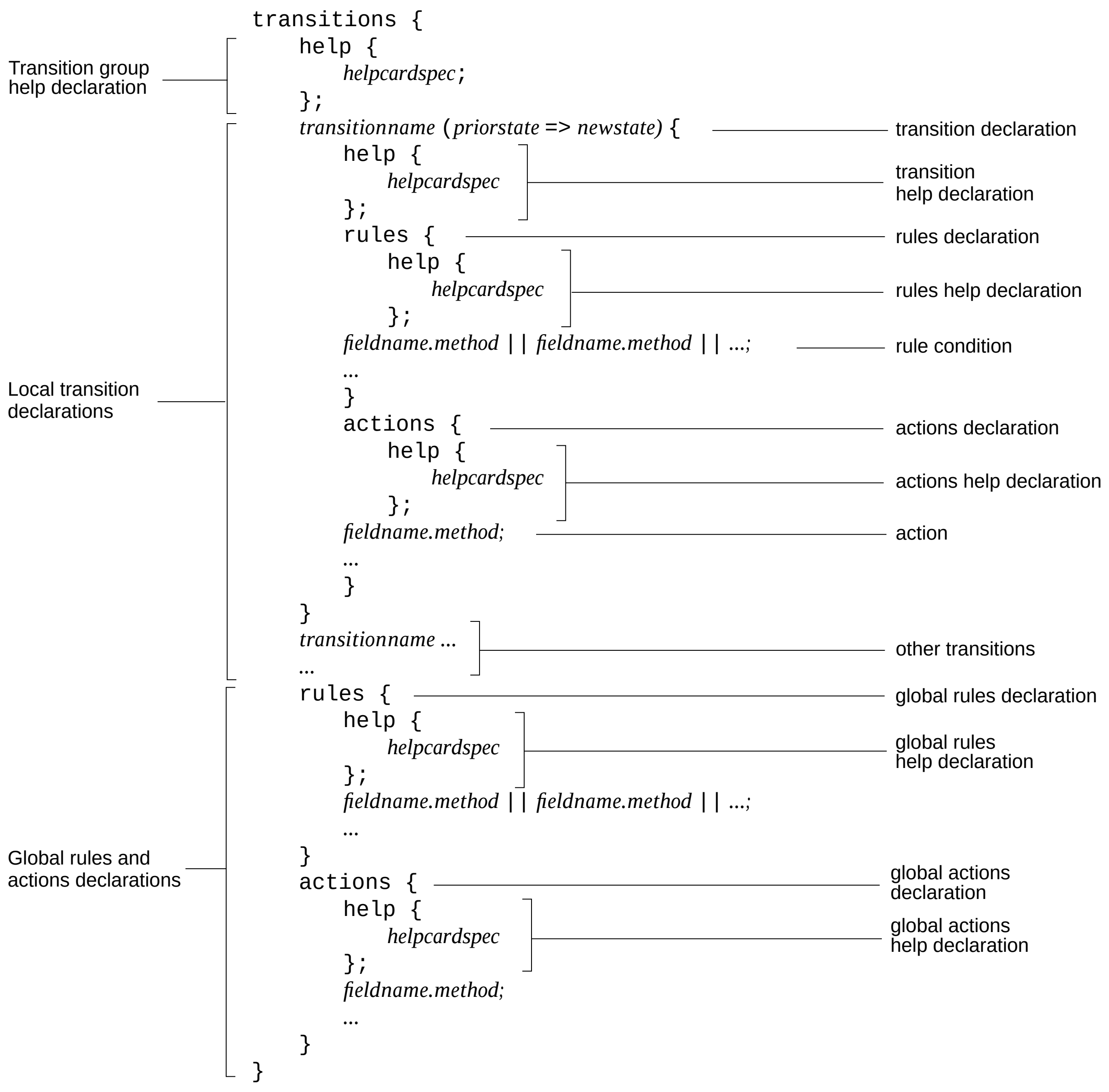
Transition Declarations

Transitions are the operations performed on a request to change its state or data. To declare a transition, do the following:

- Enter the name of the transition in the transitions section of the PDL file. This name will appear in the Modes menu and will be enabled for appropriate states.
- Define the change in state (if any) as a result of this transition.
- Define any rules required for the transition to take place.
- Define any actions that result from the operation.
- Define global actions and rules for your transition group. These are specified at the bottom of the transitions section and are applied to all transitions after the local rules or actions are applied to the individual transition.
- Define on-line help as desired. You can define help for the transitions, individually and as a group, for a set of rules belonging to a transition, for a set of actions belonging to a transition, and for the global rules and transitions.

  Note: Wherever you provide help text for a rule or an action you can use the key word `include-pdl` to include the actual rules or actions declarations in the help card; this is recommended only for sophisticated end users or for debugging transitions.

The format for the transitions section is shown in Figure 2-4. Reserved words are shown in normal font; variables are shown in italics. On-line help declarations are also shown.



```
transitions {
    help {
        helpcardspec;
    };
    transitionname (priorstate => newstate) {
        help {
            helpcardspec
        };
        rules {
            help {
                helpcardspec
            };
            fieldname.method || fieldname.method || ...;
            ...
        }
        actions {
            help {
                helpcardspec
            };
            fieldname.method;
            ...
        }
    }
    transitionname ...
    ...
    rules {
        help {
            helpcardspec
        };
        fieldname.method || fieldname.method || ...;
        ...
    }
    actions {
        help {
            helpcardspec
        };
        fieldname.method;
        ...
    }
}
```

Transition group help declaration
Local transition declarations
Global rules and actions declarations
transition declaration
transition help declaration
rules declaration
rules help declaration
rule condition
actions declaration
actions help declaration
action
other transitions
global rules declaration
global rules help declaration
global actions declaration
global actions help declaration

Figure 2-4 Format for the Transitions Section in PDL File

Transition Example

The following code segment provides an example of how transitions are declared. The code segment is the declaration for the RESOLVE transition in the *rtsquery* application. RESOLVE is used when a request has been executed and needs to be signed off by the approving authority.

```
RESOLVE(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
   help {
      help-title ’RESOLVE Transition’;
      short-help-title ’RESOLVE’;
      fixed-width-help-text’
Use RESOLVE to close out a request. RESOLVE takes a
request from the AWAITING_RESPONSE state to the
AWAITING_APPROVAL state.’;
   };
   rules {
      resolution_description.isSet;
      resolved_in.isSet;
   }
   actions {
      recommendation.setValue(RESOLUTION);
   }
}
```

The first line contains the transition name RESOLVE and the change of state, from AWAITING_RESPONSE to AWAITING_APPROVAL.

The help declaration (which is optional) comes next. Notice that this example has a full title and a short title. The fixed-width option is also used in the help text declaration. This option is described in “Creating a Help Declaration” later in this chapter.

Transition declarations can have a rules section for establishing information required for the transition and an action section for specifying what takes place as a result of the transition. Tracker also provides methods, that is, operations on fields, that are useful in creating rules and actions.

In the example, the rules section contains two rules, both of which must be true for the transition to be performed. They both use the `isSet` method.

The rule

```
resolution_description.isSet;
```

requires that the *Resolution* field in the main view be filled in.

The line

```
resolved_in.isSet;
```

requires that the *Resolved in* field in the RTS files view be entered.

The actions section has only one action in this example. It uses the *setValue* method to change the *Recommendation* field to the value RESOLUTION, signifying that the request has been resolved.

Declaring States

Request states are declared inside parentheses that follow transition names, separated by the transition operator, =>, as follows:

(*priorstate (priorstate)*$_n$ => *newstate)*

These are the options for using the transition operator:

(=>statename)	creates the request where none previously existed.
(priorstate=>newstate)	signifies a change from the prior state to the new state. Both states must be specified.
(=>)	signifies that any prior state is permitted and that there is no change in state as a result of this transition.
(S_1, S_2, S_3 ...=>S_4)	the prior state can be any one of S_1, S_2, S_3, etc. All prior states go to one new state, that is, S_4. For example, a number of different states could all go to CLOSE. All states must be specified.

RTS transitions such as “NOTIFYME” and “EDIT” have no required states. Transitions for creating new requests require a new state but no prior state.

Caution: If you use the same state in more than one transition, it must match in case and spelling; otherwise, different versions of the name will be interpreted as different states.

The *rtsquery* application (see Appendix B) defines these states:

- AWAITING_RESPONSE
- AWAITING_APPROVAL
- CLOSED
- DELETED

Declaring Rules and Actions

After the transition's state change is declared, you declare the rules, if any, and the resulting actions, if any. Tracker supplies predefined methods for declaring rules and actions. Tracker also lets you use the following logical operators (shown in Table 2-2), borrowed from the C language.

Table 2-2 Logical Operators

Operator	Description
&&	and
||	or
!	not
?:	ternary (if ... then ... else ...) For example, `priority.setValue(product_released.value ? 3 : 5)` sets `priority` to `3` if `product_released` is `true` or to `5` if `product_released` is `false`.

All the operators except the *&&* (*and*) operator function in top-level rules. The use of *&&* is restricted to embedded expressions in the PDL file; for example:

```
rules {
   x.is(x.setValue(name.isSet ? address.isSet && company.isSet :
                    False));
      }
```

For more information on *and*ing top-level conditions, refer to “Rules” below.

In addition, Tracker automatically provides actions setting the *$STATE* field to the new state and the *$TRANSITION* field to the transition name.

Rules

Rules are declared after the state declaration; they are preceded by the key word *rules* and are enclosed by braces ({}). In the transition example, the RESOLVE transition has these rules:

```
rules {
    resolution_description.isSet;
    resolved_in.isSet;
}
```

They require that there be entries in both the *Resolution* field and the *Resolved in* field.

An individual rule is terminated by a semicolon (;) and consists of one or more conditions. You define conditions using the boolean Tracker methods. An individual condition can be *or*ed with other conditions, using the operator `||`. This means that the entire rule (terminated by the semicolon) is satisfied if any one (or more) of the individual conditions is satisfied. Rules separated by semicolons (;) are effectively *and*ed together, that is, all of the rules must be satisfied in order to permit the transition.

If all conditions are not met, the transition will not be permitted (its *Apply* button will not be enabled) and incorrect fields will be highlighted accordingly. It is useful to think of a rule as having *top-level* and *secondary-level conditions*. A top-level condition is the main part of a rule; the secondary level refers to conditions inside nested expressions. When a rule is not met, only the fields in top-level rules are highlighted. The fields at the secondary level are not highlighted. For example, consider the rule:

```
owner.is(submitter.value);
```

The rule means that the *owner* value must be the same as the *submitter*. If this condition is not met, then the *owner* field (which is a top-level rule) will be highlighted. The *submitter* field is in a nested expression and will not be highlighted. It is a good idea to explain such relationships between fields in an on-line help card, either in the transition help card or the associated rule help card if it exists.

Actions

Actions immediately follow rules. They are preceded by the key word *actions*. Actions can change the values of fields. Like rules, they are defined in terms of methods. Where rules mainly use boolean methods, actions use boolean methods only to set values subject to conditions.

Predefined Methods

The predefined methods are presented in Table 2-3 through Table 2-8 according to these categories:

- external methods
- methods for testing field data
- methods for retrieving field data
- methods for making field comparisons
- methods for changing field values
- methods for field value computations
- methods for changing field characteristics

External Methods

Three methods provide access to shell commands (and environment variables) external to the PDL: *execCommand*, *execFilter*, and *execSelect*. They take a single string parameter and are not associated with fields. These methods are described in Table 2-3. For more information, see “Using the exec Functions” in Chapter 6, “Advanced Design Techniques.”

Table 2-3 External Methods

Method	**Description**
execCommand	Returns the completion status of the command as a `boolean` value. It is most useful when the output of the command is not required by the PDL, but the exit status of the command may control further PDL execution or indicate the validity of field data in a rules section.
execFilter	Returns the output of the command executed as a *long-text* value. It is most useful in the actions section of transitions. For example, use it to capture the output of a command and assign it to a field.
execSelect	Given a DML (data manipulation language) select statement with semicolon (;) terminator, returns a *long-text* value. The *select* statement is executed. If the result is a single record, the value of the first field in the *select* statement is the return value; otherwise, the return value is the empty string.

Here are some useful Tracker environment variables, accessible through the external methods:

- *$FIELD_LIST* is set to name all the fields of the request (structured as a string with names separated by spaces).
- *$MODIFIED_FIELDS* is set to name all fields that have changed.
- *$<fieldname>* is created for each field of the request, using the name of the field as the name of the variable and generally containing the value of the field.
- *$<fieldname>_old* holds the former value of the field if it changes. The field name also appears in $MODIFIED_FIELDS.

- *$<fieldname>_file* contains the actual value of a field if it is too long for the *$<fieldname>* variable. In this case, *$<fieldname>* contains the single `!` character and *$<fieldname>_file* contains the actual field value(s).
- *$<fieldname>_old_file* holds the former value of a long field if it changes. (In such a case, *$<fieldname>_old* contains the single `!` character.)

Methods for Testing Field Data

The methods for testing field data are called against a field, for example:

```
description.isSet
```

They do not take parameters and do not accept the empty parentheses that C uses: for example, `description.isset()` is incorrect. They are predicates; they return a `boolean` result, so they can be used in rules. The methods are listed in Table 2-4.

Table 2-4 Methods for Testing Field Data

Method	**Description**
<*fieldname*>.changed	Any field type. Determines whether it has been changed during the current editing session.
<*fieldname*>.isSet	Any field type. Determines whether the field is presently set (some fields may not yet have been filled in).
<*fieldname*>.not	Must be a field of type `boolean`. If the field is set to `True`, then this says false; if the field is `False`, this says true.

In addition, you can test any field of type `boolean` with the `value` method (described in Table 2-11), as in this example:

```
boolfield.value;
```

If `boolfield` is true, then this rule will be satisfied; if `boolfield` is false, or if it is unset, this rule will be unsatisfied (`boolfield` will be highlighted in the GUI).

Methods for Retrieving Field Data

The methods in Table 2-5 return other types of data in addition to booleans. They are used only inside expressions within parameter lists, typically in conjunction with a boolean method. Since there are no side effects from these methods, they are not appropriate for use as top-level methods in actions, although this is legal.

Table 2-5 Methods for Retrieving Field Data

Method	Description
<*fieldname*>.fname	Any field type. Returns the name of the field.
<*fieldname*>.length	Must be a field of type `list`. Returns the number of items in a list.
<*fieldname*>.old	Any field type. Returns the value of this field in the database, even if the user has made changes (but has not yet committed them to the database).
<*fieldname*>.size	For a text string, tells how long it is.
<*fieldname*>.text	Returns a text string representation of any kind of field.
<*fieldname*>.value	Returns the value of the field, represented as the appropriate type.

Methods for Making Field Comparisons

The methods for making field comparisons are called against a field and take an expression as a parameter.

For example,

```
priority.isGreater(3)
```

tests whether the priority field is greater than the value 3. These methods are shown in Table 2-6.

Table 2-6 Methods for Making Field Comparisons

Method	**Description**
<*fieldname*>.is(*expr*)	Checks to see if the value of the field is *expr*.
<*fieldname*>.is_cf(*expr*)	*fieldname* and *expr* must be strings. Checks to see if they're the same value, but with case folding (ignoring case).
<*fieldname*>.isLess(*expr*)	Can be performed on most types, but only where *fieldname* is the same type as *expr*. Checks whether *fieldname* is less than *expr*, according to their types.
<*fieldname*>.isLessEq(*expr*)	Can be performed on most types, but generally only where *fieldname* is the same type as *expr*. Checks whether *fieldname* is less than or equal to *expr*, according to their types.
<*fieldname*>.isGreaterEq(*expr*)	Can be performed on most types, but generally only where *fieldname* is the same type as *expr*. Checks whether *fieldname* is greater than or equal to *expr*, according to type.
<*fieldname*>.isGreater(*expr*)	Can be performed on most types, but generally only where *fieldname* is the same type as *expr*. Checks whether *fieldname* is greater than *expr*, according to their types.
<*fieldname*>.isNot(*expr*)	Can be performed on most types, but generally only where *fieldname* is the same type as *expr*. Checks whether *fieldname* is not equal to *expr*, according to their types (opposite of "is").

Methods for Field Value Computations

The methods for field value computations let you perform computations using field values. They take an expression, which can contain nested methods as well. For example, the action

```
due_date.is(due_date.setDefault('now + 30:00:00:00'));
```

computes the value of the expression (`'now + 30:00:00:00'`), which is equal to the current date plus 30 days, sets that value as a default in the date field, and uses that value if another has not been entered.

These methods have two categories: methods that actually change the field values and methods that perform calculations with field values.

Table 2-7 shows the methods for changing field values.

Table 2-7 Methods for Changing Field Values

Method	Description
<*fieldname*>.setValue(*expr*)	Sets the value of the field to *expr*.
<*fieldname*>.unsetValue	Causes the field to have no assigned value. A subsequent `isSet` method will return `false`.
<*fieldname*>.setDefault(*expr*)	Sets the field to the value *expr* if the field value has not yet been set.
<*fieldname*>.append(*expr*)	Applies to lists only, adding *expr* to the end of the list. Resulting expression is the list itself.
<*fieldname*>.remove(*expr*)	Applies to lists only, removing the `n`th value from the list where n = *expr*. Return expression is a `boolean`: `true` if item is removed and `false` if `n` is greater than the number of items in the list.

Table 2-8 shows the methods for performing calculations with field values.

Table 2-8 Methods for Calculating with Field Values

Method	**Description**
<*fieldname*>.subscript(*expr*)	Applies to lists only, returning the value of the field whose subscript = *expr*. The result type is the base type of the list. If *expr* is greater than the number of items in the list, the list field will be unset, that is, it will no longer have values and `isSet` will return `false`.
<*fieldname*>.and(*expr*)	Both must be boolean. Returns `true` if both are true; `false` otherwise.
<*fieldname*>.or(*expr*)	Both must be boolean. Returns `true` if either is true; `false` otherwise.
<*fieldname*>.add(*expr*)	Both must be integers. Returns the sum of the two.
<*fieldname*>.subtract(*expr*)	Both must be integers. Returns the difference of the two.
<*fieldname*>.multiply(*expr*)	Both must be integers. Returns the product of the two.
<*fieldname*>.divide(*expr*)	Both must be integers. Returns the result of dividing *fieldname* by *expr*.

Methods for Changing Field Characteristics

Currently, there is only one such method:
<*fieldname*>`.setReadOnly`

This method takes a boolean argument; it defaults to `ReadOnly` (true). It can be invoked in any type of field and its action is to render the field read-only in the GUI. This means the user cannot modify the data in the field, although the PDL code itself may still do so.

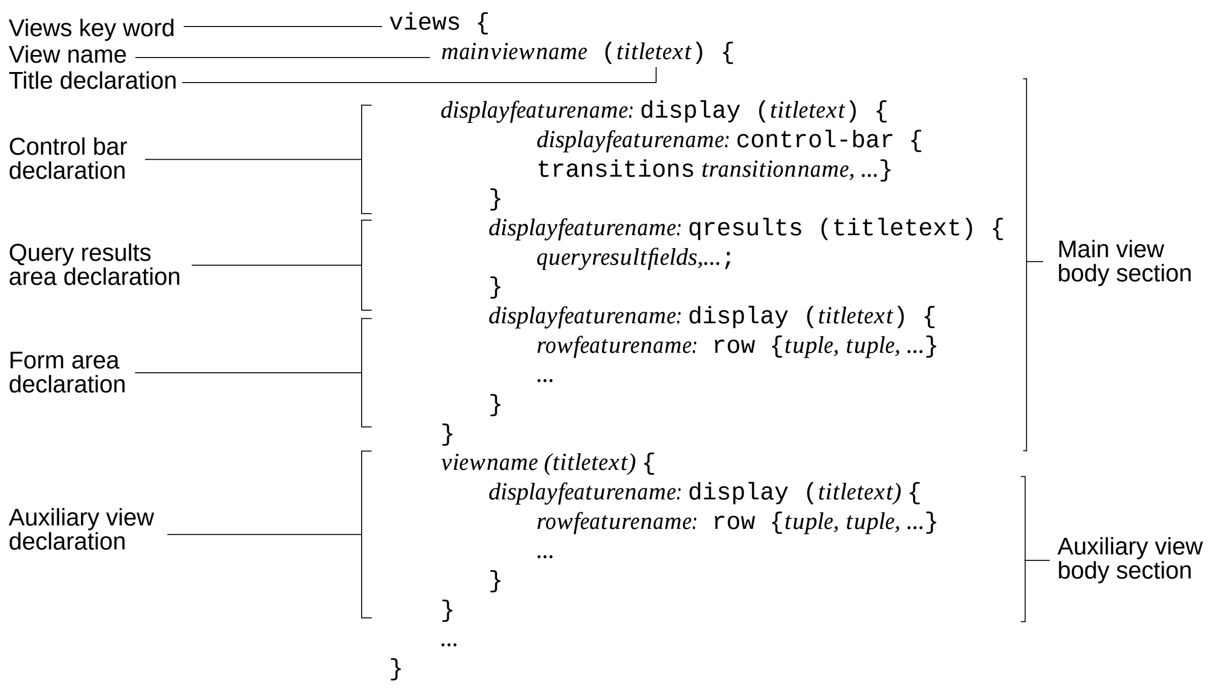
The action of this method lasts only for the current transition. Once the transition has been applied or cancelled, the field reverts to its original state.

View Declarations

The formats for Tracker windows are declared in the views section in the PDL file. You can declare any number of views to be applied to the same data in the request tracking system. You can also permit end users to customize their windows.

View Format

The views section is identified by the key word *views*. The declaration of the main view is first, followed by any auxiliary view declarations. Each view has a name, a title declaration enclosed in parentheses, and a body section enclosed by braces in which various labels, fields, and controls are declared. The general format for the views section is shown in Figure 2-5.



```
views {
    mainviewname (titletext) {
        displayfeaturename: display (titletext) {
                displayfeaturename: control-bar {
                transitions transitionname, ...}
            }
            displayfeaturename: qresults (titletext) {
                queryresultfields,...;
            }
            displayfeaturename: display (titletext) {
                rowfeaturename: row {tuple, tuple, ...}
                ...
            }
    }
    viewname (titletext) {
            displayfeaturename: display (titletext) {
                rowfeaturename: row {tuple, tuple, ...}
                ...
            }
    }
    ...
}
```

Figure 2-5 Format for the Views Section of PDL File

View Name

The main view name declared in the PDL file:

- appears in the title bar
- serves as the X11 application class name to
  - retrieve resources from each user's personal resource file (usually *~/.Xdefaults*)
  - select an application's *app-defaults* file
  - select the icon used for the application by *4Dwm*
- is used to invoke the application (after conversion to all lower case)

The auxiliary view names appear as items in the Views menu in the main view and also in their respective title bars.

View Title Section

Each view can have a title declaration, within the parentheses following the view name. If the title declaration is empty, no title will be drawn above the view. In declaring a title, you can use strings inside single quotes and field names defined in the fields declaration section. The strings and field values are assembled into a one-line title at the top of the entire view.

Fields that display in a title are not editable inside the title area. You can use white space to separate items, in which case they are positioned next to each other, or you can use one or more commas to separate them. To stretch a title so that it reaches across the window, use commas; these add padding so that all the comma-separated sections are of equal width.

View Body

The body of the view is enclosed in braces and declares the features of the window, that is, labels, fields, controls, and associated on-line help. A window is defined feature by feature from top to bottom. Tracker provides a number of key words to make the process easier.

The key word `display` lets you define an area within the window. You can specify a title, defined in the same way as the title of the entire view. You can also select resources to control both the appearance of the display as a whole and items within the display area. See “Feature Names” in the next section.

Within the view body, you can also declare the control bar, which contains the Modes menu, button controls, and the query results area for displaying request summaries matching the query criteria.

The key word `row` helps you organize fields and their labels, entered as literal strings called *tuples*, into a row. Similar to title declarations, the items in a row are specified with or without comma separators. Padding is added at the commas, so that the groups separated by commas are evenly spaced. If the row is inside a display, the first literal string in each group is treated as a label. The width of a given column is expanded according to the longest label in the display. Rows can be given names, which are used in resource selection, but do not display any title text. Tuples are discussed in “Defining a Request Form Area” later in this chapter.

User Interface Formatting

Tracker provides two mechanisms for formatting the interface:

- PDL key words for defining application-specific parts of the user interface
- predefined feature names attached to X11 widgets

Display Key Words

Tracker supplies a number of key words to let you declare the standard features of Tracker applications. These are the reserved words and their meanings in PDL files.

views {}

Designates the start of the views section. All view specifications for this application are inside the braces.

display () {}

Groups the items specified inside the braces as a defined (formatting) area. The parentheses contain the title, if any.

```
control-bar ()
  {transitions ...}
```

Displays the control bar, including the Modes menu (see Figure 2-7). The parentheses contain the title, if any. The braces contain the transition commands in the Modes menu. If you want to include all of the defined transitions, enter the key word `transitions` with no other entries. If you want a subset of the transitions, enter them explicitly after `transitions`. If you leave the braces empty, there will be no transitions; effectively, you will have a query-only application.

These conditions will cause error messages:

- invalid transition names
- duplicate transition names in the list
- more than one control bar declaration

```
qresults ()
  {index ...}
```

Displays the query results area (see Figure 2-8). The parentheses contain the title, if any. The title defaults to the field name. Insert fields to be displayed in the summary line inside the braces after the key word `index`. Data in this area is aligned in columns, and you can define the width of each column, and add a custom title to it. For example, the PDL file in the sample RTS system has this query results definition:

```
qresults() {
    index   type:3:’Type’
            $ENTITY_ID:5:’ID#’
            $STATE:20:’State’
            owner:20:’Owner’
            summary::’Summary’;
 };
```

The first four field definitions are followed by a width (number of characters) and a title string. The last column defaults to the total width permissible. Any part of the field definition left blank will use the default width and / or title. These field definitions need not be set out on individual lines; they can be listed sequentially.

An error message is displayed if there is no `qresults` declaration or if there is more than one.

In order for the users to sort on a field in the GUI, the field must be placed in the declaration for the query result area. Users can only sort on one field at a time.

row {...};

Displays items specified inside the braces in a row. Each row is divided into a set of tuples, which is a combination of fields and / or string-literal labels, typically one label and its associated field. The tuples are separated by commas. A single column contains one tuple.

For more information, refer to “Defining a Request Form Area” later in this chapter.

Feature Names

A *feature name* identifies an element in the user interface and is attached to the X11 widget that implements the display, which in turn retrieves resources. Feature names precede the PDL key words (described in the previous section) and are followed by colons. A number of useful appearance variations are provided in the standard Tracker application defaults file.

The feature names applied to views, displays, and rows are used to select resources that fine tune the appearance and behavior of the interface. Due to the hierarchical way in which X11 manages resources, a name attached to a view or display can also be used to select resources for contained display items that are nested several layers deep.

