A Specification of FILLIN Using the DARWIN Requirements Model

Steven Wartik

Computer Science Report No. RM-85-02
June 3, 1985

Author's address: Department of Computer Science, Thornton Hall, University of Virginia, Charlottesville, VA 22903
CSnet: spw@virginia
UUCP: uvacs!spw

Abstract

To be successful, the DARWIN project must be proven useful on useful systems. This paper presents a specification of such a system: FILLIN, a form-filling user interface tool. FILLIN is a non-trivial tool that has been used for some time by employees of the TRW Defense Systems Group. However, it is to be re-implemented at UVA, with some changes to functionality and interface; as such it is an ideal candidate for a specification. The specification is considered a reasonable approximation of what one might expect from DARWIN: both customer and designer models are used, with varying degrees of formality. A concluding section summarizes the experiment.

CONTENTS

1. INTRODUCTION

The DARWIN model of software requirements for interactive systems [9] has been applied to a wide range of systems. These include the monitor for a simple relational database management system patterned after INGRES [7], a forms management user interface tool called FILLIN [5], and the Unix[1] [4] library "CURSES" [1]. These requirements, while useful, have been written to facilitate experiments with various parts of the DARWIN model; none encompasses both customer and designer entities.

We are planning to rewrite the FILLIN system in the near future. Many of the changes are internal, and are intented solely to improve system maintenance and performance. Some of the changes are to the user interface and the form language, however. Accordingly, a requirements specification for FILLIN is needed. As FILLIN is a highly interactive system, the DARWIN specification model is a natural choice. Moreover, this is an excellent opportunity to write a full-sized DARWIN specification. Fillin tool is a non-trivial tool that will require several man-months of effort to implement, but it is highly modular and amenable to DARWIN's packaging techniques. It is to be implemented in C, not Ada;[2] however, the lessons learned from this example are valuable enough to justify the effort. Moreover, Ada technology is still too primitive; FILLIN's usefulness would be limited were Ada to be the target language.

This document presents a specification of FILLIN. It is an example of how a real DARWIN specification might appear on paper. A paper document is, of course, no substitution for the tool that should support FILLIN; for example, the designer has not run consistency checks, and the developer cannot make automated queries to help locate entities, nor can he automatically convert portions of the requirements into design. However, the document gives a good idea of the readability of DARWIN requirements. It also contains numerous examples of how customer concepts can be related to designer concepts so that both parties can understand, to some extent, each other's work. This is an important feature of DARWIN.

Each of the next sections of this document contains one "level" of the requirements. Section 2 contains requirements written in the customer model. Section 3 contains a revision of the information in section 2, at a level of formality between the customer and designer models. Section 4 contains a designer-level specification of FILLIN.

This is not a complete specification: certain designer entities are deliberately left undefined. The DARWIN tool base for analyzing designer concepts is not implemented, and checking correctness of functionality by hand is too difficult to warrent investing the time. The objective of this document is instead to present a large example that establishes relationships between customer and designer objects. Moreover, this example is sufficiently complete to permit an implementation, and we anticipate that the new version of FILLIN will in fact be based on it.

1. Unix is a registered trademark of AT&T Bell Laboratories.

2. Ada is a registered trademark of the United States Government, Ada Joint Projects Office.

2. CUSTOMER REQUIREMENTS

This section contains the requirements for fillin as written in the customer's model. There are three parts: term definitions, conversations, and event definitions.

Term definitions are listed in no particular order; this suggests the random order in which they occurred to the author. As pointed out in [9], many orderings for terms (and events) are possible, including alphabetical and packaged. The next version of the requirements will demonstrate this by using the latter. However, an alphabetical listing of terms follows the definitions.

The customer requirements use two conversations. The first is the fillin conversation, which is the main conversation and defines the system. The second is the AddText conversation, which specifies how text is entered in data fields. Much is deliberately left unspecified at this point, such as the format of form template files and form data files; more significantly, the user interface is fairly vague. Such details are as yet unimportant, and will become more precise in later models.