For example, if a view named *viewOne* contains an unnamed display, which in turn contains an unnamed row, which finally contains a certain field, you can specify resources for that field in the resource file or the application defaults file by the name *viewOne*, and they will be correctly applied to the field.

Specifying Application Resources

Resources specified by the application name itself generally take precedence over those specified by the application class name. Resources specified in individual users’ resource files generally take precedence over those specified in the application defaults file.

Complete details on resource selection are located in the *Xlib Programming Manual for Version 11*under “Managing User Preferences.”

A number of resources are defined in the Tracker application default file (and included in the application defaults file generated for Tracker applications). To use them, you must give the indicated name to the feature you want controlled (or to a display or row that contains it). Predefined feature names are illustrated in Table 2-9.

Table 2-9 Predefined Feature Names

Names	Description
oneRowList	Defines a list field that is one row high.
fatBox	Draws a high-visibility rectangle around the specified display area to set it off from the rest of the interface.
oneRowLongText twoRowLongText fourRowLongText eightRowLongText sixteenRowLongText	Defines a long-text field to be the specified number of rows.
threeColumnDisplay fourColumnDisplay	Defines a display area to be the specified number of columns.

For more information on feature names, refer to “Using Dates” in Chapter 6, “Advanced Design Techniques.”

Formatting Example

Figure 2-6 shows the declarations in the views section of a PDL file. The *rtsquery* application has two views: *RTSQuery* and *RTSFiles*. The main view has three customizable display areas: a control bar, a query results area, and a request form area. The auxiliary view *RTSfiles* is used to list files so it has a simple structure of three long-text fields and their labels.

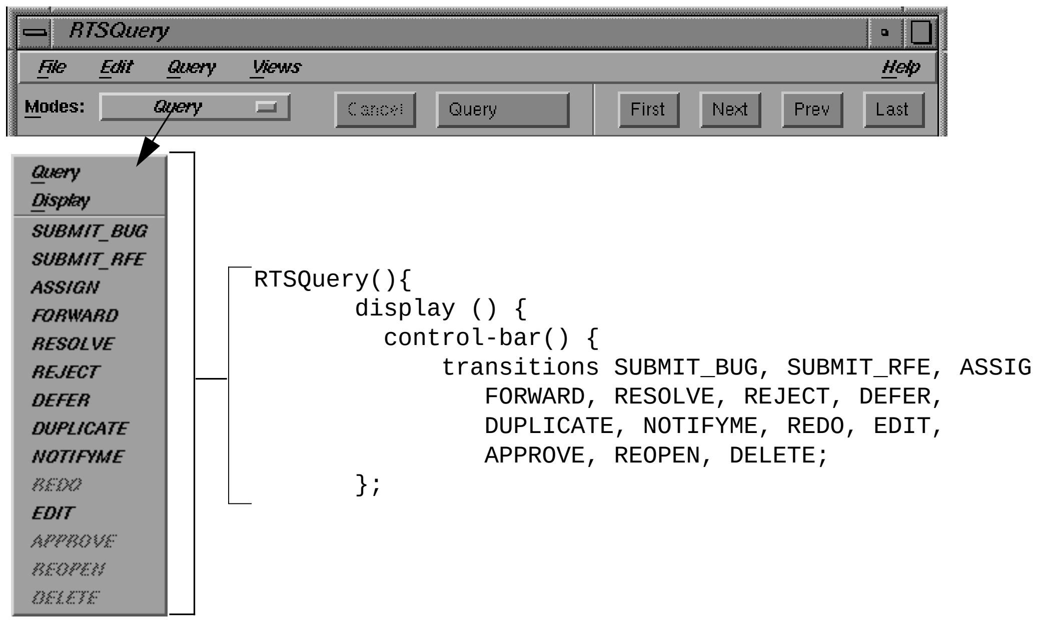
```
View label ————————————————— views {

RTSQuery view declaration —— RTSQuery(){
                                display () {
Control bar ————————————————       control-bar() {
                                      transitions SUBMIT_BUG, SUBMIT_RFE, ASSIGN,
                                         FORWARD, NOTIFYME, EDIT, DEFER, RESOLVE,
                                         REJECT, DUPLICATE, REDO, APPROVE, REOPEN,
                                         DELETE;
                                   };
Query results area declaration ——  };
                                Query_Results: qresults() {
                                   index    type ' #' $ENTITY_ID, $STATE, owner,
                                                summary;
Request form area declaration ——  };
                                Header: display() {
                                   row{   'Report #:' $ENTITY_ID,
                                          'Status:' $STATE,
                                          'Type:' type};
                                   row{   'Submitter:' submitter,
                                          'Date:' submit_date,
                                          'Recommend:' recommendation};
                                   row{   'Project:' project,
                                          'Priority:' priority,
Resource tag ————————————————             'Owner:' owner};
                                   oneRowList:
                                   row{   'System:' system,
                                          'Notify:' interested_parties,
                                          'Due Date:' due_date};
                                   row{   'Close Date:' close_date,
                                          'Reopen Date:' reopen_date,
                                          'Approver:' approver};
                                   row{   'Summary:' summary};
Resource tag ————————————————      row{   'Description:' ' '};
                                   fourRowLongText:
                                   row{   description};
Resource tag ————————————————      row{   'Resolution:' ' '};
                                   fourRowLongText:
                                   row{   resolution_description};
                                }
                             }
RTSFiles view declaration —— RTSFiles(    type ' #' $ENTITY_ID, $STATE, owner,
                                          summary) {
                                   row{'Found in:' ' ', ' '};
                                   row{found_in};
                                   row{'Resolved in:' ' ', ' '};
                                   row{resolved_in};
                                   row{'Fixed Releases:' ' ',,,,,};
                                   row{fixed_releases};
                                }
                             }
```

Figure 2-6 Typical Declarations in the Views Section of a PDL File

Defining a Control Bar

Control bars are identified by the reserved word *control-bar*. They use the reserved word *transitions* to specify the selections in the Modes menu. In the control bar, you have the ability to define which transitions are available in the Modes menu (as shown in Figure 2-7).



```
RTSQuery(){
       display () {
         control-bar() {
             transitions SUBMIT_BUG, SUBMIT_RFE, ASSIG
                FORWARD, RESOLVE, REJECT, DEFER,
                DUPLICATE, NOTIFYME, REDO, EDIT,
                APPROVE, REOPEN, DELETE;
       };
```

Figure 2-7 Portion of PDL Defining Modes Menu

Defining a Query Results Area

Query results areas are indicated by the key word *qresults()*, which is preceded by the tag for the area. Following `qresults()` is the declaration of information to appear in each line of the list, enclosed by braces.

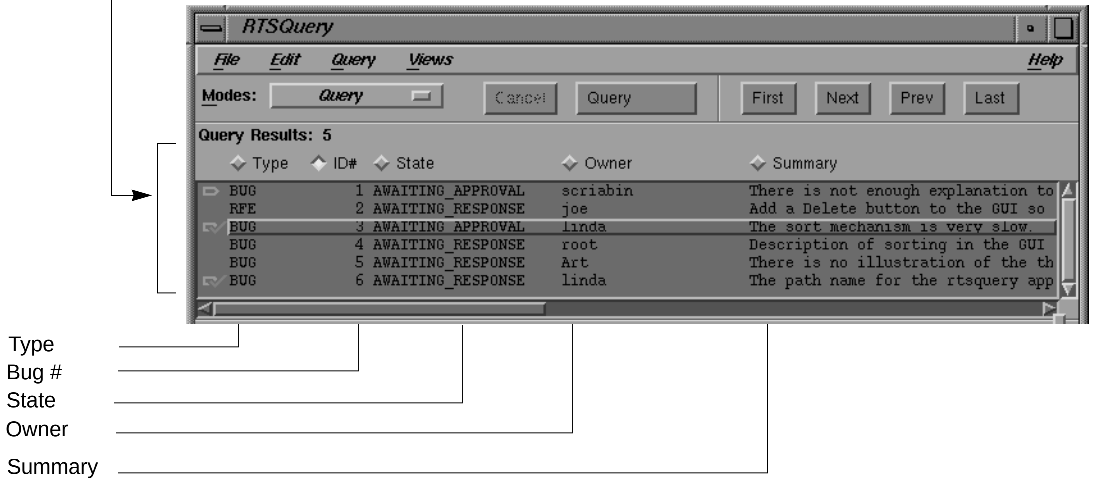
Figure 2-8 shows the part of the PDL file that declares the query results area for *rtsquery* with the corresponding display area. Notice that the declaration uses the two predefined fields $STATE and $ENTITY_ID. The numbers enclosed by colons (:) indicate the width of each column.

In the display area, numeric fields such as ID# are right-justified, and text fields are left-justified.

Format

Tracker provides two icons in the query results area. A small rectangle next to a request in the list indicates that the request has been displayed in the form area. A check mark indicates that the request has been edited (see Bug #1 in Figure 2-8).

```
RTSQuery(){
   ...
   Query_Results: qresults() {
     index   type:3:'Type'
             $ENTITY_ID:5:'ID#'
             $STATE:20:'State'
             owner:20:'Owner'
             summary::'Summary'
    };
   ...
  }
```



```
Query Results: 5
   Type   ID#  State                 Owner              Summary
   BUG      1 AWAITING_APPROVAL     scriabin           There is not enough explanation to
   RFE      2 AWAITING_RESPONSE     joe                Add a Delete button to the GUI so
   BUG      3 AWAITING_APPROVAL     linda              The sort mechanism is very slow.
   BUG      4 AWAITING_RESPONSE     root               Description of sorting in the GUI
   BUG      5 AWAITING_RESPONSE     Art                There is no illustration of the th
   BUG      6 AWAITING_RESPONSE     linda              The path name for the rtsquery app
```

Figure 2-8 Portion of PDL Defining Query Results Area

Setting Defaults

Two entries in the *app-defaults* file affect the query results area. The entry

```
*query_results.visibleItemCount:5
```

sets the number of lines to display at startup (request summaries) in the query results to five. If your end users prefer a different initial size, you can change this entry.

When you perform a query on the request database, Tracker retrieves the first 50 entities that it finds so that it doesn't need to perform another retrieval until the user scrolls past the fiftieth line. The entry

```
*QueryFetchCount: 50
```

controls the number of summaries retrieved. Retrieving more entities slows the initial query results area display but the trade-off is that you have more entities to scroll through. If performance on queries becomes a problem, you can adjust the fetch quantity.

Defining a Request Form Area

The *rtsquery* request form area uses the key word *display* to specify the items that make up the form in the GUI. The form area in the main view body section of the PDL file is identified by this key word (see Figure 2-5). Fields are declared in the row in which they appear from left to right, separated by commas.

Fields are declared in the form of tuples. Here is a description of a typical lay-out for fields and labels. However, it is not the only way, and you can experiment as much as you wish.

Each tuple contains the field name and a label. The label for the field is entered first, enclosed in single quotes, followed by the name of the field as declared at the beginning of the PDL file. Figure 2-9 shows the code in the PDL file and the resulting view window.

```
display() {
   row{'Report #:' $ENTITY_ID,
       'Status:' $STATE,
       'Type:' type,
       'Submitter:' submitter};
   row{'Date:' submit_date,
       'Recommend:' recommendation,
       'Project:' project,
       'Priority:' priority};
   oneRowList:
   row{'Owner:' owner,
       'System:' system,
       'Notify:' interested_parties,
       'Due Date:' due_date};
   row{'Close Date:' close_date,
       'Reopen Date:' reopen_date,
       'Approver:' approver,
       'Dup of:' is_duplicate_of};
   row{'Summary:' summary};
}
display() {
   row{'Description:' ' '};
   fourRowLongText:
   row{description};
}
display() {
   row{'Resolution:' ' '};
   fourRowLongText:
   row{resolution_description};
}
```

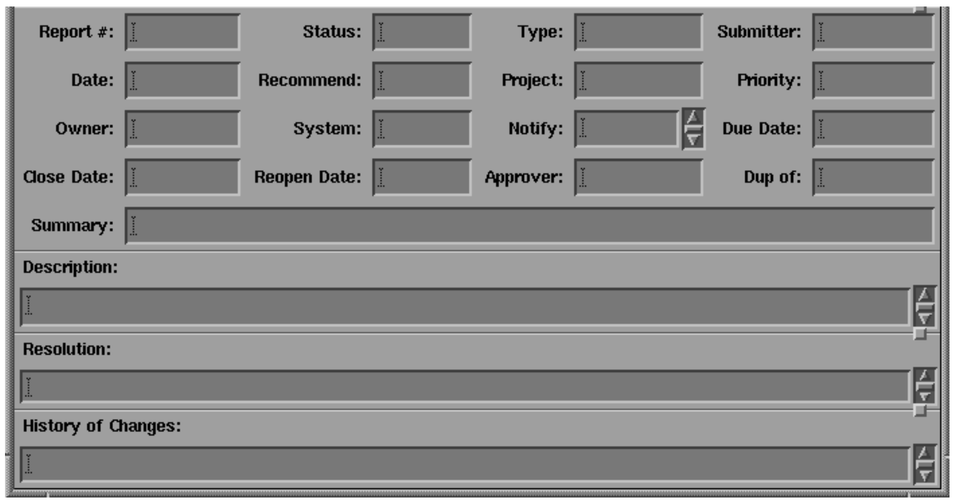


Figure 2-9 Portion of PDL Defining Request Form Area

Customizing the Request form Area

You can arrange the request form area in any way you like by using tuples in the `row` statement. If a row consists of a single tuple, that tuple will be expanded to fill the row. For example, this format

```
row{'Description' 'hello there'}
```

fills the row. If you divide the row up into several tuples, the row width is evenly divided among the tuples. For example, this format

```
row{'Description' 'hello there', , }
```

would cause the first tuple, consisting of `'Description'` and `'hello there'`, to use one third of the full row width. The other two tuples are empty, so the rest of the row will be empty.

You can use empty strings within a tuple to get a similar effect. For example,

```
row{'Found in:' ' ', ' '};
```

creates two tuples of equal width in the row. The first tuple contains the text "Found in:" plus a specified amount of space. The second tuple contains only white space. If you use this format:

```
row{'Found in:'};
```

the text fills up the row completely.

To calculate the amount of space required for each item in a tuple, consider these guidelines.

- Literal strings default to their actual width in characters, but may expand to fill available space.
- Text and text field widgets (which are used to enter field data) have a preferred width determined by the application defaults or other resources.
- Tuple spacing is affected by the lay-out of corresponding tuples in the rows above and below.

The tuple is laid out from left to right, with a single space between items. The last item is then "attached" to the rightmost edge of the tuple and the tuple is stretched to fill its allotted space in the row.

When the width of the tuple is adjusted, you may have a larger space opening up between the rightmost item and its neighboring item to the left. The amount of space created will depend on how wide the tuple is, which in turn depends on the row width and number of tuples in the row. This example shows how to use tuples to left-justify a label:

```
display() {
    row{'Resolution:' ' '};
    fourRowLongText:
    row{resolution_description};
}
```

The first row contains one tuple with two items, the label `'Resolution:'` and the space `' '`. The label will occupy a 10-character space starting from the left, and is separated from the space item by a 1-character space. The space item then expands to fill the rest of the row.

In addition to your custom settings, Tracker adjusts the width of tuples so that they form justified columns in a display. A tuple in a given column in a row is always allotted the width of the widest item in that column.

For example, in this three-column display with three rows, the width of the second column will be equivalent to the width of the `'Recommendation:'` tuple in the third row:

```
threeColumn: display() {
    row{'Report #:' $ENTITY_ID,
        'Status:' $STATE,
        'Priority:' priority};
    row{'Type:' type,
        'Submitter:' submitter,
        'Owner:' owner};
    row{'Date:' submit_date,
        'Closed:' close_date,
        'Recommendation:' recommendation};
}
```

Figure 2-10 shows how display appears on the screen.

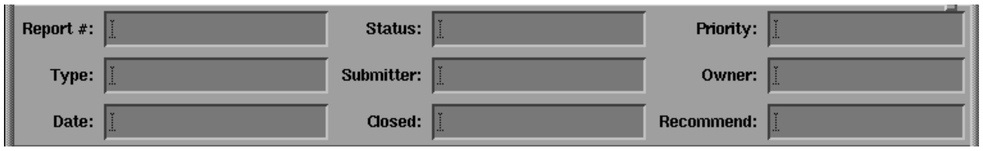


Figure 2-10 An example of a three-column display

Defining Field Pop-up Menus

When a user holds down the right mouse button over a field, a menu appears displaying the commands “Reuse,” “Revert,” “Clear,” and, optionally, “Values” as an aid to filling in the field.

The “Reuse” command reuses the last value displayed in the field.

The “Revert” command reverts to the prior value for the request.

The “Clear” command clears the field.

If the field type (as declared in the *Fields* section of the PDL file) uses the “one-of” designator, then the selection “Value” appears with a rollover menu that contains the declared set of values.

Users can select a value from the menu or type directly into the field.

You can “tear off” the rollover menu by clicking on the perforation line at the top of the menu. It then appears as a freestanding menu (see Figure 2-11).

The field declaration and view declaration for the Status field are shown in Figure 2-11, along with the resulting pop-up and rollover menus.

```
fields {
        ...
        status: one-of
          AWAITING_RESPONSE, AWAITING_APPROVAL, CLOSED, DELETED;
        ...
        }

views {
        RTSQuery () {
        ...
           row{
           ...
           'Status:' status};
```

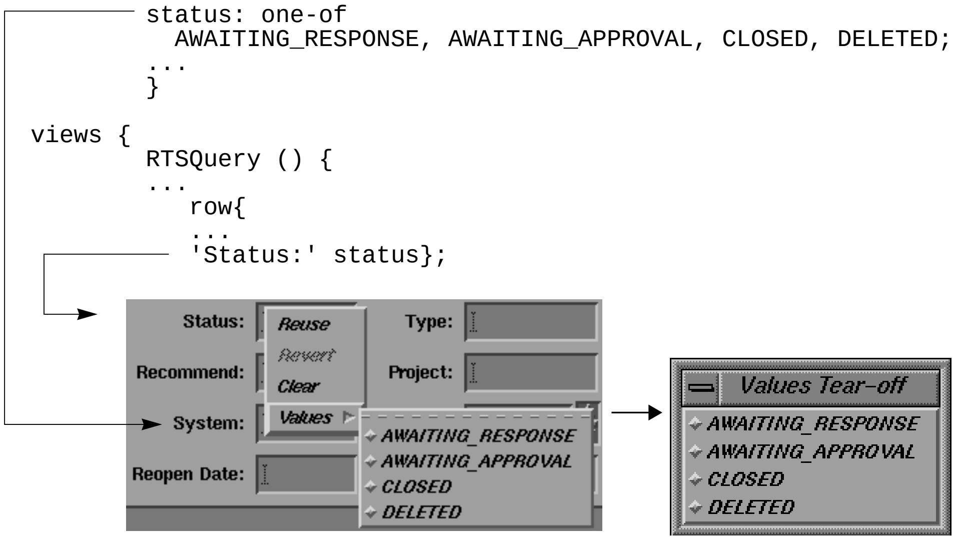


Figure 2-11 Portion of PDL Defining Status Field

Help Declarations

If you are making major modifications to the RTS applications or creating your own tracking system, you need to provide on-line help to your users. Tracker provides the same on-line help system available in all CASEVision environments. If you are unfamiliar with the operation of on-line help, please refer to the *CASEVision Environment Guide*.

On-line help is easily provided by making help declarations in the PDL files. You can create help cards for such topics as:

- the entire application
- fields as a group and individually
- transitions as a group and individually
- rules and actions individually within transitions and globally
- all and individual views
- individual display areas
- control bar
- query results area
- rows in a display

Running *tvgen* (and *rtsgen*) automatically creates a hierarchy of on-line help based on the help declarations. Users can search for help by topic or by clicking the item on the screen while in context-sensitive help mode.

Creating a Help Declaration

To create a help declaration, use this general format:

```
help {
   help-title 'fulltitle';
   short-help-title 'shorttitle'; //optional
   help-text 'helpbody';
};
```

The entry following `help-title` is the complete title for the help topic. This title appears in the help index.

If you prefer a shorter version of the title in the graphic help browser, add the `short-help-title` line with an abbreviated version of the title. The help information is entered after the key word `help-text`. All of the text must be enclosed within single quotes. By default, help text appears in proportional font; if you prefer fixed width spacing for your help text, then use the key word `fixed-width-help-text` in place of the key word `help-text`.

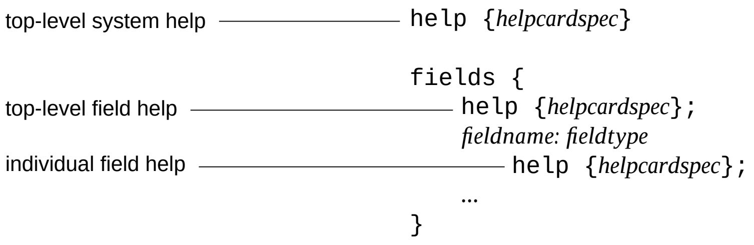
In addition, you can specify that the actual rules and actions declarations for a transition appear in a help card by using the key word `include-pdl`; this is covered in more depth in the “Transition Declarations” section.

Note: If you wish to enter a single quote (apostrophe) in help text, you must precede it with a backslash (\’).

For the sake of conciseness, the entries inside the braces following the key word `help` are presented as the single term *helpcardspec* (short for help card specification) in this guide.

Help Declaration Locations in the PDL File

The location of the help cards in the on-line help hierarchy is a function of the declarations in the PDL file. The locations for help declarations are shown in Figure 2-12 and Figure 2-13.



```
help {helpcardspec}

fields {
    help {helpcardspec};
    fieldname: fieldtype
        help {helpcardspec};
    ...
}
```

Figure 2-12 Help Locations at Beginning of PDL File

```
                                    transitions {
top-level transition help ————————————  help {helpcardspec};
                                        transitionname (priorstate => newstate) {
individual transition help ————————————————  help {helpcardspec};
                                            rules {
rules help for individual transition ——————————  help {helpcardspec};
                                                fieldname.method || fieldname.method || ...;
                                            }
                                            actions {
actions help for individual transition ————————  help {helpcardspec};
                                                fieldname.method;
                                            }
                                        }
                                        ...
                                        rules {
global rules help ———————————————————————  help {helpcardspec};
for all transitions
                                            fieldname.method || fieldname.method || ...;
                                        }
                                        actions {
global actions help —————————————————————  help {helpcardspec};
for all transitions
                                            fieldname.method;
                                        }
                                    }
                                    views {
application help ————————————————————  help {helpcardspec};
                                        viewname (titletext) {
individual view help ——————————————————————  help {helpcardspec};
                                            control-bar (titletext) {
control-bar help ——————————————————————————————  help {helpcardspec};
                                                transitions transitionname, ...
                                            }
                                            qresults (titletext) {
query results help ————————————————————————————  help {helpcardspec};
                                                index listfields,...;
                                            }
                                            display (titletext) {
display area help —————————————————————————————  help {helpcardspec};
                                                row {
row help ——————————————————————————————————————————  help {helpcardspec};
                                                    tuple, tuple, ...
                                                }
                                    ...
```

Figure 2-13 Help Locations in Transitions and Views Sections

Help Implementation Strategy

In providing on-line help, you are not just documenting isolated features—you are building a system for providing information to your users. Users get information from the on-line help system in two modes: context-sensitive mode and browsing mode. This section covers

- the help card hierarchy
- top-level help card strategy
- bottom-level help card strategy

The Help System Hierarchy

It is important to remember that on-line help has a hierarchical structure. Figure 2-14 shows a typical Tracker hierarchy; it represents the hierarchy from a user's point of view. The stacked help cards in the diagram indicate that there can be multiple help cards and subtrees at that location. The order in which the help cards appear (top to bottom and left to right) correspond to the sequence in which they are declared in the PDL files, except for the help cards that are shown in the illustration in white. These cards are actually declared elsewhere in the hierarchy, where they appear in gray; they are in the hierarchy redundantly as a convenience for users.

Top-level Help Card Strategy

In a Tracker application, you must declare the top-level system help card (at the beginning of the PDL file) and the top-level view help card (at the beginning of the view declaration section). If these are missing, you will get warning messages when *tvgen* (and *rtsgen*) is run; more importantly, your end users may get error messages if they try to use on-line help.

At the top level, it is important to make the resulting hierarchy easy for users to find what they are looking for. The top-level system help card file forms the root of the on-line help hierarchy. Below it are the top-level cards for the fields and transitions and the application help cards, which group the views.

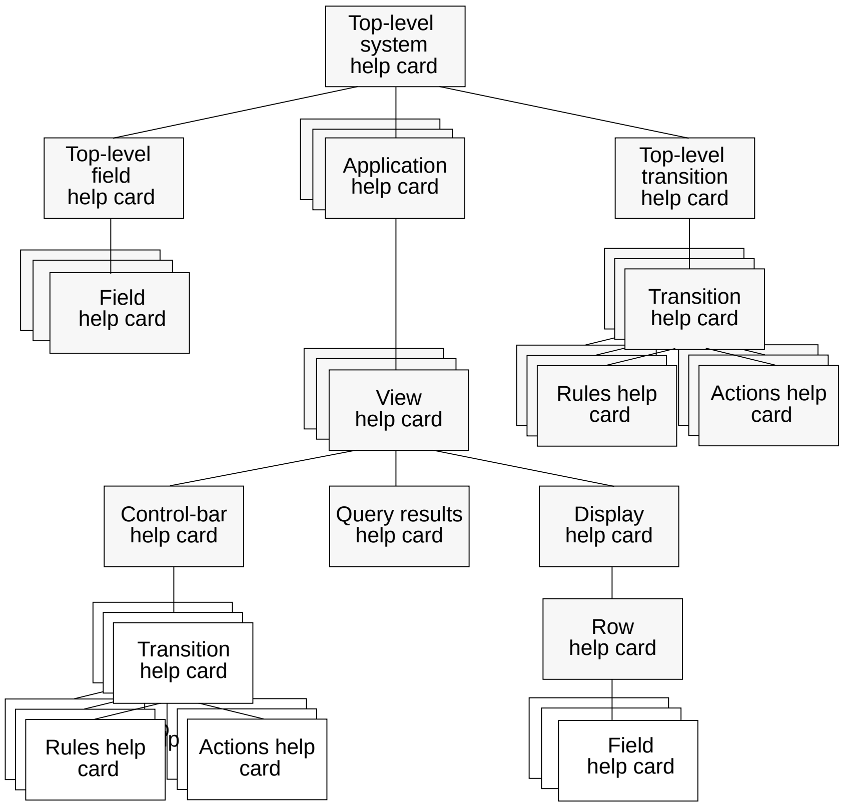


Figure 2-14 Typical Tracker On-line Help Hierarchy

Note that the help cards for individual fields and transitions (rules and actions included) can appear redundantly in the hierarchy, under the top-level field and transition cards and in the view subtrees that contain them. These help cards are actually declared in the fields section and transitions section of the PDL file respectively. These cards are shown in Figure 2-14 in white.

It is a good idea to include a help card for the control-bar in each application. The control-bar help card groups all relevant transition help cards as subtopics under the control-bar help card. If the user selects any item in the control-bar using context-sensitive help, help on the set of transitions provided by the application will be readily available in the subtopics window of the help viewer.

Bottom-level Help Card Strategy

At the bottom level, you should provide detailed information for the individual transition cards on how the particular transition relates to the overall tracking process. You can describe the rules and actions in the card in the transition help card or separately in the associated rules and actions help cards.

Use help declarations for individual fields to clarify the intended use of the field and to give suggestions for using specific values for the field. The field help declarations appear at the bottom of the view help hierarchy. They are grouped under the containing row if you declare it; otherwise, they appear under the containing display area. If you declare neither the containing row nor display area, then the field help declarations are grouped directly under the view help card.

After adding or changing help declarations in the PDL, you can review the changes by running the application (after running *tvgen* and *tvinstall*) and by selecting items in the "Help" menu. Select "Browser" in the help viewer window to examine your help hierarchy.

Caution: *tvgen* does not remove help cards from the *<databasedir>/tools/help* directory before generating new on-line help. If you rename (or remove) an application in your system, you should remove the corresponding subtree in *<databasedir>/tools/help*.

Using the Data Manipulation Language (DML)

The data manipulation language (DML) provides an interface to the Tracker database. It is similar to SQL and other fourth-generation database query languages. DML runs on top of Raima Data Manager™ from Raima Corporation. DML supports both database query and modification. DML complements the Tracker graphical user interface (GUI) by enabling more complex queries on the database, report generation, and batch database modifications.

This chapter covers these topics:

- Overview
- Specifying literal values
- *Select* statement
- *Insert* statement
- *Update* statement
- *Delete* statement
- Locking statements
- Transaction statements

Tracker Database Overview

This section explains the structure of Tracker databases, tells you how to invoke DML, relates DML to the rest of Tracker, and summarizes the basic DML statements.

Caution: DML lets you modify data and enter new records directly. To avoid the potential hazards of entering bad data, its modification features should be used sparingly and by the Tracker system administrator only. If you do change data using DML, make sure that any affected PDL files are updated accordingly.

Database Structure

Unlike many database systems, a Tracker database has no explicit schema or rigid structure, which lets it adapt readily to the changes that occur as an application matures.

Data in a Tracker database is stored in fields. Each field has a name, a value, and a type. The database field types are the same as the PDL field data types.

Fields are organized into *entities* or records. Each entity represents something from the real world, such as a bug report, project list, or project team, and the fields represent that item's properties or attributes. Every entity has a field named *$ENTITY_ID* that contains a permanent integer value that is assigned automatically by the database upon creation.

Entity Classes

A single database can hold different classes of entities. Each *entity class* (only one per PDL file) has a unique name, which is declared in the PDL file at the beginning of the field declaration section. If no declaration is made, the name defaults to *tracker_request*.

An entity can belong to only one entity class; its *$ENTITY_ID* uniquely identifies it within its class. Since entities from different classes in the same database can have the same *$ENTITY_ID* value, both the *$ENTITY_ID* and the entity class must be known to uniquely identify an entity.

Entity Fields

Although the entities within a class tend to have the same set of fields, this is not required by the Tracker database. An entity cannot have more than one field with a given name. However, different entities can have fields with the same name; fields with the same name can be of the same or different types.

As an example, the RTS database has two entity classes: one named *tracker_request* (the default) and the other named *project* (see Figure 3-1). Entities in the *tracker_request* entity class represent requests in the database. Potentially, they can use all of the fields defined in the PDL.

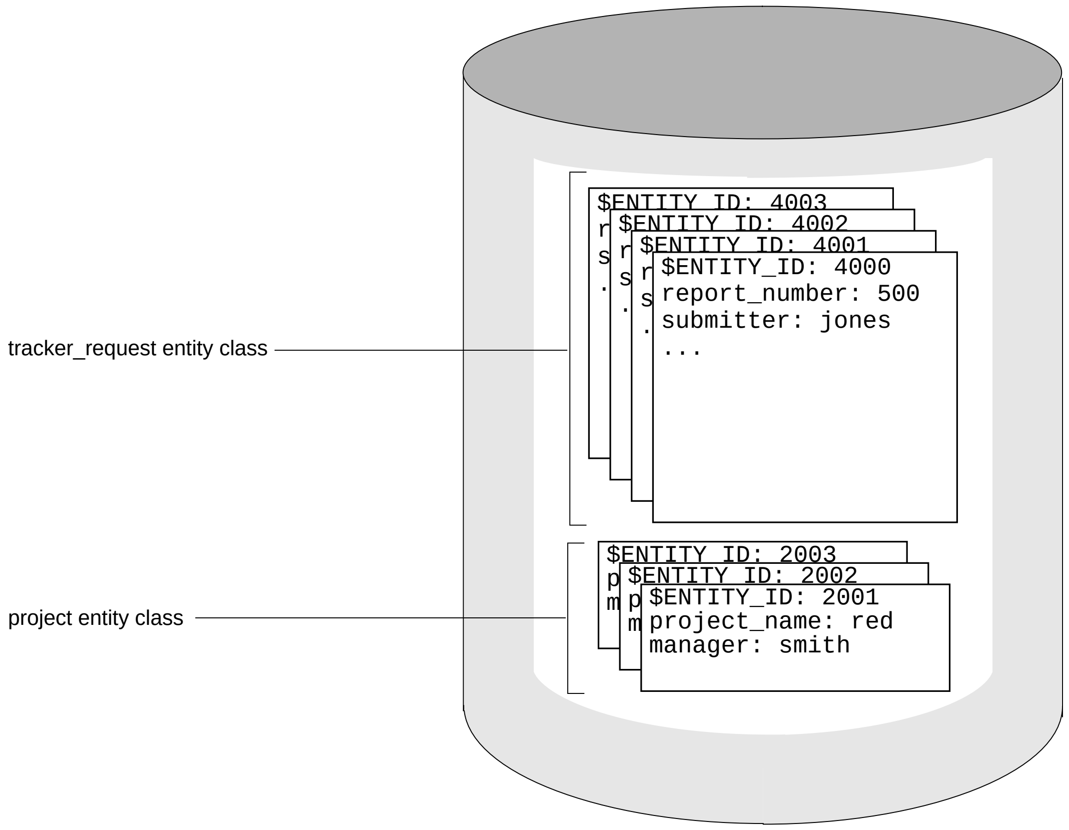


Figure 3-1 RTS Database with Entity Classes

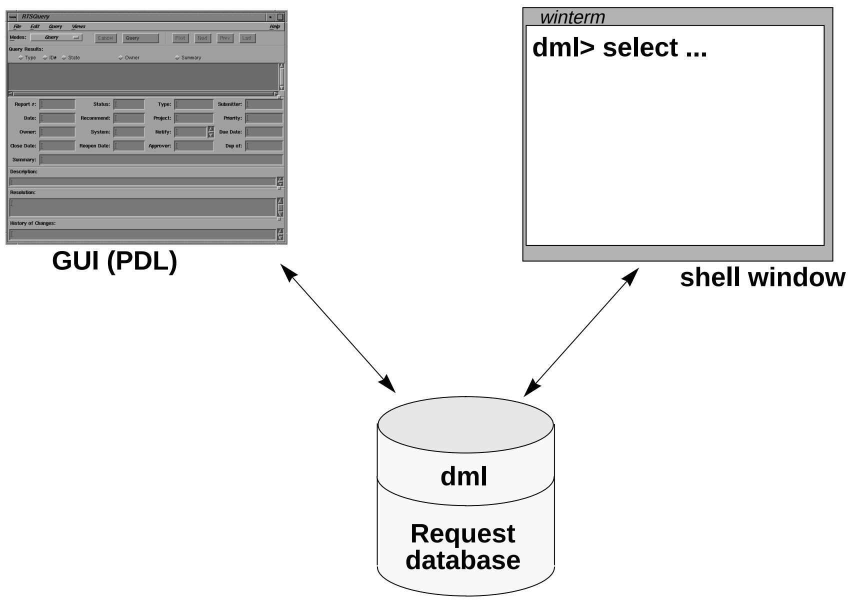
The actual information depends on which field values were entered. The entities in the *project* entity class are used to keep track of project names and managers.

When you enter a new entity into a database, you can enter a value for any field declared in the PDL file. You can also enter new fields if you have a special requirement.

Access to the Database

Queries from the graphical user interface are transmitted in DML to the database and can access only fields defined in the associated PDL file. From the DML interface, you can use PDL field definitions or you can define your own fields.

Figure 3-2 illustrates the relation of the *dml* program to the two interfaces.



```
dml> select ...
```

Figure 3-2 Relation of DML to Tracker and PDL

The dml Program

The program *dml* provides access to the Tracker database using the DML. You can use *dml* interactively to post ad-hoc queries and to modify the database, or as a script interpreter to generate reports or implement batch processing of the Tracker database.

DML Shell Commands

Two shell commands, *dmlrpt* and *dmlcount*, are provided to demonstrate the use of scripts. *dmlrpt* creates a list of entities containing specified fields. *dmlcount* counts the entities in a database containing a given field condition.

dml, *dmlrpt*, and *dmlcount* are described in more detail in the man pages.

Controlling Database Access

The database designer or system administrator can control who is able to make changes to a database through DML. This is done by means of the UNIX file permissions on a file called *Tracker.sec* in the database directory. The *Tracker.sec* file is the control point for access to the database. Its permissions, which match those on the database, provide security for the information in the database.

When a database is created, the creator can set the file permissions on *Tracker.sec* to give read and / or write permission to selected people or groups. Tracker sets the initial permissions to give ownership to the person who is running *tvgen*.

Each database has its own *Tracker.sec* file, which may allow different users access. For example, database A may have file permissions which give read / write permission to the owner, but read permission only to everyone else (`-rw-r--r--`), and database B may give read / write access to the owner and specified group, but no access to others (`-rw-rw----`). A user without read permission for a particular database cannot even start the DML interpreter program, *dml*.

Note: If you use databases created with an earlier version of Tracker (version 1.0 or 1.0.1), you can add a *Tracker.sec* file to these databases if you wish.

Required Permissions

In order to use the *select* statement, a user must have read permission in the database. To make any modifications using *update*, *insert*, or *delete*, a user must have write permission in the database.

If the *Tracker.sec* file is deleted or missing, users are given access at the default level, which is read-only. If authorized users have difficulty gaining unrestricted access, check to see that:

- a *Tracker.sec* file with the correct file permissions is in the database directory
- the server and the user's system are running the same version of Tracker

Access Through PDL

PDL already has the capability for restricting access to certain users. As a database client, the GUI has read / write access to the database. It is advisable, therefore, to continue writing PDL code to restrict user access where desired. Restricting access to the database via DML does not eliminate the need to control access via PDL.

DML Statements

This section describes how to specify literal values and documents these DML statements:

- *select*
- *insert*
- *update*
- *delete*
- *lock / unlock*
- *begin / end transaction*

General Characteristics of DML Statements

All DML statements are terminated by a semi-colon character.

It is important when using DML to specify the names of the fields exactly, since incorrect field names can be interpreted as non-existent or new. If you don't have a copy of the PDL field declarations handy, you can select a single entity designating ***** as the field list and get the proper spellings. Refer to "Select Statement" for more information.

Specifying Literal Values

For your convenience, DML provides alternatives for entering literal values in statements:

Implicit typing	Certain types of literal values can be entered in a simple format and DML will interpret them automatically.
Explicit typing	You can also enter the literals with their types to eliminate ambiguity.
Nested select statements	DML also lets you use a nested *select* statement to specify a list of integers.

Implicit Typing

DML provides implicit interpretation of certain types of literal values. Other types must be specified explicitly. To eliminate ambiguity, you can always specify types explicitly. The DML command line option `-dml` lets you turn on value typing automatically, otherwise it defaults to value typing off.

The effect of using the command is that all field values retrieved from the database are displayed with explicit type information. For example,

```
dml> value typing on;
dml> select $ENTITY_ID from tracker_request where $ENTITY_ID
<3;
$ENTITY_ID: int ’1’
```

If value typing is off, the same *select* query produces the result:

```
$ENTITY_ID: 1
```

Table 3-1 demonstrates those types that are typed implicitly.

Table 3-1 Implicit Typing Examples

Examples	Comments and Implied Type
999 -1	Numbers are assumed to be integers, of type *int*
’’ ’your text’ '3 lines long with an escaped apostrophe\' in the second line.'	Text inside single quotes is taken as short-text. Use a backslash (\) if you need to embed an apostrophe.
true false	`boolean` types
/usr/tmp/xxx saturn:/usr/lib	file types
foobar	Strings that don’t fall into any other categories are assumed to be identifiers of type *one-of*
1/20/93 Jun 10 1993 06:17PM Wed Jun 10 18:19:22 PDT 1993	date types
(1,2,3) (’a’,’b’,’c’)	Strings inside parentheses, separated by commas, are assumed to be a list of some type.These examples are respectively *list-of int* and *list-of short-text*.

Explicit Typing

Explicitly typed entries contain a field type name followed by a quoted string. The quoted string must contain a literal value. Table 3-2 shows examples of explicit typing.

Table 3-2 Explicit Typing Examples

Examples	Comments
int '99'	*E*xplicit integer.
long-text 'hello'	Explicit long-text specification. long-text cannot be implicitly specified.
date '1993'	Certain date formats are recognized only by using this form. This example would be considered an integer if not explicitly specified.
one-of 'select'	Lets you specify select as a string rather than as a DML keyword.
short-text 'Joe\'s	A backslash inside quotes lets you specify a literal apostrophe.
(one-of 'RED', one-of 'GREEN', one-of 'BLUE')	Explicitly typed literals can be used in lists.

Use explicit typing to lessen the chance of the literal being interpreted in an unexpected way. Be sure, however, that the entry inside the quotes correctly matches the specified type. Table 3-3 demonstrates the incorrect use of explicit typing.

Table 3-3 Illegal Formats

Examples	Comments
int 'not-an-integer'	The quoted value has no valid integer interpretation.
date 'not-a-date'	The quoted value has no valid date interpretation.
boolean 'maybe'	The quoted value has no valid boolean interpretation.

Select Statement

The *select* statement queries the Tracker database and returns field values. It includes an optional *order by* statement that lets you sort the items within each field in ascending or descending order.

The general form is:

```
select field-list from entity-class;
```

field-list names the Tracker fields returned by the query for each selected entity. This form selects all entities of class *entity-class*. The optional additions to this statement, shown in square brackets, are:

```
select field-list from entity-class
     [order by field1 descending, field2, ...];
```

```
select field-list from entity-class [where condition];
```

The items in the fields listed in *order by* without a modifier will be sorted in the default order, which is ascending. To sort in descending order, add the keyword `descending` after the field name. The condition expression in the second form determines which entities are selected by the query. For example, the statement:

```
select $ENTITY_ID, Customer, Submit_date from bug where
Engineers_pri = 1;
order by Submit_date descending;
```

returns a table containing the *$ENTITY_ID*, *Customer*, and *Submit_date* fields for all bugs with *Engineers_pri* equal to 1, and listed according to date of submission. For example:

```
$ENTITY_ID: 40274
customer: acme
submit_date:Wed Jun 10 21:57:33 1992

$ENTITY_ID: 39567
customer: xyzco
submit_date: Fri May 29 10:09:00 1992
```

You can substitute an asterisk (*) for *field-list* in a *select* statement, which then retrieves all fields for qualifying entities:

```
select * from bug where Engineers_pri = 1;
```

Typically, entities do not have data in all available fields. When performing a query, the DML retrieves those fields named in the field list that do exist and ignores the entity's empty fields.

The *condition* expression can contain multiple field comparisons combined using the *and*, *or*, and *not* operators and is not limited to equality comparisons.

For example,

```
select $ENTITY_ID, Customer, Submit_date from bug where
(Engineers_pri > 1 and Type = BUG) or
(Engineers_pri < 4 and Type = RFE) or
(Type <> BUG and Type <> RFE and Priority = P1);
```

Table 3-4 lists the available comparison operators.

Table 3-4 Comparison Operators

Operator	Name	Applicable Types
=	equal to	All field types
<>	not equal to	
<	less than	int, short-text, long-text
<=	less than or equal to	
>	greater than	
>=	greater than or equal to	
match	regular expression match	long-text, journal fields
contains	list contains element	list-of fields
contains any	list contains any from list	
contains only	list contains only from list	
= null	unset or non-existent test	All field types
<> null	set or existing test	

The following example further illustrates comparison operators:

```
select $ENTITY_ID from bug where
Engineers_pri = null and  //unset fields match
Type = [BUG, TAKEN] and   //Type=BUG or Type=TAKEN
Severity = [1, 5:10];     //Severity=1 or is
                          //between 5 and 10
```

Value ranges take the form

[..., *value*:*value*, ...]

They are recognized only within value lists, that is, inside square brackets ([]). For example,

```
Severity = 5:10
```

is not legal; rather, you must use the following form:

```
Severity = [5:10]
```

The *match* comparison operator allows regular expression matching in text fields. The regular expression must be supplied as a quoted string literal in the form described by the *regcmp(3X)* man page. For example, the query:

```
select owner from bug where summary match '[Ww]indow';
```

retrieves all entities whose *summary* fields contain the word *window* or *Window.*

The *contains* operator determines if a *list-of* field contains a specified list element. For example, the query:

```
select * from project where engineers contains 'billy';
```

retrieves all project entities where the *engineers* field includes *billy.*

The *contains any* operator is similar to *contains*; it is used to specify multiple list elements to be contained within the *list-of* field. Thus, the supplied literal value must be a list itself. It is equivalent to *Or*ed *contains* statements. For example,

```
select * from project where engineers contains any
('billy', 'bob');
```

is equivalent to:

```
select * from project where engineers contains 'billy' or
engineers contains 'bob';
```

The *contains only* operator determines if the *list-of* field contains some subset of elements from the supplied list. If a field contains any elements not found in the supplied list, the entity is not selected. For example, the following statement:

```
select * from project where engineers contains only
('fred', 'bob');
```

will select only entities whose *engineers* field is one of the following (ignoring duplicate elements):

```
()            // empty list
('fred')
('bob')
('fred', 'bob')
('bob', 'fred')
```

Both *contains any* and *contains only* can substitute a nested *select* statement for a list. The nested *select* builds an integer list from the *$ENTITY_ID* fields of the selected entities and uses that list in the enclosing expression.

For example, consider the clause

```
... where project_ids contains any (select $ENTITY_ID from
projects where name match ’TV’;
```

The *$ENTITY_ID* values for those projects matching the string `TV` are substituted into the *where* clause.

Nested *Select* Statements

Nested *select* statements offer an alternative to specifying lists of integers in DML statements. The *select* statement return value must be of type *list-of int*.

In evaluating a nested *select*, DML first executes the nested *select* statement, then builds an integer list from the *$ENTITY_ID* fields of the selected entities, and uses the list in the enclosing expression.

Currently, only the *$ENTITY_ID* fields of the selected entities can be used in the enclosing expression. Even if the field list in the nested *select* selects another field or fields, the *$ENTITY_ID* fields always construct the list that evaluates the enclosing expression.

For example, consider the clause

```
... where project_ids = (select $ENTITY_ID from projects
where name match ’TV’;
```

The *$ENTITY_ID* values for those projects matching the string `TV` are substituted into the where clause. The *where* clause is true if the variable *project_ids* equals the list of qualifying projects.

Insert Statement

The *insert* statement is used to add new entities to the database. The two general forms are:

`insert into` *entity-class* `set` *field1* `=` *value1*`, ...` *fieldN* `=` *valueN*`;`

`insert into` *entity-class* `set` *field1* `=` *value1*`, ...` *fieldN* `=` *valueN*
`where` *condition*`;`

Each *field* = *value* pair creates a field in the database and assigns it a value. The *insert* statement automatically creates a *$ENTITY_ID* field and assigns it a unique integer value within the class; it cannot appear explicitly in the list of field assignments. For example, consider the statement:

```
insert into bug set Id = 45799, System = SCR,
Customer = 'John Doe, Inc.', Description = long-text
'Description with embedded newlines.';
```

It creates a new entity with five fields: the four specified explicitly and the *$ENTITY_ID* field.

The second form of the *insert* statement adds a new entity to the database if no entity matching the *where* clause already exists. If one or more entities matching the *where* clause do exist, they are updated as if an *update* statement had been executed and no new entity is created.

Update Statement

The *update* statement is used to modify existing entities. The general form is:

`update` *entity-class* `set` *field1* `=` *value1*`, ...` *fieldN* `=` *value1*
`where` *condition*`;`

Each field assignment in the list is applied to all existing entities selected by the *condition* expression. The *condition* expression is identical to that described for the *select* statement. For example, the statement:

```
update bug set Engineers_pri = 3
where Engineers_pri = 2;
```

modifies all bugs with *Engineers_pri* equal to 2, changing the *Engineers_pri* field to 3.

Delete Statement

The *delete* statement removes entities and their fields from the database. The general form is:

```
delete entity-class where condition;
```

The *where* clause holds the condition necessary to delete the request from the database. Use the *delete* statement with caution so that you don't inadvertently remove good data.

Locking Statements

The *lock* and *unlock* statements obtain shared, non-exclusive locks on the specified entities. A locked entity can be modified only by the holder of the lock, although other users can read its field values. Any attempt to update or delete a locked entity will generate an error from the DML processor.

The general form of the statement is:

```
lock entity_class1 where condition1
[, entity_class2 where condition2 ...];
```

or

```
unlock entity_class1 where condition1
[, entity_class2 where condition2 ...];
```

The Tracker GUI automatically locks and unlocks entities as needed to guarantee that the edits made by the user can be committed to the database. All locks are released when the holder disconnects from a Tracker database.

Transaction Statements

The *begin* and *end transaction* statements are used to combine a series of database modifications into an atomic operation; if any part of a transaction cannot be performed, then none of it is performed. They are used as follows:

```
begin transaction;
    dml statement;
    dml statement;
    ...
end transaction;
```

You cannot nest transactions.

The use of transactions can have a dramatic effect on the performance of certain database operations. For example, in creating a script to import a large number of entities, say more than 100, performance can be improved by grouping the *insert* statements into transactions with several *insert* statements per transaction. Without grouping into explicit transactions, each statement that modifies the database is treated as a transaction. By reducing the total number of transactions through grouping, the resources needed to accomplish a large task can be reduced.

Chapter 4

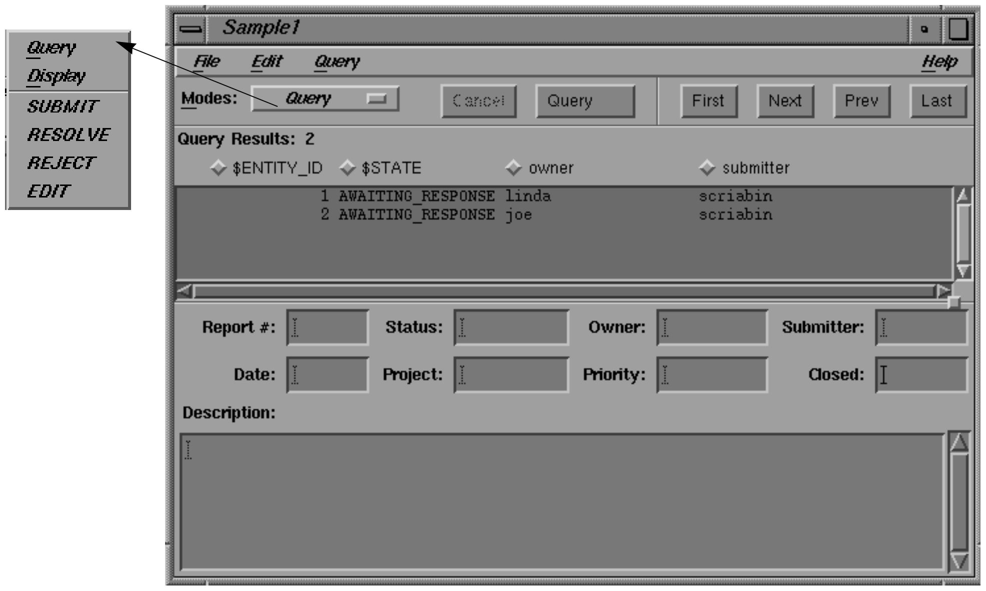
Tutorial—A Basic Tracking System

This chapter provides a tutorial based on *sample1*, a basic tracking system, and covers these topics:

- Analyzing the *sample1* PDL file
- Using *tvgen* and *tvinstall* to generate *sample1*
- Expanding the *sample1* application

Analyzing the *sample1* PDL File

The *sample1* tracking system is a useful tool that teaches you how to build your own tracking system. The user interface for *sample1* is shown in Figure 4-1.