The event definitions are listed in alphabetical order, for easy reference. Note the event "fillin," which is an event derived from the fillin conversation. This is an example of DARWIN bottom-up development.

2.1 TERM DEFINITIONS

Form

A form is an abstract object that is intended to be equivalent to a user's perception of a paper form, namely:

1. A list of "data fields" that users may fill and edit, and that tools may access to use the data for whatever application is at hand. This list is ordered, i.e., there are $n>0$ distinct fields whose ordinality is fixed.
2. Accompanying text (called "trim") that identifies these data areas for the user.
3. A "current data field" (meaningful only when the form is being "displayed").

Forms may be treated as "form images," or may be converted into "form data files," or may processed directly by programs. A form must have at least one data field.

Form Object

A "form object" is actually equivalent to a form. It is used as a data type name, for two reasons. First, it avoids conflict with "form", which is useful as a parameter and environment object name. Second, it suggests that the object in question is just that: an object in a conversation's environment.

Form Image

A form image is the electronic image of a form, displayed on a user's screen. Hence, it is that part of a form that the user sees. A form image, being equivalent to a paper form, is inherently a two-dimensional object, and must be viewed on a VDT or equivalent device.

Form Template File

A form template file contains an abstract representation of a form image. This representation is used by fillin to determine the contents of a form: its data fields and trim.

Form Data File

Users enter data in data fields. This data may be stored in form data files by fillin.

Trim

A form consists of two parts: trim and data fields. Trim is text that exists solely for the user's benefit, i.e., it is not part of the actual form data, nor is it part of the identification of this data. Trim is often separator lines, such as strings of "=" characters.

Data Field

A form consists of two parts: trim and data fields. A data field may be considered as part of both the form template file and the form image. If the former, then it is a specification of the data field that appears in the form image. If the latter, then it is an area of the form where data may be entered. A data field in a form template file consists of specifications of:

1. The type of data (i.e., integer, string).
2. The width of the data entry area.
3. The textual identification that is to appear in the form image (to identify the data entry area to the user), and information stating where this text is to be placed in relation to the data entry area.

There are two types of data fields: single-line data fields and multi-line data fields.

Single-Line Data Field	A single-line data field is one that occupies exactly one line on the screen. This type of data field is most useful for entering items such as names or dates that have a predetermined fixed maximum length.
Multi-Line Data Field	A multi-line data field occupies at least two lines on the screen, and may occupy the entire screen if necessary. It may be filled with arbitrary-length text strings. Furthermore, although the data field has a fixed length on the screen, data stored in it has no maximum length; if the data is too large, the data field is scrolled as necessary.
Data Entry Area	"Data entry area" refers to an area in the form image where data may be typed (hence, it is a portion of a data field).
Filling	Filling a form is the operation of a user entering data in an new form.
Editing	Editing a form is the operation of a user entering data in an existing form.
Merging	Merging a form and a form data file is the operation of placing the contents of the form data file in the appropriate fields of the form. This works as follows. A form data file should contain as many fields as the form that it is being merged with. Entry *i* of the form data file is placed in field *i* of the form. However, if the entry in the data file has a name, then it is to be placed in the data field of the form with the same name.
Label	A data field may contain a "label." A label is a textual string that is solely for the user's benefit, and identifies the purpose of the data field. For example, a data field that is to contain a last name would likely have a label "Last Name:". This differs from trim in that the label is part to the data field.
Label Width	A label has a width. This width may be explicit or implicit. An explicit width is specified by giving the width attribute in the form template file. If this width is longer than that of the label text, then the label is blank-padded on the right. If this width is shorter than that of the label text, then the label is truncated on the right. If no width is specified in the form template file, then the label's width is that of its text.
Label Position	A label mays have a position relative to the data entry area. If unspecified, then this position is to the left, with the right edge of the label one space away from the (upper) left edge of the data entry area. If specified, then this position is given as an (x,y) pair, and the label's left edge is relative to the data entry area's upper left edge, at distance (x,y). Here positive x is towards the left margin, and positive y is towards the top of the form.
Echo	To "echo" a character that is being read from the user is to display it at the cursor's current position, advancing the cursor after reading it. To not echo a character is to read the character without either moving the cursor or displaying the character.
Cursor	The cursor is that icon which users see on the terminal's screen when viewing the form image that indicates the current cursor position. In fillin, it also indicates which is the current data field.