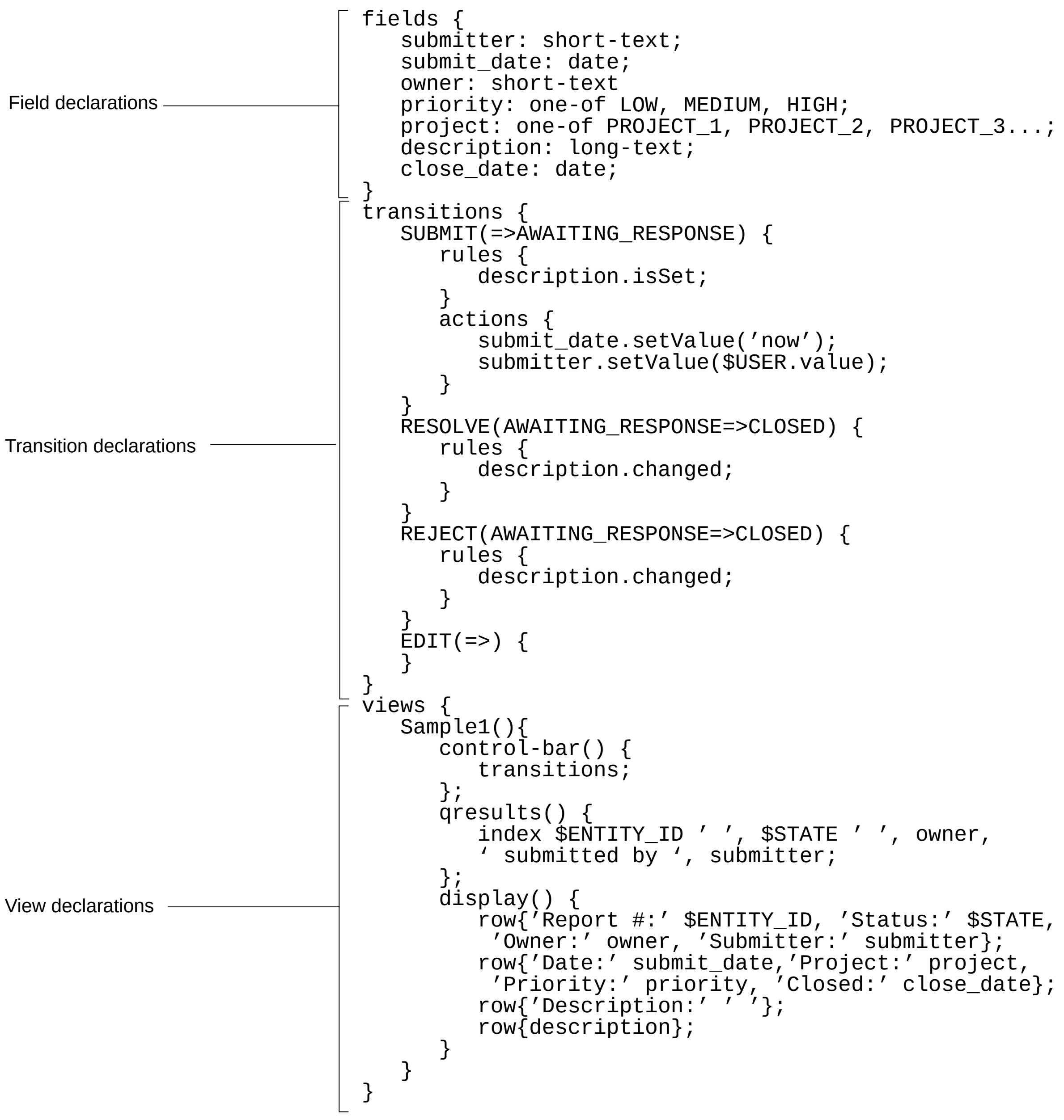
```
          1 AWAITING_RESPONSE linda                scriabin
          2 AWAITING_RESPONSE joe                  scriabin
```

Figure 4-1 sample1 User Interface

sample1 PDL File

The PDL file *sample1.pdl* is located in */usr/Tracker/samples* and is listed in Figure 4-2.

Like all PDL files, *sample1* has sections for field declarations, transition declarations, and view declarations. *sample1* is a simple application, so there are no auxiliary views.



```
fields {
   submitter: short-text;
   submit_date: date;
   owner: short-text
   priority: one-of LOW, MEDIUM, HIGH;
   project: one-of PROJECT_1, PROJECT_2, PROJECT_3...;
   description: long-text;
   close_date: date;
}
transitions {
   SUBMIT(=>AWAITING_RESPONSE) {
      rules {
         description.isSet;
      }
      actions {
         submit_date.setValue('now');
         submitter.setValue($USER.value);
      }
   }
   RESOLVE(AWAITING_RESPONSE=>CLOSED) {
      rules {
         description.changed;
      }
   }
   REJECT(AWAITING_RESPONSE=>CLOSED) {
      rules {
         description.changed;
      }
   }
   EDIT(=>) {
   }
}
views {
   Sample1(){
      control-bar() {
         transitions;
      };
      qresults() {
         index $ENTITY_ID ' ', $STATE ' ', owner,
         ' submitted by ', submitter;
      };
      display() {
         row{'Report #:' $ENTITY_ID, 'Status:' $STATE,
          'Owner:' owner, 'Submitter:' submitter};
         row{'Date:' submit_date,'Project:' project,
          'Priority:' priority, 'Closed:' close_date};
         row{'Description:' ' '};
         row{description};
      }
   }
}
```

Figure 4-2 *sample1* PDL File

Let’s analyze the sample1 PDL file, piece by piece. For details on the meaning of each term and its use, refer to Chapter 2, “Using the Process Description Language (PDL).”

sample1 Field Declarations

The field declaration follows:

```
fields {
   submitter: short-text;
   submit_date: date;
   owner: short-text
   priority: one-of LOW, MEDIUM, HIGH;
   project: one-of PROJECT_1, PROJECT_2, PROJECT_3...;
   description: long-text;
   close_date: date;
}
```

The *sample1* PDL file has seven field declarations:

submitter	is a *short-text* field; it lets users enter any text with no restrictions except size (one line only).
submit_date	is a *date* type fields; Tracker ensures that the entry is in a proper date format.
close_date	is a *date* type field.
owner	is a *short-text* field like *submitter.*
priority	is *one-of* field, limiting the user to the three selections: LOW, MEDIUM, and HIGH.
project	is also a *one-o*f field, but the ellipsis at the end makes it an open rather than closed enumeration. Thus, users can select one of the three selections or enter their own.
description	is a *long-text* field, thus permitting multiple lines of text.

sample1 Transition Declarations

The transition declarations follow the field declarations in PDL files. Each transition declaration requires a state change declaration, which can be simply the transition operator in a declaration (=>), indicating no state change with any current state valid. Rules and actions are optional.

The transition declarations for *sample1* follow:

```
transitions {
   SUBMIT(=>AWAITING_RESPONSE) {
      rules {
         description.isSet;
      }
      actions {
         submit_date.setValue('now');
         submitter.setValue($USER.value);
      }
   }
   RESOLVE(AWAITING_RESPONSE=>CLOSED) {
      rules {
         description.changed;
      }
   }
   REJECT(AWAITING_RESPONSE=>CLOSED) {
      rules {
         description.changed;
      }
   }
   EDIT(=>) {
   }
}
```

sample1 has four transitions: SUBMIT, RESOLVE, REJECT, and EDIT.

SUBMIT takes a request from a nonexistent state to AWAITING_RESPONSE. Its only rule is that there must be an entry in the *description* field. This is accomplished through the *isSet* method:

```
description.isSet;
```

SUBMIT has two actions:

The declaration

```
submit_date.setValue('now');
```

has the effect of setting the *submit_date* field to the value of '*now*', which is replaced by the current date and time.

The action declaration

```
submitter.setValue($USER.value);
```

forces the *submitter* field to the current user.

RESOLVE and REJECT have similar declarations. They both take a request from AWAITING_RESPONSE to CLOSED. Both incorporate one rule; the description must change. Its purpose is to cause the user to provide an explanation when resolving or rejecting the request.

The EDIT transition is open-ended in nature. The state does not change and EDIT can be applied to requests in any state. It has no rules or actions. You can declare no rules or actions with empty braces with or without the key words. In the example we simply use empty braces after the state declaration.

sample1 View Declaration

The view declaration for *sample1* is as follows:

```
views {
   Sample1(){
      control-bar() {
         transitions;
      };
      qresults() {
         index $ENTITY_ID ’ ’, $STATE ’ ’, owner,
         ‘ submitted by ‘, submitter;
      };
      display() {
          row{’Report #:’ $ENTITY_ID, ’Status:’ $STATE,
           ’Owner:’ owner, ’Submitter:’ submitter};
          row{’Date:’ submit_date,’Project:’ project,
           ’Priority:’ priority, ’Closed:’ close_date};
          row{’Description:’ ’ ’};
          row{description};
      }
   }
}
```

The example has one view called *Sample1* that has declarations for a standard control bar, query results area, and request form area.

The control bar is specified to use all four transitions, since the `transitions` key word is used with no transition names specified.

The query results area sets up an index of the request number (equal to the *$ENTITY_ID* variable), the state (*$STATE*), owner, and submitter. In the GUI, the area is divided into four columns, each with a title indicating the type of data it displays. The data in text columns is left-justified and vertically aligned. Data in numeric columns is right-justified.

The first two rows have four fields, plus a label for each one. The first two fields, *Report #* and *Status*, are predefined. The next six fields use field declarations from the beginning of the PDL file. The string `'Description:'` is used as a label, padded with trailing white space (‘ ‘), as described in the section on tuples (see “Defining a Request Form Area” in Chapter 2).

Generating a Tracker Application

Before you can run *sample1*, use *tvgen* to compile it and then use *tvinstall* to create the links from the public directories to the application directories. Here are the steps for generating *sample*:

1. Become super-user.
2. Create a test directory and copy the PDL file *sample1.pdl* from the */usr/Tracker/samples* directory to your test directory.
3. From any directory, type:

   `tvgen` *<yourTestDirPath>*/**`db`** *<yourTestDirPath>*/**`sample1.pdl`**

   where the expression <*yourTestDirPath*>is the absolute path to your test directory. (If *sample1.pdl* is in your database directory, you can use a relative path.)

   Running *tvgen* creates a script called *sample1* in */yourTestDirPath/db/tools/bin*, a number of database files in */yourTestDirPath/db*, help files in *<yourTestDirPath>/db/tools/help*, and an *app-defaults* file called *Sample1* in *<yourTestDirPath>/db/tools/lib/X11*.
4. Then type:

   `tvinstall` *<yourTestDirPath>* **`/db/tools`**

As before, <*yourTestDirPath*> must be an absolute path. Running *tvinstall* has the following effects:

- All directories and executable files in <*yourTestDirPath*>*/db/tools/bin* are linked into */usr/local/bin*.
- All directories and files in <*yourTestDirPath*>*/lib* are linked into */usr/local/lib*.
- All files in <*yourTestDirPath*>*/db/tools/lib/X11/app-defaults* are linked into */usr/lib/X11/app-defaults*.
- All files in <*yourTestDirPath*>*/db/tools/lib/images* are linked into */usr/lib/images*.
- The files in<*yourTestDirPath*>*/db/tools/help* are linked into */usr/lib/onlineHelp*.
- The on-line help database is rescanned and any currently running help server is killed.

5. To run *sample1*, at the shell prompt, type:

   ```
   /usr/local/bin/sample1
   ```

   You can add */usr/local/bin* to your path so that in future you need only type *sample1* to run the application.

Note: The end users must `rcp` or NFS-mount the tools directory (*...db/tools*) onto their own systems. They then install the default Tracker subsystems on their systems and run *tvinstall* to set up the required links from the tools directory to their systems.

Expanding the *sample1* Application

The *sample1* example assumes a simple tracking process in which the owner closes his or her own requests. This section introduces the concept of an approval authority, which adds a level of complexity to the application by adding another transition state, APPROVE.

The two existing intermediate states are AWAITING_RESPONSE and CLOSED. When a third intermediate state, AWAITING_APPROVAL, is added, the other transitions may also have to be rerouted.

To incorporate an approver, follow these steps:

- declare a field named *Approver*
- add the *Approver* field to the main view
- add an intermediate state called AWAITING_APPROVAL and change the affected transitions accordingly
- add a new transition named APPROVE

These steps are detailed in the following sections. You can edit your *sample1* PDL file accordingly.

Adding a Field Declaration

To add a field declaration, you:

- specify an internal field name
- select a type for it
- edit the field declarations section of the PDL file.

A complete list of field types is provided in Table 2-1 in Chapter 2. In this example, we add a field named *approver* of type *short-text*, (see Figure 4-3).

```
fields {
   submitter: short-text;
   submit_date: date;
   owner: short-text
   priority: one-of LOW, MEDIUM, HIGH;
   project: one-of PROJECT_1, PROJECT_2, PROJECT_3...;
   description: long-text;
   close_date: date;
   approver: short-text;
}
```

New field

Figure 4-3 Adding a Field Declaration

Adding a Field to a View

Once you have defined a field, you need to add it to the user interface, by editing the view declarations portion of the PDL file.

In this example, we add a tuple that comprises the *approver* field and the “Approver” label. There are currently eight short fields in two groups of four, plus the larger description field.

To keep things symmetrical, change the view to three rows of three fields each, adding the approver field last (see Figure 4-4).

```
views {
   Sample1(){
      control-bar() {
         transitions;
      };
      qresults() {
         index $ENTITY_ID ’ ’, $STATE ’ ’, owner,
         ‘ submitted by ‘, submitter;
      };
      display() {
         row{’Report #:’ $ENTITY_ID, ’Status:’ $STATE,
             ’Owner:’ owner};
         row{’Submitter:’ submitter
             ’Date:’ submit_date,’Project:’ project];
         row{’Priority:’ priority, ’Closed:’ close_date
             ’Approver:’ approver;
         row{’Description:’ ’ ’};
         row{description};
      }
   }
}
```

New tuple

Figure 4-4 Changing a View Declaration

Adding a State

Adding a new state is tricky because it actually changes the process. All transitions that enter or leave that state have to be changed accordingly. NEED MORE EXPLANATION

Figure 4-5 shows the state transitions of a request as *sample1* was originally designed.

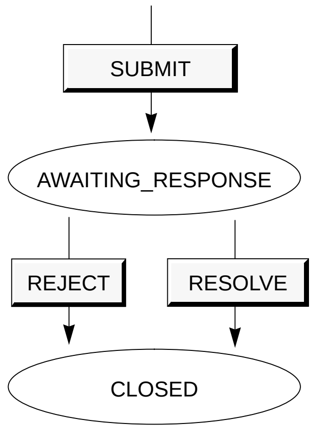


Figure 4-5 Original State Transitions of a Request in *sample1*

To implement the notion of an approval authority, you need to add an intermediate state called AWAITING_APPROVAL after the owner is through with the request, and prior to its being closed. This is shown in Figure 4-6.

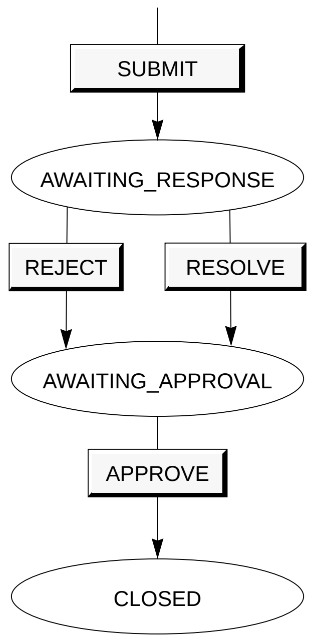


Figure 4-6 State Transitions after Adding State

As a result, you need to change the RESOLVE and REJECT transitions so that instead of going from AWAITING_RESPONSE to CLOSED, they go to AWAITING_APPROVAL. The transition declarations are shown in Figure 4-7.

Change CLOSED to AWAITING_APPROVAL

Change CLOSED to AWAITING_APPROVAL

```
transitions {
    SUBMIT(=>AWAITING_RESPONSE) {
       rules {
          description.isSet;
       }
       actions {
          submit_date.setValue('now');
          submitter.setValue($USER.value);
       }
    }
    RESOLVE(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
       rules {
          description.changed;
       }
    }
    REJECT(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
       rules {
          description.changed;
       }
    EDIT(=>) {
    }
}
```

Figure 4-7 Transition Declarations after New State

Adding a Transition

You now need to create a new transition called APPROVE that takes a request from the AWAITING_APPROVAL state to CLOSED. To set a condition requiring the approver to add an explanation to the description for closing the request, add the rule:

```
description.changed;
```

You can also add an action:

```
close_date.setValue('now');
```

This sets the closed date to the date on which the APPROVE transition is performed. Figure 4-8 shows the transitions section after you have added the new rule and action.

```
transitions {
   SUBMIT(=>AWAITING_RESPONSE) {
      rules {
         description.isSet;
      }
      actions {
         submit_date.setValue('now');
         submitter.setValue($USER.value);
      }
   }
   RESOLVE(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
      rules {
         description.changed;
      }
   }
   REJECT(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
      rules {
         description.changed;
      }
   APPROVE(AWAITING_APPROVAL=>CLOSED) {
      rules {
         description.changed;
      }
      actions {
         close_date.setValue('now');
      }
   }
   EDIT(=>) {
   }
}
```

Add APPROVE transition declaration

Figure 4-8 Transition Declarations after New Transition

Installing RTS Applications

This chapter explains how to install the RTS applications and describes the basic customization.

Procedures for Installing RTS Applications

If you are installing RTS applications, you typically follow these steps.

1. Install the Tracker software (see Step One).
2. As superuser, run *rtsgen* by typing:

   `/usr/Tracker/RTS/rtsgen` *dbdir*

   See Step Two.
3. As superuser, run *tvinstall* to perform the necessary links:

   `tvinstall` *<dbdir>/tools*

   See Step Three.
4. As superuser, run *rtsquery* by typing:

   `/usr/local/bin/rtsquery`

   See Step Four.
5. Edit the *projects.h* file (see Step Five).
6. Edit the *Tracker.pdl* file (see Step Six).
7. Rerun *rtsgen*, *tvinstall*, and the applications to see the effects of your changes (see Step Seven).

Step One

The installation procedures are covered in depth in the *CASEVision/Tracker Release Notes* and the *IRIS® Software Installation Guide*. You will need to install all of the Tracker subsystems, the CASEVision environment, and ToolTalk. The system that runs the database server needs the *Tracker.sw.designer*, *Tracker.sw.designerLinks*, and *Tracker.sw.rts* subsystems as well.

After you have installed the Tracker software, you will have the following PDL files in */usr/Tracker/RTS*:

- *Tracker.pdl*
- *rtsapprove.pdl*
- *rtssubmit.pdl*
- *rtsrespond.pdl*
- *projects.h*

The *Tracker.pdl* file specifies the main application *rtsquery*; the others specify the supplementary applications corresponding to their names. The *projects.h* file holds project names and managers and is included by the *Tracker.pdl* file.

The installation process also installs the script file *rtsgen* in */usr/Tracker/RTS*. *rtsgen* generates the RTS applications, based on the PDL files.

Step Two

The expression *dbdir* is the absolute path to your database directory. You cannot use a relative path. If *dbdir* is not specified, then the files are stored in */usr/Tracker/db*.

rtsgen allows these qualifiers:

- **`-v`** for verbose output
- **`-f`** for forced override of existing files
- **`-n`** for echo with no execution
- **`-h`** for help, that is, the usage note
- **`-d`** for debug mode, useful if you change the *rtsgen* script

The specified directory now contains the *Tracker.pdl* file, the other PDL files, and the *projects.h* file; use these versions for subsequent modifications. This directory also contains the database files, which are named *Tracker* with various extensions appended.

The script files *rtsquery, rtssubmit, rtsrespond,* and *rtsapprove* are stored in *<dbdir>/tools/bin*. The help files are stored in *<dbdir>/tools/help*. The four *app-defaults* files *Rtsapprove, Rtsquery, Rtsrespond*, and *Rtssubmit* are stored in *<dbdir>/tools/lib/X11/app-defaults*.

Step Three

tvinstall creates these links:

- *<dbdir>/tools/bin => /usr/local/bin*
- *<dbdir>/tools/lib => /usr/local/lib*
- *<dbdir>/tools/lib/X11/app-defaults => /usr/lib/X11/app-defaults*
- *<dbdir>/tools/lib/images => /usr/lib/images*
- *<dbdir>/tools/help => /usr/lib/onlineHelp*

In addition, the on-line help database is rescanned and any currently running help server is killed.

Step Four

Note: You can add */usr/local/bin* to your directory path so that in future you need only type **`rtsquery`** to run the application.

You should familiarize yourself with *rtsquery* and the supplementary applications to see how the RTS applications accommodate your needs. For additional information on their operation, refer to *CASEVision/Tracker User's Guide, RTS Applications*.

Step Five

The *projects.h* file contains placeholder values for projects and their associated managers. The actual file follows:

```
//
// project.h
//
// This file is included by Tracker.pdl, the RTS PDL.
// Replace the sample data below with projects you wish
// to track. Make sure to separate the project name and
// project manager on each line with a comma and '//'
// as is done in the sample data.

// Project Name Project Manager

 PROJECT_1, // manager1
 PROJECT_2, // manager2

// No comma on the last line!
 PROJECT_3 // manager3
```

Replace the `PROJECT_*` and `manager*` placeholders with values appropriate to your organization. Use commas after the project names except for the last one. Use a double slash (//) to separate the manager's name from the project name. The manager's name is not used in the PDL file but is stored in the database when you run *rtsgen*. Here is an example of a changed *projects.h* file:

```
// Project Name Project Manager

 Acme, // Robinson

 Framis // Uyeno
```

Step Six

The *Tracker.pdl* file has a number of placeholder values that you need to change.

Change the following #define statements:

```
#define CZAR 'root'
#define BBOARD 'root'
```

Set the *CZAR* symbol to the email address of the tracking system administrator. Set the symbol *BBOARD* to the email address of the person responsible for screening and assigning requests, if appropriate. For example,

```
#define CZAR 'Nicholas'
#define BBOARD 'Howard'
```

Change the placeholder values in the *System* field declaration and its help text:

```
system: one-of
 SYSTEM_1, SYSTEM_2, SYSTEM_3
```

Change the help text for the *Project* field.

Review the rule regarding the *Due Date* field for both the SUBMIT_BUG and SUBMIT_RFE transitions. They both use the rule:

```
due_date.is(due_date.setDefault('now +30:00:00:00'));
```

This rule sets the *Due Date* field to 30 days from the current date. Change the value `30:00:00:00` as appropriate.

Step Seven

Now that you have made some basic changes, you should have a feel for modifying the RTS applications. The process is the same for major changes.

When the tracking system is ready for the end users, you need to make the tools directory available to them, using NFS, *rcp*, *rdist*, or other means. The users then need to run *inst* to install *Tracker.sw.user* and *Tracker.sw.userLinks*. Finally, they must run *tvinstall* giving the tools directory as an argument to create the necessary links.

Chapter 6

Advanced Design Techniques

This chapter tells you how to employ advanced techniques when designing a system with Tracker. It covers the following topics:

- Using dates
- Customizing resources
- Using the *exec* functions
- Importing data
- Preparing translation scripts

Using Dates

Date and time information is an important part of bug and enhancement request tracking systems, as it is for most database applications. The date on which a request is submitted, the due date for resolving the problem, and the actual fix date are just a few of the uses for date and time values in Tracker-based applications.

Tracker provides the date field type to support date and time data storage and manipulation. Date values are stored in Tracker databases, used in queries, displayed, and entered in a variety of formats.

Representing Date Values

Date field values represent a point in time to the nearest second. They do not represent a time interval such as *six seconds* or *two days*. Because they rely on UNIX time data types and function calls, date values represent times only since January 1st, 1970. Date values are ordered from this starting point, the lowest value, and increase toward later dates. Therefore, a date that occurs after a second date will have a higher value.

Input Formats

Internally, Tracker represents dates as integers—a convenient form for storage and retrieval. For data entry purposes, however, other formats are preferred.

Tracker provides a wide variety of date input formats. Dates can be supplied with as little information as the year or month, or specified to the nearest second. You can enter dates as a base time point plus or minus a time interval. You can also specify expressions for the current year, month, day, hour, or second.

Table 6-1 provides an example of the various values possible for a given date, along with the Tracker interpretation.

Table 6-1 Date Interpretation Examples

Date Input	Tracker Interpretation
7/28/93	Tue Jul 28 00:00:00 PDT 1993
July 28, 1993	Tue Jul 28 00:00:00 PDT 1993
July 28 1993	Tue Jul 28 00:00:00 PDT 1993
28-July-92	Tue Jul 28 00:00:00 PDT 1993
July 28 10:00 1993	Tue Jul 28 10:00:00 PDT 1993
10:00 July 28 1993	Tue Jul 28 10:00:00 PDT 1993
July 28	Tue Jul 28 00:00:00 PDT 1993
July 28 10PM	Tue Jul 28 22:00:00 PDT 1993
July 28 10PM EDT	Tue Jul 28 19:00:00 PDT 1993
July 1993	Wed Jul 1 00:00:00 PDT 1993
July	Wed Jul 1 00:00:00 PDT 1993
1993	Wed Jan 1 00:00:00 PST 1993
10PM	Tue Jul 28 22:00:00 PDT 1993
10:30:59 PM	Tue Jul 28 22:30:59 PDT 1993
today	Tue Jul 28 00:00:00 PDT 1993—Start of current day. Equivalent to “this day.”
now	Tue Jul 28 11:10:14 PDT 1993—Current time to the nearest second.
this year	Wed Jan 1 00:00:00 PST 1993—Start of current year.
this month	Wed Jul 1 00:00:00 PDT 1993—Start of current month.
today + 30:00:00:00	Thu Aug 27 00:00:00 PDT 1993—30 days from today.
July 28 - 72:00:00	Sat Jul 25 00:00:00 PDT 1993—72 hours before July 28th of the current year.

Now let’s examine the rules Tracker uses to understand date input.

Year

The year can be expressed as a 4-digit number or as a 1- or 2- digit number representing the year in the current century. For example, “92” means 1993, “44” is illegal (1944 is before 1970), and “1” is illegal now, but anytime after the beginning of the next century will mean “2001.”

Month

The month can be either a number from 1 to 12 or a month name or a 3-letter month abbreviation. Case is not considered.

Day of the Month

The day of the month is expressed as a number from 1 to 31.

Day of the Week

The day of the week can be included in date input, but is not used to interpret the date. In fact, the day of the week is ignored in date input. The day of the week can be a day of the week name or a 3-letter abbreviation.

Time of Day

The syntax for specifying the time of day is:

`HH [ :MM [ :SS ] ] [` *am_or_pm* `] [` *time zone* `]`

The minute and second are optional, as are the AM or PM designator and the time zone. If neither AM nor PM is included in the time of day, the hour is interpreted on a 24-hour basis. If a time zone is not specified, the current local time zone is used. The time zone can be either a 3-letter time zone name or a 4-digit offset from GMT (now known as Coordinated Universal Time or UTC).

The recognized time zones are shown in Table 6-2.

Table 6-2 Time Zone Interpretation

Time Zone	Difference in Hours from GMT
GMT (Greenwich Mean Time)	0
GST (Greenwich Standard Time)	0
GDT (Greenwich Daylight Time)	0
EST (Eastern Standard Time)	-5
EDT (Eastern Daylight Time)	-4
CST (Central Standard Time)	-6
CDT (Central Daylight Time)	-5
MST (Mountain Standard Time)	-7
MDT (Mountain Daylight Time)	-6
PST (Pacific Standard Time)	-8
PDT (Pacific Daylight Time)	-7
BST (British Summer Time)	1
MET	1
EET	2
JST	9

Date Plus or Minus Interval

You can specify a date in relative terms by adding or subtracting a time interval from a given base date. The adjustment always follows the base date and is specified as follows:

```
+ | - [ [ [ DD: ] HH: ] MM: ] SS
```

Current Time References

The current date and time can be referenced in date input. Tracker supports these current date and time reference forms:

- *this second* (also *now*)
- this minute
- this hour
- *this day* (also *today*)
- this month
- this year

Display Formats

Date display formats are as various as the input formats. Tracker designers and end users can customize the date display formats to meet their individual needs.

Tracker uses `cftime(3C)` to format date values for display. The environment variable *CFTIME* is used to alter the date display format. See the `cftime(3C)` man page for details on setting *CFTIME* environment variable.

Comparing Dates

Tracker uses these operators to compare date values: <, <=, =, >=, >, and <>. These are especially useful in DML. When a date field is compared with a literal date value, the granularity of the literal date value controls the granularity of the comparison. For example, consider the DML statement:

```
select * from tracker_request where due_date = July 1993;
```

The literal date value is specified only to the month. The query will therefore match all *due_date* values anytime during that month, that is, from July 1 to July 31. Consider the statement:

```
select * from tracker_request where due_date = July 28 1993;
```

The date value is a specific day and will match due dates from midnight to 23:59:59 of that day.

If the date is fully specified, as in:

```
select * from tracker_request where due_date = Jul 29
09:25:27 1993
```

then the match must be exact.

These date rules also apply to DML and the Tracker GUI.

Date Entry in PDL

Date values must be enclosed within single quotes when entered into PDL. For example:

```
due_date.setValue('now + 30:00:00:00');
```

Date Entry in DML

You need not use quotation marks for date values in DML. Date values can be quoted using the explicit typing conventions for DML, for example:

```
select * from tracker_request where reopen_date = date
'10:30PM';
```

However, using quotes around date values is sometimes necessary to enable the DML interpreter to parse the statement correctly.

Customizing Resources

As is true for any X11 application, CASEVision / Tracker applications such as *rtsquery* have a large number of resources that control their appearance and behavior. Default values for these resources are provided in an *application defaults file,* stored in the directory

```
/usr/lib/X11/app-defaults
```

If you want to change any of these values for your own environment, you can set new values for them in your own resource file:

```
~/.Xdefaults
```

However, handling resources in Tracker applications is slightly harder than handling other X11 applications. The widget names, and even the name of the application itself, are generally customized by the system designer, so this guide cannot refer to them directly.

X11 resource customization is a complex subject. This section is not a complete guide. One of the best sources for further information is the *Xlib Programming Manual*, by Adrian Nye, Volume One of the O'Reilly *X Window System Series*.

Naming Applications, Widgets, and Resources

Your first step in customizing resources is to determine the name of the application, which is the name of the command that you type to bring up the application. This name also appears in the window title bar when you run the application. The main application shipped with Tracker is named *rtsquery*, and that name is used throughout this section. You can also use a name of your choice.

Everything in X11 has two names. The "name" mentioned above is actually the *instance* name; there is also a *class* name. The class name is sometimes shared by several instances (similar to a family name). This is more often true of widget and resource names than of application names; it is never true of Tracker application names.

Each *auxiliary view* (accessed from the main view's *Views* menu) in an application has an instance name, which appears in the title bar. The class name of an auxiliary view is the class name of the IRIS IM™ widget used to implement the view: *XmForm*; the auxiliary view class name is not related to its instance name. (IRIS IM is Silicon Graphics's port of the industry-standard OSF/Motif™ for use on Silicon Graphics systems.)

All of the individual widgets that make up the application also have names, both instance and class. The system designer may have assigned names to some of the application's widgets in the PDL file that defines the application.

You can use the UNIX command *more* to look at the application file, */usr/local/bin/rtsquery,* and see these names. Tracker automatically assigns names to any widgets that are not specifically named by the designer. The generated names are not easy to predict, so they will probably not be useful to you.

Finally, the resources have both instance names and class names. The instance name of a resource tells you what it actually does; the class name generally tells you what kind of information it is. For example, the resource *useSmallFonts* (instance name) controls whether *rtsquery* uses the smaller of its two sets of fonts. The class name is boolean (as is true for many other resources), which tells you that it takes values such as *True* and *true* (meaning use small fonts), or *False* and *false* (meaning do not use small fonts).

Using Names

Names, especially the application names, are used throughout X11. In this section only two uses are discussed.

Naming Application Default Files

You can use the application class name as the name of the application defaults file. Since *rtsquery*'s class name is *Rtsquery,* its application defaults file is in */usr/lib/X11/app-defaults/Rtsquer*y.

Setting Resource Values

You can also use all of these names to set resource values. To set a resource, you name:

- the widget for which you want to set the resource
- the name of the resource itself
- the value to which you want the resource set

If the resource is used only by one widget, or if you want to set it for all widgets that use it, you can omit the widget name; all widgets using that resource will pick up the set value.

Since there are many widgets in an application, several widgets often end up with the same name. To clarify which one you want, you can specify the *widget hierarchy* that contains the particular widget. Widgets are arranged inside one another, like a set of bowls in graduated sizes, and this structure is directly reflected in the PDL text that you find in the application file. A large widget makes up each whole window (called the *view*) and contains all of the other widgets. Arranged inside and completely filling the view are several *display* widgets, which create the sashes that allow you to change their height. Inside each display widget are other widgets: *control-bars*, *rows*, and so on. If you look at your application and the application file side by side, you should have no trouble matching the features and identifying the names that the system designer has assigned.

To set a resource for some widgets of a given kind, but not for others of the same kind, you need to specify the names of some of the containing widgets, as well as the name of the widget you actually wish to control. This is done with a limited sort of *wild-carding*. For example, to exactly specify a particular widget, you can specify the application name and all the widget names down to the end, as in:

```
app.widgOne.widgTwo.widgThree.resource: value
```

Or, more commonly, you can leave out some or all of the widget names, replacing them with an asterisk:

```
app*widgThree.resource: value
```

Since an application defaults file is only used by the "right" applications, you don't need to specify even the application name. Most resources are specified like this:

```
*widgThree.resource: value
```

You can use this simple form in your *~/.Xdefaults* file too, as long as you are sure no other applications will use the value. But it is generally safer to include the application name.

You can also set resources for all data of a particular type, which includes data in lists that do not have widgets. To do this, you must concatenate the type name (from the PDL *fields* section) and the resource itself:

```
Rtsquery*file_fileDisplayStyle: \
   FileDisplayVobStorage
```

Resources that you set in your own resource file usually take precedence over settings in the application defaults file. However, precedence is also based upon exactly how the resource is specified. To override a setting from the application defaults file, follow these steps:

1. Copy the line from the application defaults file into your *.Xdefaults* file.
2. Add the application name before the asterisk.
3. Change the value (the part after the colon).
4. Reload your resources, either by logging out or by typing

   ```
   xrdb -load ~/.Xdefaults
   ```
5. Restart the application.

Personal Tracker Resources

This section lists the resources you'll most likely want to set in a Tracker application.

instanceName: DefaultValue

Where appropriate, the description will mention which kinds of widgets use this resource. Where this is not mentioned, the resources control the behavior of the entire application.

executeStartupQuery: True

By default, if the file *~/.<appname>-query* (or the file specified by the resource `*startupQueryFileName`) is found upon startup, the query stored there (by the application's Query menu "Save As Default" item) is executed immediately so that the application comes up displaying its results. Setting this resource to `False` prevents this.

useInvalidDataDialogs: True

All widgets that display data from the database obey this resource. If invalid data is entered into such a widget during an edit and this resource is set to `true`, a highlight is drawn around the field and a dialog is popped up to explain the error.

useInvalidQueryDialogs: True

All widgets that display data from the database obey this resource. If invalid data is entered into such a widget during a query and this resource is set to `true`, a highlight is drawn around the field and a dialog is popped up to explain the error.

fileDisplayStyle: FileDisplayVobMount

All widgets that display data of type *file* obey this resource. File names that name a file in a CASEVision / ClearCase Versioned Object Base (VOB) can be displayed in two formats. The default format, `FileDisplayVobMount`, is a standard ClearCase version-extended pathname, which includes the file system path to the file and the version information, for example:

```
/vobs/CASE/usr/src/foo.c@@/main/37
```

This format is familiar to most users. If displayed on a system where the VOB is mounted in another location, it will show the new location. If displayed on a system without the VOB mounted, or even without ClearCase installed, the string displayed to the last person who edited the file will be used.

The other display format, `FileDisplayVobStorage`, shows where the VOB is actually located, for example:

```
(vobhost:/storage/CASE)/usr/src/foo.c@@/main/37
```

This presentation can be useful to administrators investigating user complaints that the first form names a file that doesn't exist.

queryFetchCount: 50

This resource controls the number of entities retrieved from the database each time the query results pane is updated. Setting this resource higher than the default may improve the scrolling performance of query results. Setting it too high may increase the delay before the first query results are displayed.

maxAssistValues: 25

This resource applies to the layout of the field value options in the GUI. If the number of options is 25 (the default) or fewer, they appear on a rollover submenu reached from the “Values” menu item on the field menu. If there are more than 25 options, Tracker displays them in a scrolling list. You can set the default at which the scrolling list appears to any number you choose.

keyboardFocusPolicy: explicit

This setting means that you must click the mouse button in a field before you can enter data. Setting it to *pointer* means that you can type into whichever field is under the mouse pointer.

XmForm.traversalOn: True

This setting means that you can move from one input field to another using **`<Ctrl Tab>`** (move forward) and **`<Shift Tab>`** (move backward). **`<Tab>`** also moves you forward from single line fields, but in multiline fields it merely inserts a tab. Setting it to `False` means you must move the mouse pointer to another field to enter data in it.

scheme: Lascaux

A scheme is a set of coordinated colors, fonts, and other properties. A number of schemes are provided in the CASEVision environment (installed along with Tracker). You select which scheme your CASEVision applications use by setting this resource. The default scheme, Lascaux, matches many other IRIX tools. You can preview the other schemes with *cvscheme*, which is also a part of CASEVision. (For more information, see the *cvscheme* man page or the *CASEVision Environment Guide*.)

useSmallFonts: True

Every scheme provides two entire sets of fonts. By default, Tracker applications use the set of smaller fonts because Tracker windows tend to be large and cluttered with the larger fonts. If you prefer the bigger fonts, set this resource to `False`.

Using the *exec* Functions

The Tracker PDL provides many built-in facilities for implementing applications. It supports data, process, and GUI definition. The PDL does not, however, provide everything an application designer might need to implement Tracker applications. Some applications may require resources external to the PDL. The *exec* functions let you access external resources, thus allowing hybrid applications to be constructed.

The *exec* functions are PDL functions that provide access to UNIX commands and the DML language. The next section examines the *exec* functions for UNIX command access.

Executing UNIX Commands from PDL

Two *exec* functions provide access to UNIX commands. They are *execCommand* and *execFilter*. *execCommand* executes a UNIX command and returns its exit status. *execFilter* also executes a UNIX command, but it returns the standard output of the command instead of the exit status.

The *execCommand* and *execFilter* functions are very useful in implementing Tracker applications, and the next example uses *execFilter* to demonstrate this. Suppose you define a field, *id_string*, that you wish to contain a short identifying string for each bug. You want this field to be filled in automatically upon submission of the bug report with the *$ENTITY_ID* field value and the *submitter* field value, as follows:

```
4-john
```

This cannot be accomplished with the standard methods provided for PDL fields, but you can use *execFilter* to execute the UNIX command *echo* and save the result into the *id_string* field. The PDL source to do this looks like this:

```
actions {
     id_string.setValue(
     execFilter("/bin/echo $ENTITY_ID-$submitter"));
}
```

Both *execFilter* and *execCommand* make the values of all fields available in environment variables. Thus *$ENTITY_ID* and *$submitter* reference environment variables hold the values of the two fields *$ENTITY_ID* and *submitter* respectively. Both also store all long-text field values in temporary files. The name of the temporary file is made available in an environment variable named *<fieldname>_file*, and the value of the environment variable *fieldname* is set to the special value **!** (exclamation point) to signify the field's special treatment.

Both *execFilter* and *execCommand* make other information available in environment variables. The variable *FIELD_LIST* contains the names of all the request fields. The variable *MODIFIED_FIELDS* contains the names of all fields whose values have been changed by the current transition, either by user actions or by PDL methods like *setValue*. For those fields in the *MODIFIED_FIELDS* list, the old values are also available in variables named *<fieldname>_old*, or in the temporary file named in the *<fieldname>_old_file* variable for long-text fields.

For an example usage of the *execCommand* function, see the RTS main application PDL file, */usr/Tracker/RTS/Tracker.pdl*. This file uses *execCommand* to execute the shell script *rts_notify*. See the shell script for examples of using environment variables to access field data.

Executing DML Select Statements

The *execSelect* function is used to execute a DML *select* statement against the Tracker database from PDL. Its argument is the complete *select* statement text, and its return value is the result of the query.

This example of *execSelect* usage appears in */usr/Tracker/RTS/Tracker.pdl*:

```
tempShortText.
   setValue(execFilter(’echo “select manager from project
      where name = ”$project”;”’));
owner.setValue(owner.isSet ?
   owner.value :
   (project.isSet ?
owner.setValue(execSelect(tempShortText.value)) :
owner.setValue(execFilter(’/bin/echo $bboard’))));
```

In this example, *execSelect* retrieves the name of the project manager from the database. The request is then conditionally assigned to the project manager. First the *select* statement itself is constructed using *execFilter* with *echo* to insert the project field's value into the *select* statement text.

The result of this *execFilter* is stored in a temporary scratch field and then later used as the argument to *execSelect*. The result of the *execSelect* is then used to set the value of the owner field.

The *execSelect* function is useful only when the result of the query is a single field value from a single request. If the *select* statement results in more than one request being selected, then *execSelect* returns no value. If more than one field is selected, only the first field mentioned in the *select* statement is returned.

Importing Data

Most users do not have the luxury of creating a new request tracking system. By the time you decide you need a product like Tracker, you already have a collection of defect reports. Once you have formalized your tracking process into a Tracker system, you will need to copy the information from your old system into your new Tracker system.

If your old system is on-line and capable of producing text files in a fairly consistent format, you should be able to build a tool to import your data into your Tracker system. This section presents one such tool, which was used at Silicon Graphics to import the entire Silicon Graphics bug history, over 40,000 bugs, from a previous system into our Tracker system. The example uses real data, including one of the actual bugs filed against Tracker during early development (now fixed).

You may find it easy to do your import in a two-step process:

1. Move the data into Tracker.
2. Use *dml(1)* interactively to clean up any minor problems that arise during import.

This is usually quicker than polishing your import script until it’s flawless and loading and reloading the database while you test it. You can make your script do the entire job, if the effort seems justified. The script presented here is in continuous unattended use, importing data on an hourly basis from the old system (which is still in use in some parts of Silicon Graphics). Some of the lessons learned while developing this script may save you some trouble in developing your script.

Basics

The basic approach you'll use is to translate your data into the Tracker Data Manipulation Language (DML). DML is similar to many fourth generation database languages, such as SQL. It lets you enter, examine, and modify the data in the database. You can write an application that reads your current database directly, translates the data into DML, and feeds the DML to the interpreter, *dml*. More likely, however, you'll need to go through an intermediate text format, as outlined here.

First take a look at the DML aspects that are particularly important during an import; if you're able to translate directly from one to the other, that may be all you will need.

Explicit Typing

When importing data, it's best to explicitly type the data. This is an optional DML feature because most data values are automatically recognized as belonging to a particular type; however, explicit typing, can clear up ambiguities, such as the difference between short-text and long-text values. More importantly, while you're importing data from another system, some data might not match expectations—either Tracker's expectations, or the expectations you had when you wrote the import script. Explicit typing helps to ensure that such surprises don't slip past you into the Tracker database. Even if you take the approach recommended above, you'll at least want to know that there is a problem.

Explicit typing is accomplished by enclosing the data item in single quotes (apostrophes), and preceding it with the type name, that is, the name used to declare the field in the PDL file. Both long-text and short-text string types already require the surrounding quotes; you need to add only the type-name.

Duplicate Suppression

Your first attempt to import a large database may not be flawless, but you should be able to import a majority of your data easily, and then focus on the few reports with difficulties. You don't want to discard the reports you've inserted correctly, yet you don't want to turn one real report into two records in the database.

You should focus your attention on getting the import script right, rather than on remembering exactly which reports worked and which failed. The DML includes a very useful construct for this sort of situation, the `insert ... where` statement (this is not in standard SQL).

A normal `insert` statement creates a new record unconditionally, which is fine for the first time, but not so good when you're doing updates. The `update ... where` statement updates records already in the database, but does nothing if the record described in the `where` clause doesn't already exist. If only these two statements were available, you would be required to know the exact state of your database at all times and to adjust your import process accordingly. This is inconvenient during an import.

The `insert ... where` statement solves the problem. It's a synthesis of the `insert` statement and the `update` statement: if one or more records can be found that match the `where` clause, then it behaves as an `update` statement, changing the existing record(s) and creating no new ones. If, however, no such record exists, `insert ... where` behaves as an `insert` statement, creating one new record with all the values included in the statement.

When you're importing a block of data using your import script, you'll probably encounter a few reports that cause problems in some way. Often, you can make a minor enhancement to your import script, and re-import the same block of reports. If you've written your script to use `insert ... where`, the reports that succeeded before will merely update themselves to the same values; the reports that failed before will be inserted now.

To use `insert ... where` effectively, you'll need to know the field(s) of the incoming reports that uniquely identify them. (Your Tracker database assigns an *$ENTITY_ID* uniquely to each report, but you won't know that value for the incoming data, so it can't be used here.) Your old system may already have a sequence number or something similar. If not, you may be able to create a unique key using several fields. For example, if your old system records who originally submits a given report, and also records the time of submission fairly precisely, the submitter plus submission time will probably be unique because no single person can submit more than one report per second.

Multiline Data

Some fields values may cover more than one line. For example, nearly every system has at least one field where the problem is described, which is usually more than a single line. Similarly, lists of items can span several lines. The example presented here uses *nawk(1)* to translate the text, however, and *nawk* is primarily line-oriented. A problem arises: you'll be writing this *nawk* script to recognize certain lines that identify fields and their values in the input, but you don't want to be misled by similar text buried within a body of text.

The solution used here is to notice when a multiline item begins; its beginning is easily recognized in the same way any other field value is recognized. A flag is set, indicating that the parsing is currently somewhere within a multiline item. A few rules concerned with noticing the end of such an item appear at the top of the script (so that they're considered before the general rules); these rules match only while the flag is set. These rules are principally responsible for recognizing the end of the multiline data.

Embedded Quotes

The string data itself may contain quotes besides the quotes used to delimit it. The DML requires that such a quote has a backslash in front of it, like this: (\'). This in turn makes backslashes special: they, too, require backslashes in front of them.

Performance Considerations

If you have so many defect reports to import that you're considering writing a script to do it, you can probably determine how long this script will take to run. You can improve the performance of your script by making it surround groups of insertions in transactions; otherwise, each individual insertion will be a transaction by itself, which is time-consuming for the server. The exact number of reports you include in a given transaction isn't too critical in this context; grouping at least ten together is advantageous, but above that the rate of gain begins to fall off. If you make the transactions too large (say, hundreds of reports), things will slow down because the server grows too large (storing the pending transaction) and begins to page unduly. Twenty-five reports per transaction seemed to be a good compromise.

Text Output from Your Old Database

Your first requirement is to deal with the text output from your old database. The details of the format are not too important, since you'll be writing a translator anyway; make sure, though, that the format is easy to translate:

- Start each new field on a separate line.
- Start each line with the name of the field as you've declared it in the PDL.
- Use consistent punctuation style.
- If at all possible, enquote the string values in this text so that strings begin and end with an apostrophe, and embedded apostrophes are escaped with backslashes.

The input format for this example is shown in Figure 6-1. Several details of this format are worth pointing out:

- The line of asterisks is part of the output; it precedes each report (called *incidents* in this system).
- The *incident_id* is the unique identifier for the old system.
- The data is enquoted, but is not entirely consistent; some integers are quoted, some are not.
- Fields whose values are potentially multiline (such as description) have a double colon; all other fields have a single colon.
- There are many fields in this form. In fact, the system as a whole has even more than these; when a field is blank for a given report, this text copy simply ignores that field.

Finally, notice that the first two lines of the description can be easily mistaken for field values. The older system that was being replaced here was based upon *netnews* and *email* messages—simple text messages that were read by an administrator to sort out details such as priority and assigned engineer. This stylized form of text entry made that job easier.

```
*****************************************************

incident_id : 106064
submitter : ‘jackr’
submitter_machine : ‘dblues’
opened_date : ‘May 04 1993 03:00AM’
category : ‘software’
classification : ‘rfe’
summary : ‘RFE: Mouse motion posts redundant popups’
priority : ‘4’
reproducible : T
SGI_only : T
message_id : ‘kcamfic@sgi.sgi.com’
newsgroups : ‘sgi.engr.case.bugs’
released_product : T
reported_by_customer : T
tvbug_id : 38969
description :: ‘assign to: jackr
priority: 4
When there\’s a syntax error in the entry form, every time
the mouse passes through that field, another copy of the
error message is popped up.

Since we expect mouse motion while these alerts are up
(context-sensitive help and so on), this is tacky. Perhaps
each pane or field could keep track of whether it already
has an alert up? Does the Vk message thingie return the
widget ID (so you could check if it\’s still alive)?’
resolution_id : 106064
project : ‘tracker’
status : ‘closed’
dev_priority : ‘4’
assigned_engineer : ‘johnt’
fixed_by : ‘jackr’
closed_date : ‘May 21 1993 01:13AM’
age : 79
fix_description :: ‘fixed by previous take’
modified_date : ‘May 21 1993 12:13PM’
modified_user : ‘jackr’
```

Figure 6-1 Input Format

Preparing Translation Scripts

Keep your PDL file handy while preparing your translation script so that you can easily generate the code necessary to include the explicit typing. The *fields* section of the PDL file used in this system follows.

```
fields {
 Product         : short-text;
 SGI_only        : boolean;
 alpha           : short-text;
 assigned_engineer   : short-text;
 assigned_group  : short-text;
 category        : one-of software, hardware,
                         documentation ...;
 classification  : one-of bug, rfe, note;
 closed_date     : date;
 command         : short-text;
 description     : long-text;
 doc_affected    : short-text;
 fix_descriptio  : long-text;
 fix_policy      : short-text;
 fixed_by        : short-text;
 incident_id     : int;
 irix_release    : short-text;
 machine         : short-text;
 message_id      : short-text;
 model_cpu       : short-text;
 model_gfx       : short-text;
 modified_date   : date;
 modified_user   : short-text;
 newsgroups      : list-of one-of sgi_engr_case_bugs,
                                 sgi_bugs_compilers,
                                 sgi_bugs_dogwood,
                                 sgi_bugs_cypress,
                                 sgi_bugs_printware,
                                 sgi_bugs_lonestar
                                  ...;
 opened_date      : date;
 peripheral       : short-text;
 priority, dev_priority, CSD_priority  : int;
 product_version  : short-text;
 project          : short-text;
 released_product : boolean;
 reported_by_customer : boolean;
 reproducible     : boolean;
 resolution_id    : short-text;
 submitter        : short-text;
 submitter_domain : short-text;
 submitter_machine : short-text;
 summary          : short-text;
 to_incident_id   : short-text;
 importdate       : date;
}
```

The Translation Script

The translation tool in this case is merely a *nawk* script that recognizes each field by name and translates each one in a separate *nawk* production. This approach enables you to clearly recognize when an unexpected field comes along. This inflates the script quite a bit, with repetitious and uninteresting text. Most of the useful details are either at the head or at the tail.

Here are a few highlights of the script, the text of which follows immediately afterwards.

- The script expects to be run with one or more file names in its command line. A verbose flag, `-v`, may be provided first, producing some diagnostic output. You'll probably be doing a lot of diagnosing of this script, as you learn what your data actually looks like (instead of what you thought it looked like); you can include any debugging aids.
- A collection of *nawk* functions are defined at the beginning:
    - *rpt()* merely reports a message to *stderr* so that it doesn't get mixed into the output stream, which is probably being fed to *dml*.
    - *error()* reports a message, annotated as an error, which is an important distinction, even when you're both the author and the user of the script.
    - *barf()* reports an error and also arranges to skip the rest of the current incident.
    - *beginbug()* performs initial tasks at the beginning of each bug.
    - *where()* emits the `where` clause of the `insert ... where` statement used to put the data into the Tracker database. As you'll see below,

the *incident_id* is noted when encountered in the body of the input, so it may be used here. *where()* is only used from within.

- *endbug()*, performs necessary closing tasks. The counters maintained here keep track of when to close one transaction and open another.
- *intval(), enumval(), boolval(), strval(), lstrval(),* and *dateval()* all reformat the input format into the proper format for each particular data type.
- *startLongText()* is called whenever a *long-text* field is encountered. The test at the top checks if the field actually has only one line of text. If there are several lines, then the flag *InLongText* is set.
- Two productions follow, guarded with by the *InLongText* flag. They recognize two different kinds of transitions from one report to the next.
- The (`skipbug == True`) production implements the skipping of unparsable data set up by *barf()*. A production guarded by the flag *InBugHeader* is concerned with skipping the blank lines that follow the line of asterisks that begin a bug. The final *InLongText* line recognizes and passes through the lines that make up the body of a long-text field. They also notice the end of the field (notice the careful handling of apostrophes and back slashes).
- A long string of trivial productions follows, each of which recognizes one particular field and calls the appropriate formatting routine (*intval()* and the other routines described above).
- Eventually, the exception conditions are handled (delineated by a long comment line consisting of hash marks).
- The *END* production is executed by *nawk* when it runs out of input data. In addition to ending the current bug as it would have if a new bug had been encountered, *END* deals with the possibility that the input set is not an exact multiple of the number of reports being bundled into a transaction. In this case, the `end transaction` is not printed by endbug; this would result in an error, and the entire transaction would not be performed, so *END* adds it.

Here is the translation script.

```
#!/bin/ksh

if [[ “$1” = -v ]] ; then
 shift
 verbose=-v
else
 verbose=
fi
/usr/bin/nawk ‘
func rpt(msg) {
 system(“echo >&2 “ msg);
}
func error(msg) {
 rpt(“ERROR: “msg);
}
func barf(reason) {
 error(“\\\”” FILENAME “\\\”, line “ FNR “: error: “ reason);
 skipbug = True;
 next;
}
func beginbug() {
 if (counter == 0) {
 printf “begin transaction;\n”;
 }
 printf “insert into tracker_request set \n”;
}
func where() {
 printf “ where incident_id = “incident_id;
 printf “;\n”
}
func endbug() {
 # Do nothing on first pass (a trick to handle startup)
 # Elsewise, print the where-clause.
 # Once in a while, also end the transaction (and set
counter == 0,
# to trigger beginbug() to start a new one).
 if (counter == -1) {
 counter = 0;
 }
 else if (counter == 24) {
 printf “importdate = date ‘\’’”importdate”’\’’\n”;
 where();
 printf “end transaction;\n”;
 counter = 0;
 fullcount += 1;
 }
 else {
 printf “importdate = “importdate”\n”;
 where();
 counter += 1;
 fullcount += 1;
 }
 incident_id = 0;
 tvbug_id = 0;
}
func intval() {
 gsub(“‘\’’”,””,$0);
 return “int ‘\’’”$3”’\’’”;
};
func enumval() {
 gsub(“‘\’’”,””);
 gsub(“,”,”’\’’, one-of ‘\’’”);
 $1 = ““;
 $2 = ““;
 $3 = “one-of ‘\’’”$3;
 return $0”’\’’”;
};
func boolval() {
 gsub(“‘\’’”,””,$0);
 if ($3 = 1) {
 return “boolean ‘\’’True’\’’”;
 }
 else {
 return “boolean ‘\’’False’\’’”;
 };
};
func strval() {
 $1 = ““$1;
 $2 = “= short-text”;
 return $0;
};
func lstrval() {
 $1 = ““$1;
 $2 = “= long-text”;
 return $0;
};
func dateval() {
 $1 = “date”;
 $2 = ““;
 gsub(“AM”,””,$0);
 gsub(“PM”,””,$0);
 return $0;
};
func startLongText() {
 if ($0 ~ /.*[^\\]’\’’$/) {
 print “,”;
 }
 else{
 InLongText = True;
 }
}
BEGIN {
 True = 1;
 False = 0;
 # Set things to recognize the new one
 InBugHeader = True;
 InLongText = False;
 skipbug = False;
 counter = -1;
 fullcount = 0;
}
(InLongText != True) && ($0 ~ /^Results for pvquery
command:/) {
 if (verbose==”-v”) {
 rpt(“\””$0”\””);
 }
 next;
};
(InLongText != True) && ($0 ~
/^\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\
*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*$/) {
 endbug();
 beginbug();
 skipbug = False;
 InBugHeader = True;
 next;
}
(skipbug == True) {next;}
(InBugHeader == True) && ($0 ~ /^$/) {
 # Skip
 InBugHeader = False;
 next;
}
```

```
(InLongText == True){
 print $0;
 if (($0 ~ /^’\’’[ ]*$/) ||
 ($0 ~ /.*[^\\]’\’’[ ]*$/) ||
 ($0 ~ /.*[^\\]\\\\’\’’[ ]*$/)) {
 print “,”;
 InLongText = False;
 };
 next;
}
/^tvbug_id : / {
 tvbug_id = $3;
 next;
}
/^incident_id : / {
 # integer
 incident_id = intval();
 print “incident_id = “intval()”,”;
 #rpt(“Incident: “incident_id);
 next;
}
/^CSD_priority : ‘\’’/ {
 print “CSD_priority = “intval()”,”;
 next;
}
/^SGI_only : / {
 # bool
 print “SGI_only = “boolval()”,”;
 next;
}
/^age : / {
 next; # Ignore this computed value
}
/^alpha : ‘\’’/ {
 print strval()”,”;
 next;
}
/^assigned_engineer : ‘\’’/ {
 print strval()”,”;
 next;
}
/^assigned_group : ‘\’’/ {
 print strval()”,”;
 next;
}
```

```
/^category : ‘\’’/ {
 # software (hardware?)
 print “category = “enumval()”,”;
 next;
}
/^classification : ‘\’’/ {
 # bug, rfe
 print “classification = “enumval()”,”;
 next;
}
/^closed_date : ‘\’’/ {
 # date
 print “closed_date = “dateval()”,”;
 next;
}
/^command : ‘\’’/ {
 # person
 print strval()”,”;
 next;
}
/^description :: ‘\’’/ {
 # long-text
 print lstrval();
 startLongText();
 next;
}
/^dev_priority : ‘\’’/ {
 print “dev_priority = “intval()”,”;
 next;
}
/^doc_affected : ‘\’’/ {
 print strval()”,”;
 next;
}
/^fix_description :: ‘\’’/ {
 print lstrval();
 startLongText();
 next;
}
/^fix_policy : ‘\’’/ {
 gsub(“-”,”_”,$0)
 print strval()”,”;
 next;
}
/^fixed_by : ‘\’’/ {
 # person
 print strval()”,”;
 next;
}
/^irix_release : ‘\’’/ {
 # short-text
 print strval()”,”;
 next;
}
/^machine : ‘\’’/ {
 # string
 print strval()”,”;
 next;
}
/^message_id : ‘\’’/ {
 # short-text
 print strval()”,”;
 next;
}
/^model_cpu : ‘\’’/ {
 # string
 print strval()”,”;
 next;
}
/^model_gfx : ‘\’’/ {
 # string
 print strval()”,”;
 next;
}
/^modified_date : ‘\’’/ {
 # date
 print “modified_date = “dateval()”,”;
 next;
}
/^modified_user : ‘\’’/ {
 # person
 print strval()”,”;
 next;
}
/^newsgroups : ‘\’’/ {
 # list-of one-of
 gsub(“\\.”,”_”,$0);
 print “newsgroups = (“enumval()”),”;
 next;
}
```

```
/^opened_date : ‘\’’/ {
 # date
 print “opened_date = “dateval()”,”;
 next;
}
/^peripheral : ‘\’’/ {
 print strval()”,”;
 next;
}
/^priority : ‘\’’/ {
 print “priority = “intval()”,”;
 next;
}
/^product : / {
 # list-of short-text?
 $1 = “Product”;
 print strval()”,”;
 next;
}
/^product_version : ‘\’’/ {
 # person
 print strval()”,”;
 next;
}
/^project : ‘\’’/ {
 print strval()”,”;
 next;
}
/^released_product : / {
 # bool
 print “released_product = “boolval()”,”;
 next;
}
/^reported_by_customer : / {
 # bool
 print “reported_by_customer = “boolval()”,”;
 next;
}
/^reproducible : / {
 # bool
 print “reproducible = “boolval()”,”;
 next;
}
/^resolution_id : / {
 # integer
 print “resolution_id = “intval()”,”;
 next;
}
/^status : ‘\’’/ {
 print “$STATE = “enumval()”,”;
 next;
}
/^submitter : ‘\’’/ {
 # login
 print strval()”,”;
 next;
}
/^submitter_domain : ‘\’’/ {
 # string
 print strval()”,”;
 next;
}
/^submitter_machine : ‘\’’/ {
 # string
 print strval()”,”;
 next;
}
/^summary : ‘\’’/ {
 # short-text
 print strval()”,”;
 next;
}
#########################################################
/^No incidents match criteria$/ {exit;}
/^$/ {next;}
/^[^ ]* :: ‘\’’/{
 # untested
 error(“Unrecognized long text in incident “ incident_id
 “: “ $0);
 print strval();
 startLongText();
 next;
}
{
 error(“Unrecognized short text in incident “ incident_id “:
“ $0);
 print strval()”,”;
 next;
}
END{
 endbug();
 if (counter != 0) {
 # endbug wont have noticed we need a transend here.
 print “end transaction;”;
 };

 if (verbose==”-v”) {
 rpt(“Total incidents imported: “fullcount);
 }
}
‘ importdate=”$(/bin/date)” verbose=${verbose} “$@”
```

The Resulting DML

Here is the DML text produced for the sample incident, after passing through the filter.

```
begin transaction;
insert into tracker_request set
   incident_id = int ‘106064’,
   submitter = short-text ‘jackr’,
   submitter_machine = short-text ‘dblues’,
   opened_date = date ‘May 04 1993 03:00’,
   category = one-of ‘software’,
   classification = one-of ‘rfe’,
   summary = short-text ‘RFE: Mouse motion posts redundant
syntax-warning popups’,
   priority = int ‘4’,
   reproducible = boolean ‘True’,
   SGI_only = boolean ‘True’,
   message_id = short-text ‘kcamfic@sgi.sgi.com’,
   newsgroups = ( one-of ‘sgi_engr_case_bugs’),
   released_product = boolean ‘True’,
   reported_by_customer = boolean ‘True’,
   description = long-text ‘assign to: jackr
priority: 4
When there\’s a syntax error in the entry form, every time
the mouse passes through that field, another copy of the
error message is popped up.
```

```
Since we expect mouse motion while these alerts are up
(context-sensitive help and so on), this is tacky. Perhaps
each pane or field could keep track of whether it already
has an alert up? Does the Vk message thingie return the
widget ID (so you could check if it\’s still alive)?’
,
resolution_id = int ‘106064’,
   project = short-text ‘tracker’,
   $STATE = one-of ‘closed’,
   dev_priority = int ‘4’,
   assigned_engineer = short-text ‘johnt’,
   fixed_by = short-text ‘jackr’,
   closed_date = date ‘May 21 1993 01:13’,
   fix_description = long-text ‘fixed by previous take’
,
   modified_date = date ‘May 21 1993 12:13’,
   modified_user = short-text ‘jackr’,
   importdate = Wed Jul 22 16:37:16 PDT 1993
 where incident_id = int ‘106064’;
end transaction;
```

Chapter 7

Configuration Management

This chapter describes the integration of Atria® Software's ClearCase configuration management system with Silicon Graphics Inc.'s Tracker bug-tracking tools. The ClearCase / Tracker Integration provides each application with tools for exchanging information about bugs and bug-fixing activity. It updates the Tracker database automatically, and provides reporting capabilities for both the ClearCase and Tracker user.

This integration is designed to work specifically with the RTS sample application, and is dependent on the state and field names used in RTS. It can be modified to work with custom-designed applications by changing information in the trigger scripts, which are described later in this chapter.

Note: The integration system files are contained in the Tracker.sw.clearcase subsystem. You can install this subsystem only if you have ClearCase already installed. For more information, see the *Tracker 2.0 Release Notes*.

This chapter covers these topics:

- The purpose of the integration
- The integration architecture
- The trigger scripts
- Checkout and checkin triggers
- Some typical tasks performed with the integration

The *checkout/checkin* Model

The *checkout/checkin* model is ClearCase's standard mechanism for managing the growth of an element's version tree: *checkout* creates an editable copy of a file in a user's view; *checkin* adds a new version to the element's version tree.

The ClearCase / Tracker Integration extends this model to include information about bug fixing. When checking out a file, a developer enters a bug number. The checkout succeeds only if the bug number can be validated against an existing Tracker bug report. Information about the checkout is recorded in a ClearCase *versioned object base*, or VOB, in the form of an *event record* and *attributes* (see Figure 7-1). In the Tracker database, the update appears in the "Resolved in:" field.

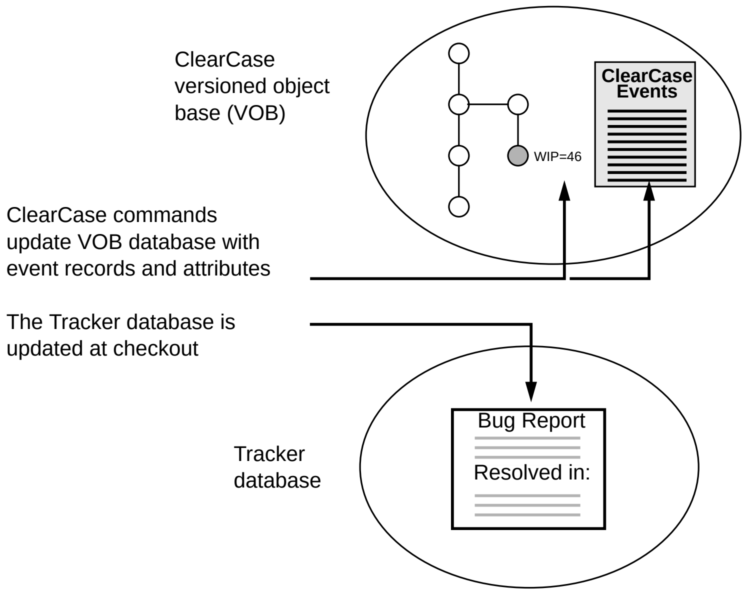

ClearCase versioned object base (VOB)
WIP=46
ClearCase Events
ClearCase commands update VOB database with event records and attributes
The Tracker database is updated at checkout
Tracker database
Bug Report
Resolved in:

Figure 7-1 Updating ClearCase and Tracker Databases

When checking in a file, a developer enters a bug number again. This begins the same validation sequence that was performed before: the checkin succeeds only if a corresponding Tracker bug report exists. Both the VOB database and Tracker database are updated to reflect the file’s change in condition.

Not every checkin constitutes a “fix,” however. The developer can create “checkpoint” (intermediate) versions of a source file during the course of bug-fixing work. Only the final version “fixes” the bug, not the intermediate versions.

A developer can also cancel a checkout with the *uncheckout* command, removing information about the checkout from both the VOB database and Tracker database.

The integration does not impose a hard connection between a ClearCase checkin (a state transition) and a “fixed bug” (a status). It includes tools to create a closer connection, but the decision to call a bug “fixed” remains a site policy decision.

Updating Databases

When you install the ClearCase / Tracker Integration, it prepares ClearCase VOBs for use with Tracker (see Figure 7-2). The integration creates *attribute types* and *trigger types* in ClearCase VOBs. The attribute types are used to track changes to source files as they proceed through the bug-fixing life-cycle, from “work in progress” (*WIP* attribute) to “fixed” (*FIXES* attribute).

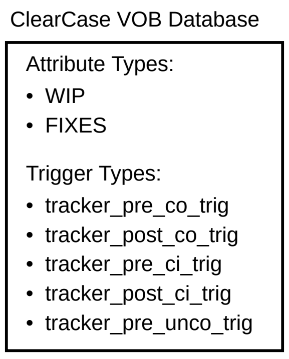


Figure 7-2 Database Modifications by ClearCase / Tracker Integration

The trigger types implement the mechanism through which ClearCase and Tracker communicate and update their databases. For details on this mechanism, see “The Integration Architecture” in the next section.

Using the Reporting Capabilities

The integration provides ClearCase users with tools for tracking status:

- The *find_wip* script lists every version of an element with the *WIP* attribute. For example:

  ```
  % find_wip base.c
  base.c@@/main/CHECKEDOUT WIP 3
  base.c@@/main/ports/2 WIP 6
  ```

- The *find_fixes* script lists every bug that a particular version of an element fixes, using values of the *FIXES* attribute. For example:

  ```
  % find_fixes base.c
  Version base.c@@/main/5 fixes the following bugs:
  23 6 3
  ```

Note: Each version can have only one instance of the FIXES attribute, but this does not mean that it can fix only one bug. For more details, see “Using the find_fixes Utility” later in this chapter.

Configuration Tools

The ClearCase / Tracker Integration includes a tool for enforcing site policies. It takes the form of an editable “policy file” (default: *policy_vars.sh*), which defines UNIX environment variables used by the integration software. The environment variables, which can be set in one or more policy files, include:

- bug number request policy:
  determines if a user must specify a bug number when checking out or checking in a file
- user validation policy:
  determines if a user must be authorized to work on a bug, or if *anyone* can work on the bug (no validation)

- "incomplete cycle" policy:
  determines how the system responds when operations occur out of order; for example, if the checkin bug number is different from the checkout bug number

A complete list of the environment variables and their purpose is given in Appendix A, "The policy_vars.sh File."

You can set other environment variables in the policy file, such as the Tracker Administrator's name, the character used to "checkpoint" element versions ("no bug-fix" character), and other general items.

You can also let different groups of ClearCase users have different policy files. For example, you might set up a policy file for a new development group that does not require users to enter a bug number. Or, you could set up a policy file that updates a completely different Tracker database.

Setting Up a *bug_task* Utility

The integration allows ClearCase users to set up a work environment called a *bug task*. The *bug_task* utility establishes task parameters, which include a bug number and a ClearCase view-tag.

The *bug_task* utility creates a process that is attached to a specified view. While working in that process, you need not enter a bug number during checkout or checkin; the task's bug number is used automatically.

The Integration Architecture

UNIX shell scripts, DML macros, and ClearCase triggers form the basis of the integration mechanism. Together, they allow ClearCase and Tracker to exchange information and affect each other's behavior. The integration creates *global-element triggers* in each VOB database, including:

- a pre-op and post-op checkout trigger:
  (*tracker_pre_co_trig*, *tracker_post_co_trig*
- a pre-op and post-op checkin trigger:
  tracker_pre_ci_trig, *tracker_post_ci_trig*

- a pre-op uncheckout trigger:
  tracker_pre_unco_trig

Each trigger runs a shell script. The pre-op trigger exit status determines if the operation proceeds or is cancelled.

The Trigger Scripts

The trigger scripts described in the following sections depend on several Tracker-specific scripts that access the Tracker database. If you want to use the integration for a customized database, you must modify the scripts accordingly.

All integration files, including the scripts, are located in */usr/atria/tracker*. Table 7-1 lists each script and its purpose.

Table 7-1 Script Files

Script Name	**Purpose**
validate_bug	Verifies that: - the bug number entered by a user actually exists in the Tracker database - it is an "open" bug, that is, AWAITING_RESPONSE
inform_ci	Informs Tracker about a checkin to fix a bug. It updates the "resolved_in" field of the bug and adds the name of the checked-in file to the "resolved_in" list.
inform_co, inform_unco	Inform Tracker about a checkout or uncheckout. They make no change to the Tracker database. They are provided so that the system administrator can extend the integration if desired.

checkout Triggers

The shell script run by the pre-op checkout trigger prompts you for a bug number, and then invokes a DML macro to validate the request with Tracker. The validation comprises these checks:

- the bug number exists
- no illegal state prevents the checkout

If these conditions are met, the script exits with a success status, and the checkout proceeds. The shell script run by the post-op checkout trigger attaches the *WIP=bug_number* attribute to the checked-out version. Figure 7-3 illustrates the checkout mechanism.

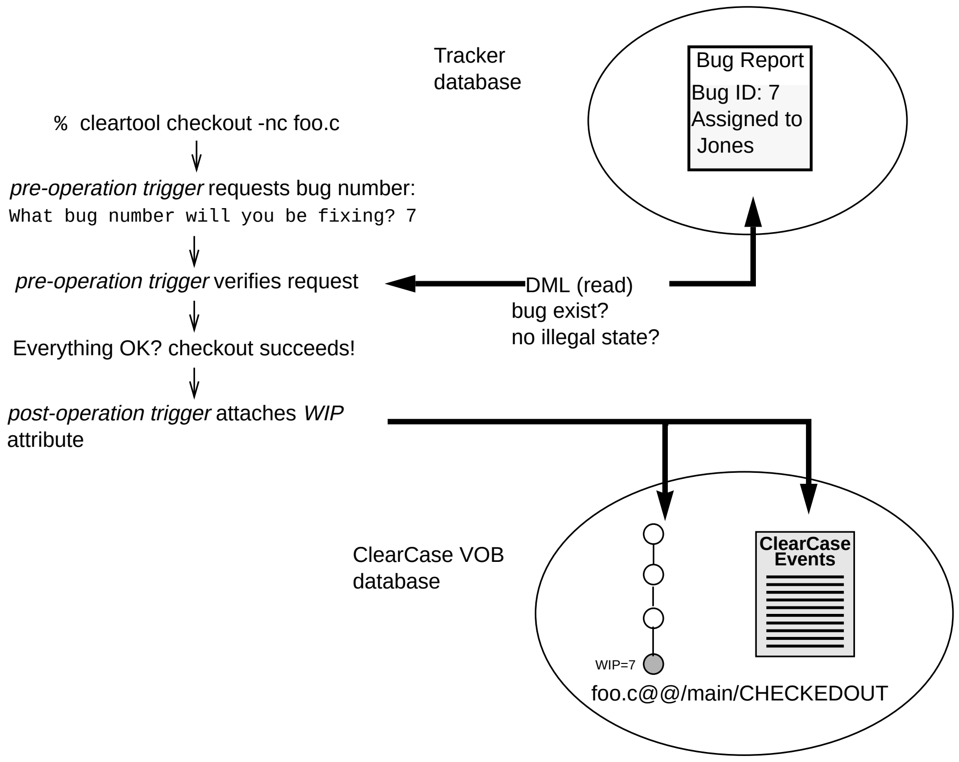

Tracker database
Bug Report
Bug ID: 7
Assigned to Jones
% cleartool checkout -nc foo.c
pre-operation trigger requests bug number:
What bug number will you be fixing? 7
pre-operation trigger verifies request
DML (read)
bug exist?
no illegal state?
Everything OK? checkout succeeds!
post-operation trigger attaches *WIP* attribute
ClearCase VOB database
ClearCase Events
WIP=7
foo.c@@/main/CHECKEDOUT

Figure 7-3 Checkout Mechanism

If the checkout request cannot be validated (for example, because the user was unauthorized to work on the bug), the pre-op trigger exits with a failure status, and the checkout is cancelled.

checkin Triggers

The shell script run by the pre-op checkin trigger checks the specified version for the *WIP* attribute and, if it exists, prompts you for the bug number that the version fixes. Entering the "no bug-fix" character (by default, `0`) "checkpoints" the version: the checkin proceeds with no change to the attribute or to the Tracker database. Attribute manipulations by the post-op checkin trigger are suppressed.

Entering a bug number invokes the same validation procedure that occurs at checkout. If the request is valid, the script exits with a success status, and the checkin proceeds.

The shell script run by the post-op checkin trigger removes all *WIP=bug_number* attributes from the element.

Note: It is possible for many versions to have the same WIP attribute value.

Then, it attaches the *FIXES=bug_number* attribute to the checked-in version. The script also invokes a DML macro to update the Tracker database with a bug report. If several versions had the *WIP=bug_number* attribute, the DML macro deletes the corresponding checkout records from Tracker, leaving only the most recent checkout record intact.

Figure 7-4 illustrates the checkin mechanism.

If a checkin request cannot be validated, the pre-op trigger exits with a failure status, and the checkin is cancelled. The post-op trigger sends mail to the Tracker Administrator explaining the problem and showing the failed DML commands. The Administrator can rerun these commands at a later time.

For more details, see "Recovering from Database Update Failures" later in this chapter.

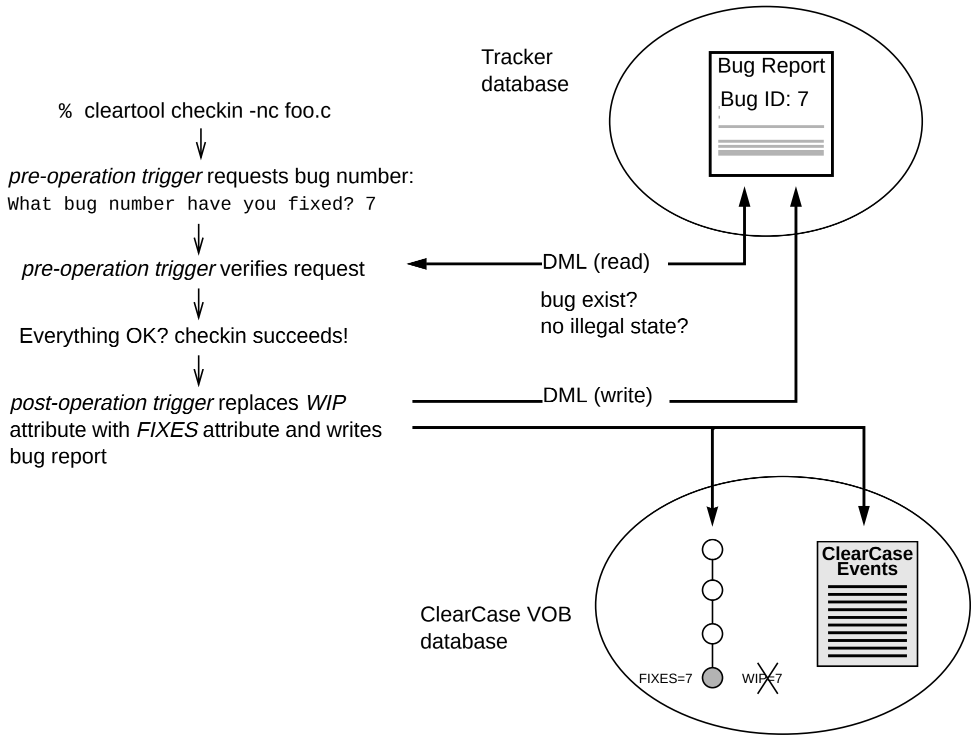

Tracker database
Bug Report
Bug ID: 7
% cleartool checkin -nc foo.c
pre-operation trigger requests bug number:
What bug number have you fixed? 7
pre-operation trigger verifies request
DML (read)
bug exist?
no illegal state?
Everything OK? checkin succeeds!
post-operation trigger replaces WIP attribute with FIXES attribute and writes bug report
DML (write)
ClearCase VOB database
ClearCase Events
FIXES=7
WIP=7

Figure 7-4 Checkin Mechanism

uncheckout Trigger

The shell script run by the pre-op *uncheckout* trigger checks the specified version for the *WIP* attribute. If the version does not have a *WIP* attribute, the uncheckout proceeds normally.

The *uncheckout* command itself removes the *WIP* attribute (if any) from the checked-out version; there is no need for the trigger to perform this operation.

Using the ClearCase/Tracker Integration

This section presents some typical usage scenarios for the ClearCase / Tracker Integration.

Scenario 1: A Typical Bug-fixing Session

A customer-reported problem has been assigned bug number 6 in Tracker. The bug involves several source files, including *parser.h*, *main.c*, and *base.c*. Using ClearCase, a developer begins working on *parser.h*.

1. The developer checks out *parser.h*, and provides the bug number:

   ```
   % cleartool checkout -nc parser.h
   What bug number will you be fixing? (0 for none) 6
   Tracker: Successful verification.
   Created attribute "WIP" on "parser.h@@/main/CHECKEDOUT".
   Checked out "parser.h" from version "/main/4".
   ```

2. After editing the file, he “checkpoints” the element by entering the “no bug-fix” character:

   ```
   % cleartool checkin -c "checkpoint" parser.h
   What bug number have you fixed? (0 for none) [6] 0
   Checked in "parser.h" version "/main/5".
   ```

3. The developer checks out *parser.h* again, and resumes working on bug 6. Instead of entering an explicit bug number, he accepts the default value by pressing **<Enter>**:

   ```
   % cleartool checkout -c "resume conditionalizing work"
   parser.h
   What bug number will you be fixing? (0 for none) [6]
   RETURN
   Tracker: Successful verification.
   Created attribute "WIP" on "parser.h@@/main/CHECKEDOUT".
   Checked out "parser.h" from version "/main/5".
   ```

4. Before getting to work, he checks status with the *find_wip* utility to see if *parser.h* has been edited to fix other bugs:

   ```
   % find_wip parser.h
   parser.h@@/main/5 WIP 6
   parser.h@@/main/CHECKEDOUT WIP 6
   ```

5. The developer checks in *parser.h*, indicating that it fixes bug 6:

```
% cleartool checkin -c "conditionalized parameters"
parser.h
What bug number have you fixed? (0 for none) [6] <Rtn>
Tracker: Successful verification.
Removed attribute "WIP" from
         "/tut_vobs/soap/parser.h@@/main/6".
Removed attribute "WIP" from
         "/tut_vobs/soap/parser.h@@/main/5".
Created attribute "FIXES" on
         "/tut_vobs/soap/parser.h@@/main/6".
Checked in "parser.h" version "/main/6".
```

6. The developer verifies that the bug fix was recorded in the VOB:

```
% find_fixes parser.h
Version parser.h@@/main/6 fixes the following bugs:
6
```

Scenario 2: Setting Up a Bug Task

Note: A developer is assigned to work on bug 5. The fix involves several files, so he or she decides to set up a *bug task* to make the job easier. The project leader has instructed everyone to make fixes in the *bug-fix* view.

1. The developer runs the *bug_task* utility to establish task parameters and start the task:

```
% bug_task
What bug number will you be fixing? 5
What view will you be using? [arb] bugfix
Starting task to fix bug "5" in view "bugfix".
Please exit shell when done.
```

   Note: *bug_task* stores the bug number in the TASK_BUGNUM environment variable. You can set this environment variable manually, and not be prompted for a bug number when you checkout or checkin a file.

2. Before getting started, the developer checks the view:

```
% cleartool pwv
Working directory view: bugfix
Set view: bugfix
```

3. The developer checks out the first of several files involved with the fix:

```
% cleartool checkout -nc main.c
Tracker: Successful verification.
Created attribute "WIP" on "main.c@@/main/CHECKEDOUT"
Checked out "main.c" from version "/main/5".
```

4. The developer checks in the file when ready:

```
% cleartool checkin -c "fixed init error" main.c
Tracker: Successful verification.
Removed attribute "WIP" from "/vobs/soap/main.c@@/main/6".
Created attribute "FIXES" on "/vobs/soap/main.c@@/main/6".
Checked in "main.c" version "/main/6".
```

5. When all files have been checked in, the developer terminates the *bug_task* by exiting the process:

```
% exit
% cleartool pwv
Working directory view: arb
Set view: arb
```

Scenario 3: Cancelling Work In Progress

A developer checks out a file with the wrong bug number, and cancels the checkout.

1. The developer performs the checkout, specifying bug 4:

```
% cleartool checkout -nc base.c
What bug number will you be fixing? (0 for none) 4
Tracker: Successful verification.
Created attribute "WIP" on "base.c@@/main/CHECKEDOUT"
Checked out "base.c" from version "/main/1".
```

2. The developer checks the file's status with *find_wip*, and realizes the mistake:

```
% find_wip base.c
base.c@@/main/CHECKEDOUT WIP 4
```

(The mistake might just as easily have been noticed at checkout time.)

3. The developer cancels the checkout, and checks the file’s status again:

```
% cleartool uncheckout -rm base.c
Checkout cancelled for "base.c".
% find_wip base.c
There are no WIP attributes on this element.
```

Scenario 4: An Incomplete Cycle

A site allows developers to checkout a file with one bug number and check it in with another (an “incomplete cycle”). A developer checks a file out and begins working on bug 3. Later, he or she checks the file in as the fix for bug 4.

1. The developer checks out the file, specifying bug number 3:

```
% cleartool checkout -nc base.c
What bug number will you be fixing? (0 for none) 3
Tracker: Successful verification.
Created attribute "WIP" on "base.c@@/main/CHECKEDOUT"
Checked out "base.c" from version "/main/2".
```

2. The developer really fixes bug 4, so he or she checks the file in with that bug number. When warned about the incomplete cycle, he or she indicates the intention of continuing anyway:

```
% cleartool checkin -c "increased buffer size" base.c
What bug number have you fixed? (0 for none) [3] 4
Tracker: Warning: A check out has not been done.
Do you wish to continue despite the warnings? [no] yes
Changing WIP value from 3 to 4
Removed attribute "WIP" from
         "/tut_vobs/soap/base.c@@/main/CHECKEDOUT".
Created attribute "WIP" on
         "/tut_vobs/soap/base.c@@/main/CHECKEDOUT".
Removing Checkout record for 3 from Bugtracking Database

Removed attribute "WIP" from
         "/tut_vobs/soap/base.c@@/main/3".
Created attribute "FIXES" on
         "/tut_vobs/soap/base.c@@/main/3".
Checked in "base.c" version "/main/3".
```

Scenario 5: An Illegal State

The Tracker Administrator has defined an *illegal state* that prevents a file from being checked in if the bug is closed. A developer tries to checkin a bugfix, but the Tracker bug status is “CLOSED”:

```
% cleartool checkin -c "changed ifdef, line 15" base.h
What bug number have you fixed? (0 for none) [7] 7
Tracker: Error: Bug Status CLOSED for bugid 7 is illegal for
checkin
cleartool: Warning: Trigger "tracker_pre_ci_trig" has
refused to let checkin proceed.
cleartool: Error: Unable to check in "base.h".
```

To checkin the file, the developer must specify another bug number, “checkpoint” the version (for example, by entering `0`), or cancel the checkout with the *uncheckout* command.

Scenario 6: Using an Alternate Policy File

A site has two Tracker databases: one for tracking *alpha* project bugs, another for tracking *beta* project bugs. The *alpha* project team uses the default policy file, *policy_vars.sh*. The *beta* project team uses an alternate policy file, */usr/atria/tracker/alt_policy_vars.sh*.

The integration software uses *policy_vars.sh* automatically, unless the ALT_POLICY environment variable is set. Therefore, *alpha* team members begin work with no special preparation. *beta* team members set the ALT_POLICY environment variable in their shell startup script:

```
setenv ALT_POLICY /usr/atria/tracker/alt_policy_vars.sh
                                                  (C shell)

ALT_POLICY=/usr/atria/tracker/alt_policy_vars.sh
                                                  (Bourne shell)
export ALT_POLICY
```

Scenario 7: Bypassing the Integration

A site allows developers to omit the bug number when they checkout or checkin a file. The “no bugfix” character is 0 (the default value). A new project has started, and the development team wants to bypass the integration mechanism altogether. Each team member sets the TASK_BUGNUM environment variable to 0:

```
% setenv TASK_BUGNUM 0                  (C shell)
% TASK_BUGNUM=0                         (Bourne shell)
% export TASK_BUGNUM
```

As an alternative, each team member starts a *bug task* with bug number 0.

Using the *find_fixes* Utility

The *find_fixes* utility compiles a list of bug fixes from values of the *FIXES* attribute. It uses the following algorithm to determine the list of bugs that a particular version of an element fixes:

- First, it lists the *FIXES* attribute value for the specified version (if any).
- Then, it lists the *FIXES* attribute value for any of that version’s ancestors.
- Finally, it lists the *FIXES* attribute value for any merge contributor that produced the specified version, or any of its ancestors.

find_fixes recursively processes merge contributor versions to determine their list of bug fixes: it examines *their* ancestor versions, any merge contributors that produced *them*, and so on.

To illustrate the *find_fixes* algorithm, consider the version tree in Figure 7-5. Several versions have the *FIXES* attribute — some on the *main* branch, others on subbranches.

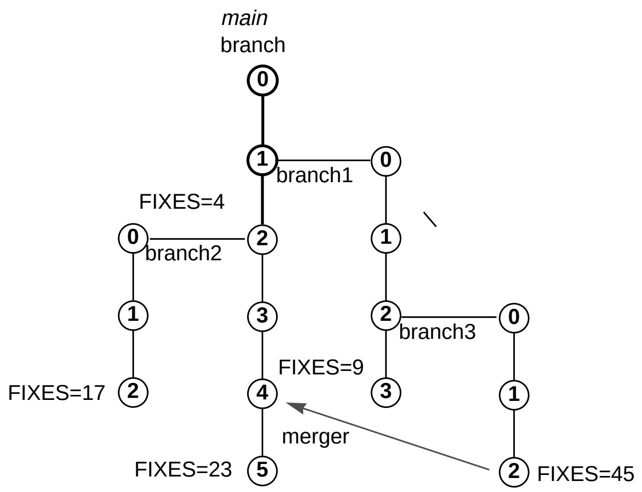


Figure 7-5 Complex Version Tree with *FIXES* Attributes

find_fixes returns the following list of bug fixes for the latest version on the *main* branch:

```
% find_fixes foo.c
Version foo.c@@/main/5 fixes the following bugs:
45 23 4
```

The listing includes bug 45 because that fix was merged into an ancestor version of *foo.c@@/main/LATEST*, which itself, fixes bug 23 (also listed). It includes bug 4 because that fix was also made in one of *foo.c*'s ancestor versions.

find_fixes returns the list of bug fixes below for the latest version on the *branch2* branch:

```
% find_fixes foo.c@@/main/branch2/LATEST
Version foo.c@@/main/branch2/2 fixes the following bugs
17 4
```

The listing includes bug 17 because the latest version on the *branch2* branch fixes that bug. It includes bug 4 because that fix was made in an ancestor version of *foo.c@@/main/branch2/2*.

Neither listing includes bug 9, however (fixed on the *branch3* branch), because neither of the specified versions have “inherited” that fix.

Recovering from Database Update Failures

If the Tracker database cannot be updated during a ClearCase checkout, checkin, or uncheckout (for example, because the Tracker database server went down), ClearCase triggers send mail to the Tracker Administrator (value of BUGTRACK_ADMIN environment variable).

Figure 7-6 shows a typical mail message, which explains the problem and includes the DML macro that failed.

```
Administrator,

The operation "checkin" of the file "/view/pete/vobs/testvob/foo.c@@/main/2
failed to be recorded in the Tracker database "/usr/tmp/RTS".

peteo was working on bug "1".

The exact DML statement that failed was:

-----------------------------------------------

update tracker_request set resolved_in =
( file '/view/pete/vobs/testvob/foo.c@@/main/2' ) where $ENTITY_ID = 1;

-----------------------------------------------

This message sent automatically by the ClearCase/Tracker
bugtracking integration trigger.
```

DML macro

Figure 7-6 Typical Mail Message to Tracker Administrator

You can recover the lost transaction (for example, a ClearCase checkin) by rerunning the DML macro manually.

Preparing VOB Databases

1. Set a view:

   ```
   % cleartool setview <any view-tag>
   ```

2. As the VOB owner or *root* user, run the *vob_prep* script over each VOB to be integrated with Tracker. The script takes one or more full pathname arguments; you can specify any pathname within the VOB. For example, these commands prepare three VOBs for use with Tracker:

   ```
   % su
   Password: <enter root password>
   # vob_prep  /vobs/scomp  /vobs/soap  /vobs/gui
   Installing types for ClearCase/Tracker integration into
   /vobs/scomp.
   -------------------------------------------------
   Created trigger type "tracker_pre_co_trig".
   Created trigger type "tracker_post_co_trig".
   Created trigger type "tracker_pre_ci_trig".
   Created trigger type "tracker_post_ci_trig".
   Created trigger type "tracker_pre_unco_trig".
   Created attribute type "WIP".
   Created attribute type "FIXES".
      .
      . similar output for /vobs/soap and /vobs/gui
      .
   ```

3. (*optional*) Instruct other ClearCase users to run *vob_prep* over any of their own VOBs that they want integrated with Tracker.

Appendix A

The *policy_vars.sh* File

This appendix lists the environment variables contained in the *policy_vars.sh* file that you can use to configure the ClearCase / Tracker Integration. They are listed in three tables:

- Tracker-specific environment variables
- environment variables that help establish and enforce site policies
- environment variables that set miscellaneous system parameters

Table A-1 lists the Tracker-specific environmental variable that can be set in the *policy_vars.sh* file. This variable is also used by commands such as *dml*.

Table A-1 Tracker-Specific Environmental Variables

Environmental Variable	Description	Values
TVBUGBASE	UNIX directory for Tracker database.	any UNIX pathname *default:* /usr/Tracker/db

Table A-2 lists the policy environmental variables in the *policy_vars.sh* file.

Table A-2 Policy Environmental Variables

Environmental Variable	Description	Values
BUGNUM_REQ_CO	must user enter a bug number on *checkout*?	TRUE, FALSE *default*: FALSE
BUGNUM_REQ_CI	must user enter a bug number on *checkin*?	TRUE, FALSE *default*: FALSE
BUG_NONE	*no bugfix* string; used only if BUGNUM_REQ_CO and BUGNUM_REQ_CI both have “FALSE” value	any character string *default*: "0"
BUGTRACK_ADMIN	user to receive mail when update of Tracker database fails	Tracker Administrator’s UNIX login name, or any other valid UNIX login name *default:* root

Table A-2 (continued) Policy Environmental Variables

Environmental Variable	Description	Values
CYCLE	determines handling of an *incomplete cycle* (for example, user enters one bug number on *checkout*, and another bug number on *checkin*)	NONE: allow incomplete cycle WARN: display message and proceed ERROR: abort ClearCase operation *default:* WARN
VALIDATE_BUG	bug-validation command, executed at *checkout* and *checkin*	validate_bug $CYCLE validate_bug $CYCLE $CLEARCASE_USER *default:* first form (no validation of username against "Assigned to" field in Tracker database)

Table A-3 lists miscellaneous environmental variables.

Table A-3 Miscellaneous Environmental Variables

Environmental Variable	Description	Values
MKTYPE_COMMENT	creation comment for meta-data types	any character string *default:* "Created for use with bug tracking triggers."
PRE_CI_TRIG POST_CI_TRIG PRE_CO_TRIG POST_CO_TRIG PRE_UNCO_TRIG	ClearCase pre-operation and post-operation trigger type names for *checkin*, *checkout*, and *uncheckout* commands	any valid ClearCase trigger name defaults: tracker_pre_ci_trig tracker_post_ci_trig tracker_pre_co_trig tracker_post_co_trig tracker_pre_unco_trig
WIP_NAME	"work in progress" attribute type	any valid ClearCase attribute name *default:* WIP
FIXES_NAME	"fixed problem" attribute type	any valid ClearCase attribute name *default:* FIXES

Appendix B

RTS PDL Files with On-line Help

This appendix contains the code listings for the RTS PDL files with the on-line help text embedded. These files are also available in */usr/Tracker/RTS*. The files are:

- *Tracker.pdl* (the main file for rtsquery)
- *rtsapprove.pdl*
- *rtsrespond.pdl*
- *rtssubmit.pdl*

Tracker.pdl

```
////////////////////////////////////////////////////////////////////
//
// File:          Tracker.pdl
// Description:   RTS default pdl
//               This is the master pdl file that defines the fields
//               and transitions for the RTS.  Each RTS app includes
//               another pdl file that defines its views.
// Author:        Pete Orelup
// Created:       Fri Apr 10 09:29:49 PDT 1992
// Language:      Text
//
// (C) Copyright 1992, Silicon Graphics, Inc.
//
//  Permission to use, copy, modify, and distribute this software for
//  any purpose except publication and without fee is hereby granted,
//  provided that the above copyright notice appear in all copies of
//  the software.
///////////////////////////////////////////////////////////////////

// This define should be changed to the login name or mail alias of
// the Tracker facilitator.

#define CZAR   'root'

// This define should be changed to contain the login name or mail
// alias for the person (or persons) responsible for assigning owner
// to reports entered without an owner or project field.

#define BBOARD 'root'

///////////////////////////////////////////////////////////////////
// Top-level Help
///////////////////////////////////////////////////////////////////
help {
    help-title 'Request Tracking System Overview';
    short-help-title 'RTS Overview';
    help-text'
The Request Tracking System (RTS) provides four applications for
accessing requests (bugs or RFEs) in request database:
    * rtsquery - the main application, it provides full functionality
    * rtssubmit - a specialized application for submitting new
      requests
    * rtsrespond - a specialized application for responding to
      requests you have received
    * rtsapprove - a specialized application for approving requests
      after resolution or rejection by the owner

Note that all applications permit you to browse requests in the
database.  Only rtsquery provides all the details; the others supply
subsets of the request data.';  }

////////////////////////////////////////////////////////////////////
//  Field Declarations
////////////////////////////////////////////////////////////////////
fields {
    help {
       help-title 'Field Entry';
       short-help-title 'Field Entry';
       help-text'
All fields in RTS display a menu if the right mouse button is held
down while the cursor is in the field.  If there are predefined values
for the field, a selection called "Values" displays that accesses a
cascading menu with the value selections. For more help, look up the
specific field.

When you select a transition from the Modes menu, all required fields
are highlighted.  If you enter an invalid value for a field, the field
becomes highlighted when you leave it.

When conducting queries, you can enter an exact value or an expression
using one of these operators:

    = <> < <= > >=             equality and inequality operators
    match                      regular expression match
    contains [any | only]      specific list value; choice of values;
                               multiple
                               specific values
    = null | <> null           test whether value is set (exists) or
                               not
    [ val1, val2, ... valN ]    test whether value is equal to one of
                               a list of values
    [ startrange:endrange, ...] range of values';
    };
```

```
//////////////////////////////////////////////////////////////////////
// Report Number Field
//////////////////////////////////////////////////////////////////////
    report_number:            int     // Equal to $ENTITY_ID by
                                      // default
       help {
           help-title 'Report # Field';
           short-help-title 'Report # Field';
           help-text'
The Report # field displays the ID assigned to the request.  You
cannot change this value.

When you are in query mode, the field becomes editable and you can
enter a specific value, a range of values, or an expression.

See also RTS Fields help.';
       };

//////////////////////////////////////////////////////////////////////
// Submitter Field
//////////////////////////////////////////////////////////////////////
    submitter:        short-text      // person
       help {
           help-title 'Submitter Field';
           short-help-title 'Submitter Field';
           help-text'
The Submitter field displays the person who created the request.  Any
text string is valid.

When you are in query mode, you can enter a specific value, a range of
values, or an expression.

See also Field Entry help card.';
       };


//////////////////////////////////////////////////////////////////////
// Date Field
//////////////////////////////////////////////////////////////////////
    submit_date:              date
       help {
           help-title 'Date Field';
           short-help-title 'Date Field';
           help-text'
The Date field contains the date on which the request is submitted.
```

```
When you submit a request, the current date is entered automatically.
When entering dates, you can use such forms as: mm/dd/yy; month day,
year;  day-month-year; day month; time. You can also enter the
variables "today" and "now". You can use additive expressions such as
(today + days) and (month day - hh:mm:ss).

When performing queries involving dates, you can enter: a range, such
as [date:date]; an operator such as < (before) or > (after); the
variables "this year" and "this month"; or any of the previously
mentioned expressions.  For a complete list of date options, see the
man page for cftime.

See also Field Entry help card.';
       };

//////////////////////////////////////////////////////////////////////
// Recommendation Field
//////////////////////////////////////////////////////////////////////
    recommendation:    one-of
                       DEFERRAL, REJECTION, RESOLUTION, DUPLICATION
       help {
           help-title 'Recommendation Field';
           short-help-title 'Recommendation Field';
           help-text'
The Recommendation field indicates the recommended disposition of the
request as of the most recent transition: DEFER (DEFERRAL), REJECT
(REJECTION), RESOLVE (RESOLUTION), and DUPLICATE (DUPLICATION).  These
recommendations are entered automatically by the transition.

See also Field Entry help card.';
       };

//////////////////////////////////////////////////////////////////////
// Type Field
//////////////////////////////////////////////////////////////////////
    type:              one-of
                       BUG, RFE
       help {
           help-title 'Type Field';
           short-help-title 'Type Field';
           help-text'
The Type field indicates the type of request: BUG for bug report and
RFE for request for enhancement.  Your tracking system administrator
may have implemented additional types. To check for these, hold down
the right mouse button while the cursor is in the type field and
```

```
select "Values" to display all allowable types.

See also Field Entry help card.';
       };
//////////////////////////////////////////////////////////////////////
// Priority Field
//////////////////////////////////////////////////////////////////////
    priority:         one-of
                      LOW, MEDIUM, HIGH
       help {
           help-title 'Priority Field';
           short-help-title 'Priority Field';
           help-text'
The Priority field indicates the designated priority for this request.
The standard values are LOW, MEDIUM, and HIGH. Your tracking system
administrator may have implemented different priorities. To check for
these, hold down the right mouse button while the cursor is in the
priority field and select "Values" to display all allowable
priorities.

See also Field Entry help card.';
       };
//////////////////////////////////////////////////////////////////////
// Owner Field
//////////////////////////////////////////////////////////////////////
    owner:            short-text  // person
       help {
           help-title 'Owner Field';
           short-help-title 'Owner Field';
           help-text
'The Owner field displays the person responsible for implementing this
request.  Any text string is valid.

When you are in query mode, you can enter a specific value, a range of
values, or an expression.

See also Field Entry help card.';
       };
```

```
////////////////////////////////////////////////////////////////////
// Project Field
////////////////////////////////////////////////////////////////////
    project:          one-of
#include "projects.h"  // This include file contains the list of
                       // projects.
                       // Edit it to change the list of known
                       // projects.

       help {
           help-title 'Project Field';
           short-help-title 'Project Field';
           help-text'
The Project field indicates the project to which the request is
assigned. The placeholder values: PROJECT_1, PROJECT_2, and PROJECT_3
are installed initially.  Your tracking system administrator has
probably implemented different project names. To check for these, hold
down the right mouse button while the cursor is in the project field
and select "Values" to display all allowable projects.

See also Field Entry help card.';
          };

////////////////////////////////////////////////////////////////////
// System Field
////////////////////////////////////////////////////////////////////
    system:           one-of
                       SYSTEM_1, SYSTEM_2, SYSTEM_3
       help {
           help-title 'System Field';
           short-help-title 'System Field';
           help-text
'The System field indicates the system to which the request is
assigned. The placeholder values: SYSTEM_1, SYSTEM_2, and SYSTEM_3 are
installed initially.  Your tracking system administrator has probably
implemented different system names. To check for these, hold down the
right mouse button while the cursor is in the system field and select
"Values" to display all allowable systems.

See also Field Entry help card.';
       };
```

```
///////////////////////////////////////////////////////////////////
// Found in Field
///////////////////////////////////////////////////////////////////
    found_in:          list-of short-text      // list-of product
       help {
           help-title 'Found in Field';
           short-help-title 'Found in Field';
           help-text'
The Found in field indicates the location(s) of the bug or
enhancement.

See also Field Entry help card.';
       };
///////////////////////////////////////////////////////////////////
// Summary Field
///////////////////////////////////////////////////////////////////
    summary:           short-text
       help {
           help-title 'Summary Field';
           short-help-title 'Summary Field';
           help-text'
The Summary field describes the request in a single line. When you
submit a request, this field defaults to the first line of the request
description, unless you have made an overriding entry. You can edit
this line at any time.  The summary line information appears in the
query results area during queries.

See also Field Entry help card.';
       };

///////////////////////////////////////////////////////////////////
// Description Field
///////////////////////////////////////////////////////////////////
    description:               long-text
       help {
           help-title 'Description Field';
           short-help-title 'Description Field';
           help-text'
The Description field contains a complete explanation of the request.
You can enter as many lines as needed. If you have set either the
$WINEDITOR or $EDITOR environment variables, then the right-button
menu for the field will have an "Edit..." selection that lets you
enter the description in your default editor and import it into the
field.
```

```
See also Field Entry help card.';
       };

////////////////////////////////////////////////////////////////////
// Dup of Field
////////////////////////////////////////////////////////////////////
    is_duplicate_of:   int            // pr-num
       help {
           help-title 'Dup of Field';
           short-help-title 'Dup of Field';
           help-text'
The Dup of field is only used when you are executing the DUPLICATE
transition.  You use DUPLICATE when you feel that a new request is a
duplicate of a previous request. If so, you must enter the report
number of the previous request in the dup of field in order to mark
the new request as a duplicate.

See also Field Entry help card.';
       };

////////////////////////////////////////////////////////////////////
// Notify Field
////////////////////////////////////////////////////////////////////
    interested_parties:        list-of short-text      // list-of person
       help {
           help-title 'Notify Field';
           short-help-title 'Notify Field';
           help-text'
The Notify field is used to add interested parties to the list of
people to be notified when changes occur to the request.  Initially,
the list contains the submitter, owner, and tracking system
administrator.

See also Field Entry help card.';
       };

////////////////////////////////////////////////////////////////////
// Due Date Field
////////////////////////////////////////////////////////////////////
    due_date:          date
       help {
           help-title 'Due Date';
           short-help-title 'Due Date';
           help-text'
The Due Date field contains the date by which the request is intended
```

```
to be fixed.  When you submit a request, the current date plus 30 days
is entered automatically. Your tracking system administrator may have
implemented a different default date.

When entering dates, you can use such forms as: mm/dd/yy; month day,
year;  day-month-year; day month; time. You can also enter the
variables "today" and "now". You can use additive expressions such as
(today + days) and (month day - hh:mm:ss).

When performing queries involving dates, you can enter: a range, such
as [date:date]; an operator such as < (before) or > (after); the
variables "this year" and "this month"; or any of the previously
mentioned expressions.  For a complete list of date options, see the
man page for cftime.
See also Field Entry help card.';
       };

////////////////////////////////////////////////////////////////////
// Close Date Field
////////////////////////////////////////////////////////////////////
    close_date:                date
       help {
           help-title 'Close Date';
           short-help-title 'Close Date';
           help-text'
The Close date field contains the date on which the request is closed.
When you approve a request, the current date is entered automatically.

When entering dates, you can use such forms as: mm/dd/yy; month day,
year;  day-month-year; day month; time. You can also enter the
variables "today" and "now". You can use additive expressions such as
(today + days) and (month day - hh:mm:ss).

When performing queries involving dates, you can enter: a range, such
as [date:date]; an operator such as < (before) or > (after); the
variables "this year" and "this month"; or any of the previously
mentioned expressions.  For a complete list of date options, see the
man page for cftime.

See also Field Entry help card.';
       };
```

```
////////////////////////////////////////////////////////////////////
// Reopen Field
////////////////////////////////////////////////////////////////////
    reopen_date:               date
       help {
           help-title 'Reopen Date';
           short-help-title 'Reopen Date';
           help-text'
The Reopen date field contains the date on which the deferred request
is to be reopened.

When entering dates, you can use such forms as: mm/dd/yy; month day,
year;  day-month-year; day month; time. You can also enter the
variables "today" and "now". You can use additive expressions such as
(today + days) and (month day - hh:mm:ss).

When performing queries involving dates, you can enter: a range, such
as [date:date]; an operator such as < (before) or > (after); the
variables "this year" and "this month"; or any of the previously
mentioned expressions.  For a complete list of date options, see the
man page for cftime.

See also Field Entry help card.';
       };

////////////////////////////////////////////////////////////////////
// Resolved In Field
////////////////////////////////////////////////////////////////////
       resolved_in:            list-of file    // list-of file
       help {
           help-title 'Resolved In Field';
           short-help-title 'Resolved In';
           help-text'
The Resolved in field lets you enter files used to implement the
request.  You can select the file access format through the
right-button menu. The text format lets you enter files as text
strings inside parenthese separated by spaces. The list format lets
you enter and delete files in a scrollable list.

See also Field Entry help card.';
       };
```

```
////////////////////////////////////////////////////////////////////
// Resolution Field
////////////////////////////////////////////////////////////////////
       resolution_description: long-text
       help {
           help-title 'Resolution Field';
           short-help-title 'Resolution';
           help-text'
The Resolution field contains ane explanation of implementation of the
request. You can enter as many lines as needed. If you have set either
the $WINEDITOR or $EDITOR environment variables, then the right-button
menu for the field will have an "Edit..." selection that lets you
enter the resolution explanation in your default editor and import it
into the field.

See also Field Entry help card.';
       };

////////////////////////////////////////////////////////////////////
// Fixed In Field
////////////////////////////////////////////////////////////////////
       fixed_releases:        list-of short-text      // list-of
                                                       // product
       help {
           help-title 'Fixed In Field';
           short-help-title 'Fixed In';
           help-text'
The Fixed in field lets you identify the product release in which the
request is implemented

See also Field Entry help card.';
       };

////////////////////////////////////////////////////////////////////
// Approver Field
////////////////////////////////////////////////////////////////////
       approver:              short-text      // person
       help {
           help-title 'Approver Field';
           short-help-title 'Approver';
           help-text'
The Approver field identifies the person with approval authority over
this request.
```

```
See also Field Entry help card.';
       };

////////////////////////////////////////////////////////////////////
// Non-displayable Fields
////////////////////////////////////////////////////////////////////

       // These fields are not visible to the users
       czar:                  short-text;     // person
       bboard:                short-text;     // person
       notify_list:           list-of short-text;     // list-of
                                                      // person
       tempShortText:         short-text;
}

////////////////////////////////////////////////////////////////////
// Transition Section
////////////////////////////////////////////////////////////////////

transitions {
    help {
       help-title 'Transition Overview';
       short-help-title 'Transitions';
       help-text'
To perform operations in the RTS applications, you choose the desired
editing mode from the Modes menu, edit the appropriate fields, and
click the <apply command> button (the third from the left in the
control area) to execute the operation or the Cancel button to void
it.

The RTS process begins with submitting a request, using either the
SUBMIT_BUG or SUBMIT_RFE transition, which puts the request in the
AWAITING_RESPONSE state.  The next step is screening the request.  Use
REJECT if the request is invalid, DEFER to postpone the fix, DUPLICATE
if an earlier request covers the suggestion, or RESOLVE to indicate
that the request has been executed.

At this point, the request is in the AWAITING_APPROVAL state.  The
approver can issue a REDO if the request has not been satisfied, which
returns the request to the AWAITING_RESPONSE state. If the fix is
satisfactory, then the approver issues the APPROVE transition which
takes the request to the CLOSED state. When a DEFERRed request is
APPROVEd, it enters the CLOSED state, but will be REOPENed by the
owner later.
```

```
After a waiting period, the tracking system administrator removes the
request from the database using the DELETE transition.';
    };

////////////////////////////////////////////////////////////////////
// SUBMIT_BUG Transition
////////////////////////////////////////////////////////////////////

    SUBMIT_BUG(=>AWAITING_RESPONSE) {
       help {
           help-title 'SUBMIT_BUG Transition';
           short-help-title 'SUBMIT_BUG';
           help-text'
SUBMIT_BUG creates a new request of type BUG.

Prior state: non-existent          New state: AWAITING_RESPONSE
Default fields:

1. The Date field is set to the current date.
2. The Type field is set to BUG.
3. The Submitter field is set to the current user.
4. The Summary field is set to the first line of the description.

Rule requirements:

1. The description field must have an entry.

Actions:

1. The owner is set to the entered value or if not entered to the
project manager if known or otherwise to the tracking system
administrator.';
       };
       rules {
           description.isSet;
           submit_date.is(submit_date.setDefault('now'));
           type.is(type.setDefault(BUG));
           submitter.is(submitter.setDefault($USER.value));
           summary.is(summary.setDefault(description.value));
           // Expect some action in one month
           due_date.is(due_date.setDefault('now +30:00:00:00'));
       }
```

```
        actions {
            bboard.setValue(BBOARD);
            // If the owner is not set but a project was entered, then
            // set the owner to the project manager.  If no project
            // was entered, then set owner to the bug board.
            tempShortText.
                setValue(execFilter(
'echo "select manager from project where name = \'$project\';"'));
            owner.setValue(owner.isSet ?
                owner.value :
                (project.isSet ?
                    owner.setValue(execSelect(tempShortText.value)) :
                    owner.setValue(execFilter('/bin/echo $bboard'))));
            tempShortText.setValue('');
        }
    }

/////////////////////////////////////////////////////////////////////
// SUBMIT_RFE Transition
/////////////////////////////////////////////////////////////////////

    SUBMIT_RFE(=>AWAITING_RESPONSE) {
        help {
            help-title 'SUBMIT_RFE Transition';
            short-help-title 'SUBMIT_RFE';
            help-text'
SUBMIT_RFE creates a new request of type RFE.

Prior state: non-existent          New state: AWAITING_RESPONSE

Default fields:

1. The Date field is set to the current date.
2. The Type field is set to RFE.
3. The Submitter field is set to the current user.
4. The Summary field is set to the first line of the description.

Rule requirements:

1. The description field must have an entry.
```

```
Actions:

1. The owner is set to the entered value or if not entered to the
   project manager if known or otherwise to the tracking system
   administrator.';
        };

        rules {
            description.isSet;
            submit_date.is(submit_date.setDefault('now'));
            type.is(type.setDefault(RFE));
            submitter.is(submitter.setDefault($USER.value));
            summary.is(summary.setDefault(description.value));
            // Expect some action in one month
            due_date.is(due_date.setDefault('now +30:00:00:00'));
        }
        actions {
            bboard.setValue(BBOARD);
            // If the owner is not set but a project was entered, then
            // set the owner to the project manager.  If no project
            // was entered, then set owner to the bug board.
            tempShortText.
                setValue(execFilter(
'echo "select manager from project where name = \'$project\';"'));
            owner.setValue(owner.isSet ?
                owner.value :
                (project.isSet ?
                    owner.setValue(execSelect(tempShortText.value)) :
                    owner.setValue(execFilter('/bin/echo $bboard'))));
            tempShortText.setValue('');
        }
    }

////////////////////////////////////////////////////////////////////
// ASSIGN Transition
////////////////////////////////////////////////////////////////////

    ASSIGN(AWAITING_RESPONSE=>AWAITING_RESPONSE) {
        help {
            help-title 'ASSIGN Transition';
            short-help-title 'ASSIGN';
            help-text'
ASSIGN establishes the owner and a due date for the request.
```

```
Prior state: AWAITING_RESPONSE     New state: AWAITING_RESPONSE

Default fields: none

Rule requirements:

1. There must be an entry in the Owner field.
2. There must be an entry in the Due Date field.

Actions: none';
       };
       rules {
           owner.isSet;
           due_date.isSet;
       }
    }

////////////////////////////////////////////////////////////////////
// FORWARD Transition
////////////////////////////////////////////////////////////////////
    FORWARD(AWAITING_RESPONSE=>AWAITING_RESPONSE) {
       help {
           help-title 'FORWARD Transition';
           short-help-title 'FORWARD';
           help-text'
FORWARD establishes a change in ownership of the request.

Prior state: AWAITING_RESPONSE     New state: AWAITING_RESPONSE

Default fields: none

Rule requirements:

1. The Owner field must be changed.

Actions: none';
       };
rules {
           owner.changed;
       }
    }
```

```
////////////////////////////////////////////////////////////////////
// RESOLVE Transition
////////////////////////////////////////////////////////////////////

    RESOLVE(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
       help {
           help-title 'RESOLVE Transition';
           short-help-title 'RESOLVE';
           help-text'
RESOLVE is issued by the request owner to indicate that the request
has been executed and is ready for approval.

Prior state: AWAITING_RESPONSE     New state: AWAITING_APPROVAL

Default fields: none

Rule requirements:

1. There must be an entry in the Resolution field.
2. There must be one or more valid files entered in the Resolved in
   field.

Actions:

1. The Recommendation field is set to RESOLUTION.';
       };
       rules {
           resolution_description.isSet;
           resolved_in.isSet;
       }
       actions {
           recommendation.setValue(RESOLUTION);
       }
    }

////////////////////////////////////////////////////////////////////
// REJECT Transition
////////////////////////////////////////////////////////////////////

    REJECT(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
       help {
           help-title 'REJECT Transition';
           short-help-title 'REJECT';
           help-text'
```

```
REJECT is issued by the request owner to indicate that the request is
not considered and needs to be confirmed (through the APPROVE
transition) as such by the approver.

Prior state: AWAITING_RESPONSE     New state: AWAITING_APPROVAL

Default fields: none

Rule requirements:

1. There must be an entry in the Resolution field.

Actions:

1. The Recommendation field is set to REJECTION.';
       };
       rules {
           resolution_description.isSet;
       }
       actions {
           recommendation.setValue(REJECTION);
       }
    }

////////////////////////////////////////////////////////////////////
// DEFER Transition
////////////////////////////////////////////////////////////////////

    DEFER(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
       help {
           help-title 'DEFER Transition';
           short-help-title 'DEFER';
           help-text'
DEFER is issued by the request owner to indicate that the request
should be postponed to the suggested date. This postponement needs to
be confirmed (through the APPROVE transition) by the approver.

Prior state: AWAITING_RESPONSE     New state: AWAITING_APPROVAL

Default fields: none

Rule requirements:

1. There must be an entry in the Resolution field.
2. There must be an entry in the Reopen date field.
```

```
Actions:

1. The Recommendation field is set to DEFERRAL.';
       };
       rules {
           resolution_description.isSet;
           reopen_date.isSet;
       }
       actions {
           recommendation.setValue(DEFERRAL);
       }
    }

////////////////////////////////////////////////////////////////////
// DUPLICATE Transition
////////////////////////////////////////////////////////////////////

    DUPLICATE(AWAITING_RESPONSE=>AWAITING_APPROVAL) {
       help {
           help-title 'DUPLICATE Transition';
           short-help-title 'DUPLICATE';
           help-text'
DUPLICATE is issued by the request owner to indicate that an earlier
request made the same suggestion and that this request is unnecessary.
The DUPLICATE transition needs to be confirmed (through the APPROVE
transition) by the approver.

Prior state: AWAITING_RESPONSE     New state: AWAITING_APPROVAL

Default fields: none
Rule requirements:

1. There must be an entry in the Resolution field.
2. There must be an entry in the Dup of field, indicating the earlier
   transition that was duplicated.

Actions:

1. The Recommendation field is set to DUPLICATION.';
       };
       rules {
           resolution_description.isSet;
           is_duplicate_of.isSet;
       }
```

```
        actions {
            recommendation.setValue(DUPLICATION);
        }
    }

////////////////////////////////////////////////////////////////////
// NOTIFY Transition
////////////////////////////////////////////////////////////////////

    NOTIFYME(=>) {
        help {
            help-title 'NOTIFY Transition';
            short-help-title 'NOTIFY';
            help-text'
NOTIFYME adds a new user to the list of users to be informed when
changes occur to this request.

Prior state: any                   New state: any

Default fields: none

Rule requirements: none

Actions:

1. The name in the Notify field is added to the list of interested
   parties for this request.';
        };
        actions {
            interested_parties.append($USER.text);
        }
    }
////////////////////////////////////////////////////////////////////
// REDO Transition
////////////////////////////////////////////////////////////////////
REDO(AWAITING_APPROVAL=>AWAITING_RESPONSE) {
        help {
            help-title 'REDO Transition';
            short-help-title 'REDO';
            help-text'
An approver issues a REDO when a request has not been adequately
satisfied so that the request is returned to the owner.

Prior state: AWAITING_APPROVAL     New state: AWAITING_RESPONSE
```

```
Default fields: none

Rule requirements: none

Actions: none';
       };
       rules {
       }
       actions {
       }
    }
//////////////////////////////////////////////////////////////////////
// EDIT Transition
//////////////////////////////////////////////////////////////////////
    EDIT(=>) {
       help {
           help-title 'EDIT Transition';
           short-help-title 'EDIT';
           help-text'
EDIT lets you change the current values of fields in the request.
Prior state: any               New state: any

Default fields: none

Rule requirements: none

Actions: none';
       };
       rules {
       }
       actions {
       }
    }
//////////////////////////////////////////////////////////////////////
// APPROVE Transition
//////////////////////////////////////////////////////////////////////

    APPROVE(AWAITING_APPROVAL=>CLOSED) {
       help {
           help-title 'APPROVE Transition';
           short-help-title 'APPROVE';
           help-text'
APPROVE is only accessible to those authorized to approve requests.
APPROVE changes the state of the request from AWAITING_APPROVAL to
CLOSED.  Prior state: AWAITING_APPROVAL        New state: CLOSED
```

```
Default fields: none

Rule requirements: none

Actions:

1. The Approver field is set to the entered value or if none, to the
   current user.
2. The Close date field is set to the current date.';
       };
       actions {
           approver.setValue(approver.isSet ?
               approver.value :
               $USER.value);
           close_date.setValue('now');
       }
    }

////////////////////////////////////////////////////////////////////
// REOPEN Transition
////////////////////////////////////////////////////////////////////

    REOPEN(CLOSED=>AWAITING_RESPONSE) {
       help {
           help-title 'REOPEN Transition';
           short-help-title 'REOPEN';
           help-text'
REOPEN is used to open a request that has been deferred.

Prior state: CLOSED            New state: AWAITING_RESPONSE

Default fields: none

Rule requirements: none

Actions: none ';
       };
    }
```

```
////////////////////////////////////////////////////////////////////
// DELETE Transition
////////////////////////////////////////////////////////////////////

    DELETE(CLOSED=>DELETED) {
       help {
           help-title 'DELETE Transition';
           short-help-title 'DELETE';
           help-text'
DELETE is used to remove a request from the request database. It is
valid for the tracking system administrator only.

Prior state: CLOSED              New state: DELETED

Default fields: none

Rule requirements:

1. Current user must be the tracking system administrator.

Actions: none ';
       };
       rules {
            $USER.is(CZAR);
       }
    }

////////////////////////////////////////////////////////////////////
// Global Rules and Actions for All Transitions
////////////////////////////////////////////////////////////////////

    rules {
       help {
           help-title 'Global Rules and Actions';
           short-help-title 'Global Rules and Actions';
           help-text'

These rules and actions are applied to all transitions.

Rule requirements:

1. Report number cannot be changed after the request has been
   submitted.
2. The Recommendation field cannot be edited.  It is set
   automatically.
Actions:

1. Add the owner and submitter to the list of interested parties to be
   notified when changes occur to the request.';
       };

       // The report_number can only be changed as part of
       // submission.
       $TRANSITION.is(SUBMIT_BUG) ||
       $TRANSITION.is(SUBMIT_RFE) ||
       !report_number.changed;

       !recommendation.changed;
    }

    actions {
       // If the report_number is not set, then set it to the value
       // of the $ENTITY_ID field.
       // report_number.setValue(report_number.isSet ?
       //      report_number.value : $ENTITY_ID.value);
       // Execute the notifier to send mail as appropriate.
       //
       czar.setValue(CZAR);

       notify_list.setValue(interested_parties.value);
       notify_list.append(owner.value);
       notify_list.append(submitter.value);
       execCommand('/usr/local/lib/rts_notify');
    };
}
views {
    help      {
           help-title 'rtsquery Views';
           short-help-title 'rtsquery Views';
           help-text
'The rtsquery view lets you access the request database.  It has four
main areas. From top to bottom these are:

       * menu bar - for accessing menus
       * control bar - select editing mode from the mode menu, enter
         data in the appropriate fields, and complete transaction by
         clicking the command (third button from left). Right four
         buttons are for selecting requests in query results area.
       * query results area - lists requests resulting from a query
       * request form area - contains detailed request information
```

```
The rtsfiles view is an auxiliary view of the rtsquery application.
It contains a request form area with three fields.  These fields list
the files associated with a request.  For more help, look up the help
cards for the individual fields. ';
    };
    RTSQuery(){
        display () {
           control-bar( ) {
               help {
                   help-title 'Control Bar';
                   short-help-title 'Control Bar';
                   help-text'
The control bar consists of the Modes menu, the Cancel and <apply
command> buttons, and the four query list control buttons.

To perform operations in the RTS applications, you choose the desired
mode from the Modes menu, edit the appropriate fields, and click the
<apply command> button (the third from the left in the control area)
to execute the operation or the Cancel button to void it.  The <apply
command> button label changes as the mode, selected from the Modes
menu, changes.

The list control buttons control the selection of requests in the
query results area.';

               };
               // Include all transitions
               transitions;
           };
       };
       qresults() {
           index type, '#' $ENTITY_ID, $STATE, owner, summary;
       };

       fourColumn: display() {
           row{'Report #:' $ENTITY_ID,
               'Status:' $STATE,
               'Type:' type,
               'Submitter:' submitter};
           row{'Date:' submit_date,
               'Recommend:' recommendation,
               'Project:' project,
               'Priority:' priority};
```

```
                oneRowList:
                row{'Owner:' owner,
                    'System:' system,
                    'Notify:' interested_parties,
                    'Due Date:' due_date};
                row{'Close Date:' close_date,
                    'Reopen Date:' reopen_date,
                    'Approver:' approver,
                    'Dup of:' is_duplicate_of};
                row{'Summary:' summary};
            }
            display() {
                row{'Description:' ' '};
                fourRowLongText:
                row{description};
            }
            display() {
                row{'Resolution:' ' '};
                fourRowLongText:
                row{resolution_description};
            }
        }
        RTSFiles(type ' #' $ENTITY_ID ' ' $STATE ' ' owner ' ' summary) {
            display() {
                row{'Found in:' ' ', ' '};
                row{found_in};
            }
            display() {
                row{'Resolved in:' ' ', ' '};
                row{resolved_in};
            }
            display() {
                row{'Fixed Releases:' ' ',,,,};
                row{fixed_releases};
            }
        }
    }
```

rtsapprove.pdl

```
//////////////////////////////////////////////////////////////////////
//
// File:        approve.pdl
// Description:  RTS approval pdl
//              This pdl file defines the report approval app's GUI
// Author:       Pete Orelup
// Created:      Fri Apr 10 09:29:49 PDT 1992
// Language:     PDL
//
// (C) Copyright 1992, Silicon Graphics, Inc.
//
//  Permission to use, copy, modify, and distribute this software for
//  any purpose except publication and without fee is hereby granted,
//  provided that the above copyright notice appear in all copies of
//  the software.
//
//////////////////////////////////////////////////////////////////////
//

views {
    RTSApprove(){
       help {
           help-title 'rtsapprove View';
           short-help-title 'rtsapprove View';
           help-text
'The rtsapprove window is a supplementary application for approving
fixed requests.  The transitions available in rtsapprove are: APPROVE,
NOTIFYME, REDO, and EDIT.

It has four main areas. From top to bottom these are:

    * menu bar - for accessing menus
    * control bar - select editing mode from the mode menu, enter data
      in the appropriate fields, and complete transaction by clicking
      the command (third button from left). Right four buttons are for
      selecting requests in query results area.
    * query results area - lists requests resulting from a query
    * request form area - contains detailed request information
      relevant for responding to requests. This includes a Resolution
      field for explaining the fix to the request and a Resolved in
      field that identifies the files that have changed as a result of
      the request.';
       };
```

```
        control-bar( ) {
            help {
                help-title 'Control Bar';
                short-help-title 'Control Bar';
                help-text'
The control bar consists of the Modes menu, the Cancel and <apply
command> buttons, and the four query list control buttons.

To perform operations in the RTS applications, you choose the desired
mode from the Modes menu, edit the appropriate fields, and click the
<apply command> button (the third from the left in the control area)
to execute the operation or the Cancel button to void it.  The <apply
command> button label changes as the mode, selected from the Modes
menu, changes.

The list control buttons control the selection of requests in the
query results area.';
            };
            transitions APPROVE, NOTIFYME, REDO, EDIT;
        };
         display () {
            qresults() {
                index type, '#' $ENTITY_ID, $STATE, owner, summary;
            };
        };
        display() {
            row{'Report #:' $ENTITY_ID,
                'Status:' $STATE};
            row{'Submitter:' submitter,
                'Date:' submit_date};
            row{'Type:' type,
                'Recommend:' recommendation};
            row{'Project:' project,
                'Priority:' priority};
            oneRowList:
            row{'System:' system,
                'Notify:' interested_parties};
            row{'Owner:' owner,
                'Due Date:' due_date};
            row{'Duplicate of:' is_duplicate_of,
                'Re-Open Date:' reopen_date};
            row{'Approver:' approver, };
            row{'Summary:' summary};
        }
        display() {
            row{'Description:' ' '};
            fourRowLongText:
            row{description};
        }
        display() {
            row{'Resolution:' ' '};
            fourRowLongText:
            row{resolution_description};
        }
        display() {
            row{'Resolved in:' ' '};
            row{resolved_in};
        }
    }
}
```

rtsrespond.pdl

```
//////////////////////////////////////////////////////////////////////
//
// File:         respond.pdl
// Description:  RTS response pdl
//              This pdl file defines the report response app's GUI
// Author:       Pete Orelup
// Created:      Fri Apr 10 09:29:49 PDT 1992
// Language:     PDL
//
// (C) Copyright 1992, Silicon Graphics, Inc.
//
//  Permission to use, copy, modify, and distribute this software for
//  any purpose except publication and without fee is hereby granted,
//  provided that the above copyright notice appear in all copies of
//  the software.
//
///////////////////////////////////////////////////////////////////////////////
//

views {
    RTSRespond(){
       help {
           help-title 'rtsrespond View';
           short-help-title 'rtsrespond View';
           help-text
'The rtsrespond window is a supplementary application for responding
to requests in the request database.  It is intended for request
owners. The transitions available in rtsrespond are: NOTIFYME,
FORWARD, EDIT, DEFER, RESOLVE, REJECT, and DUPLICATE.

It has four main areas. From top to bottom these are:

    * menu bar - for accessing menus
    * control bar - select editing mode from the mode menu, enter data
      in the appropriate fields, and complete transaction by clicking
      the command (third button from left). Right four buttons are for
      selecting requests in query results area.
    * query results area - lists requests resulting from a query
    * request form area - contains detailed request information
      relevant for responding to requests. This includes a Resolution
      field for explaining the fix to the request and a Resolved in
      field that identifies the files that have changed as a result of
      the request.';
```

```
       };
       control-bar( ) {
           help {
               help-title 'Control Bar';
               short-help-title 'Control Bar';
               help-text'
The control bar consists of the Modes menu, the Cancel and <apply
command> buttons, and the four query list control buttons.

To perform operations in the RTS applications, you choose the desired
mode from the Modes menu, edit the appropriate fields, and click the
<apply command> button (the third from the left in the control area)
to execute the operation or the Cancel button to void it.  The <apply
command> button label changes as the mode, selected from the Modes
menu, changes.

The list control buttons control the selection of requests in the
query results area.';
           };
           transitions NOTIFYME, FORWARD, EDIT, DEFER, RESOLVE,
                    REJECT, DUPLICATE;
       };
       display () {
           qresults() {
               index type, '#' $ENTITY_ID, $STATE, owner, summary;
           };
       };
       display() {
           row{'Report #:' $ENTITY_ID,
               'Status:' $STATE};
           row{'Submitter:' submitter,
               'Date:' submit_date};
           row{'Type:' type,
               'Recommend:' recommendation};
           row{'Project:' project,
               'Priority:' priority};
           oneRowList:
           row{'System:' system,
               'Notify:' interested_parties};
           row{'Owner:' owner,
               'Due Date:' due_date};
           row{'Duplicate of:' is_duplicate_of,
               'Re-Open Date:' reopen_date};
           row{'Approver:' approver, };
           row{'Summary:' summary};
```

```
        }
        display() {
            row{'Description:' ' '};
            fourRowLongText:
            row{description};
        }
        display() {
            row{'Resolution:' ' '};
            fourRowLongText:
            row{resolution_description};
        }
        display() {
            row{'Resolved in:' ' '};
            row{resolved_in};
        }
    }
}
```

rtssubmit.pdl

```
////////////////////////////////////////////////////////////////////////
//
// File:          submit.pdl
// Description:   RTS submittal pdl
//               This pdl file defines the report submission app's GUI
// Author:        Pete Orelup
// Created:       Fri Apr 10 09:29:49 PDT 1992
// Language:      PDL
//
// (C) Copyright 1992, Silicon Graphics, Inc.
//
//  Permission to use, copy, modify, and distribute this software for
//  any purpose except publication and without fee is hereby granted,
//  provided that the above copyright notice appear in all copies of
//  the software.
//
////////////////////////////////////////////////////////////////////////
//

views {
    RTSSubmit(){
       help {
           help-title 'rtssubmit View';
           short-help-title 'rtssubmit View';
           help-text '
The rtssubmit window is a supplementary application for creating new
requests in the request database. The transitions available in
rtssubmit are: SUBMIT_BUG, SUBMIT_RFE, ASSIGN, FORWARD, NOTIFYME, and
EDIT.

It has four main areas. From top to bottom these are:

    * menu bar - for accessing menus
    * control bar - select editing mode from the mode menu, enter data
      in the appropriate fields, and complete transaction by clicking
      the command (third button from left). Right four buttons are for
      selecting requests in query results area.
    * query results area - lists requests resulting from a query
    * request form area - contains detailed request information
      relevant for submitting requests. This includes a Found in field
      for identifying the location of the request.';
       };
```

```
        control-bar( ) {
            help {
                help-title 'Control Bar';
                short-help-title 'Control Bar';
                help-text'
The control bar consists of the Modes menu, the Cancel and <apply
command> buttons, and the four query list control buttons.

To perform operations in the RTS applications, you choose the desired
mode from the Modes menu, edit the appropriate fields, and click the
<apply command> button (the third from the left in the control area)
to execute the operation or the Cancel button to void it.  The <apply
command> button label changes as the mode, selected from the Modes
menu, changes.

The list control buttons control the selection of requests in the
query results area.';
            };
            transitions SUBMIT_BUG, SUBMIT_RFE, ASSIGN, FORWARD,
            NOTIFYME, EDIT;
        };
         display () {
            qresults() {
                index type ' #' $ENTITY_ID ' ', $STATE ' ', owner ' ',
                        summary;
            };
        };
        display() {
            row{'Report #:' $ENTITY_ID,
                'Status:' $STATE};
            row{'Submitter:' submitter,
                'Date:' submit_date};
            row{'Type:' type,
                'Recommend:' recommendation};
            row{'Project:' project,
                'Priority:' priority};
            oneRowList:
            row{'System:' system,
                'Notify:' interested_parties};
            row{'Owner:' owner,
                'Due Date:' due_date};
            row{'Summary:' summary};
        }
        display() {
            row{'Description:' ' '};
            fourRowLongText:
            row{description};
        }
        display() {
            row{'Found in:' ' '};
            oneRowList:
            row{found_in};
        }
    }
}
```

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1664-020.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389