Current Data Field	At any time when a form image appears on the screen, a "current data field" is defined. This field is the field that the user may next edit, fill, etc., or may move from. This field is identified by having the cursor positioned at it.
Scroll	To scroll a multi-line field is to move all the text in it up or down one line without changing the text in the remainder of the form image.
Word	In the context of a data field's value, a "word" is the longest possible string of non-blank characters, starting from the current position and ending at the next blank (or beginning of end of value) in whichever direction is appropriate.
Blank Character	In the context of a data field's value, a "blank character" is an ASCII space, newline, or carriage return character.
Standard Input	The "standard input" refers to the data stream that represents the default input for a program. This stream is used by fillin to receive commands. Because fillin is intended to be an interactive tool, it must be a terminal's keyboard. *N.B.* to be consistent with Ada, "standard_input" is a synonym for standard input.
Standard Output	"Standard output" refers to the data stream that represents the default output for a program. This stream is where fillin displays the form. Because fillin is intended to be an interactive tool, it must be a terminal's screen. *N.B.* to be consistent with Ada, "standard_output" is a synonym for standard output.

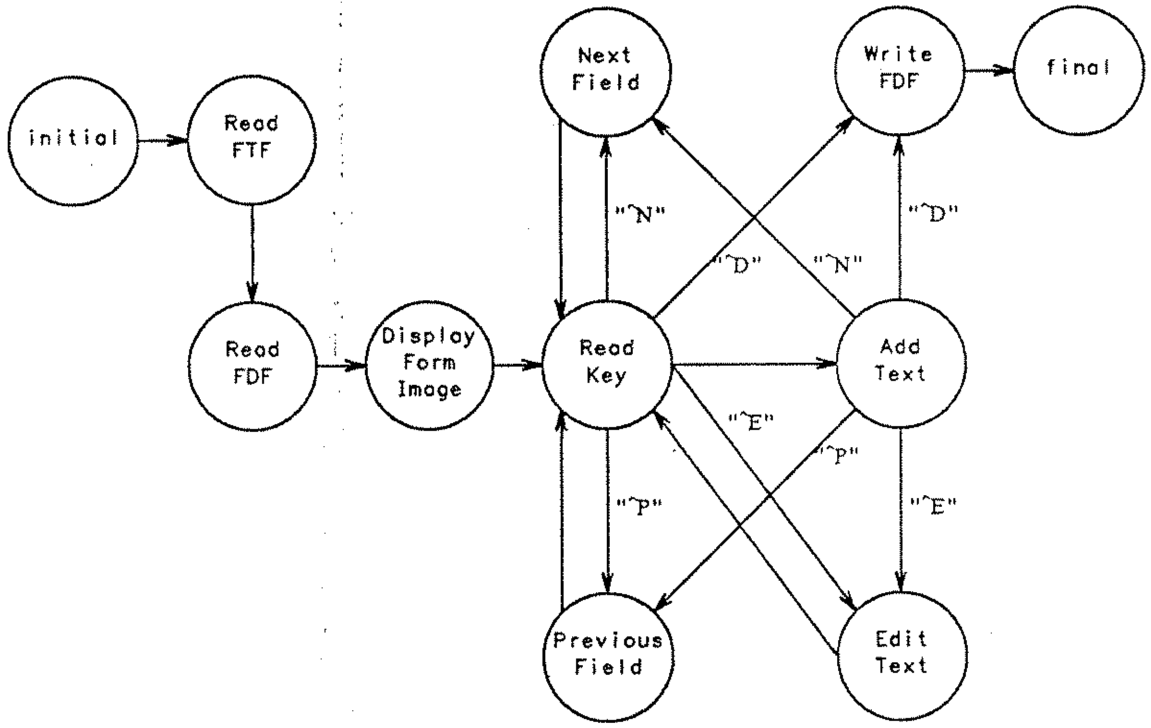
2.2 ALPHABETICAL TERM LISTING

The following is an alphabetically-ordered listing of the currently defined terms:

Blank Character, Current Data Field, Cursor, Data Entry Area, Data Field, Echo, Editing, Filling, Form, Form Data File, Form Image, Form Object, Form Template File, Label, Label Position, Label Width, Merging, Multi-Line Data Field, Scroll, Single-Line Data Field, Standard Input, Standard Output, Trim, Word

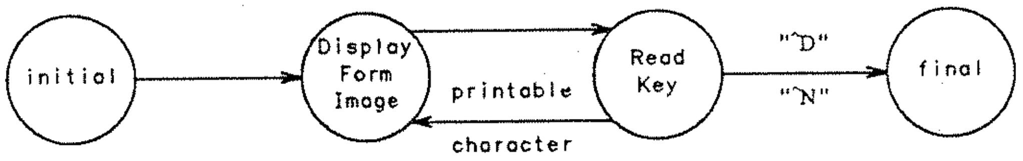
2.3 CONVERSATIONS

2.3.1 Fillin



```
CONVERSATION fillin IS

ENVIRONMENT:
        Name:  form
        Type:  form object
        English Description:
                This object holds the form abstraction being manipulated.

        Name:  form template file name
        Type:  string
        English Description:
                This object holds the name of the file containing the form template.

        Name:  form data file name
        Type:  string
        English Description:
                This object holds the name of the file from where form data is to be read,
                and to where form data is to be written.


TRANSITIONS:

        initial (argument list) CAUSES
                Determine, from the argument list, the names of the form template file and
                the form data file.  Place them in the environment.
                AND THEN ReadFTF(form template file name)

        ReadFTF ( form ) CAUSES ReadFDF(form data file name, form)

        ReadFDF ( form ) CAUSES
                Place the form in the environment.
                AND THEN DisplayFormImage(form)

        DisplayFormImage CAUSES ReadKey

        ReadKey(KeyRead) CAUSES
                CASE KeyRead IN
                  WHEN control-d => WriteFDF(form)
                  WHEN control-n => NextField(form)
                  WHEN control-p => PrevField(form)
                  WHEN control-e => EditField(form)
                  WHEN a printable character => AddText(form, KeyRead)
                END CASE

        WriteFDF CAUSES final

        AddText(form, KeyRead) CAUSES
                CASE KeyRead IN
                  WHEN control-d => WriteFDF(form)
                  WHEN control-n => NextField(form)
                  WHEN control-p => PrevField(form)
                  WHEN control-e => EditField(form)
                END CASE

        EditText(form) CAUSES ReadKey
        NextField(form) CAUSES ReadKey

        PrevField(form) CAUSES ReadKey

NODE ReadKey IS
        DIVERSION
                DESCRIPTION:
                        This diversion component handles mis-typed editing keys.
                OCCURS WHEN
                        KeyRead is one of the user's backspace, line erase, or word erase
                        characters.
                MESSAGE
                        "You have not yet entered any text."
                RECOVER BY LOOPING

        DIVERSION
                DESCRIPTION:
                        This diversion component handles other non-printing characters.
                OCCURS WHEN
                        KeyRead is any non-printing character that is not a special key and
                        is also not one of the user's editing keys.
                MESSAGE
                        Ring the terminal's bell.
                RECOVER BY LOOPING

        DIVERSION
                DESCRIPTION:
                        This diversion component is for bad editing attempts.
                OCCURS WHEN
                        KeyRead is control-e and the current field is a single-line field.
                MESSAGE
                        "You may only edit multi-line fields."
                RECOVER BY LOOPING

END ReadKey

END fillin
```

2.3.2 AddText



"AddText" is used to perform the basic operation of filling text in one of the data field areas of a form image.

```
CONVERSATION AddText IS
PARAMETERS:
        IN PARAMETERS:
                FirstCharacter : TYPE IS character
                English Description:
                        "FirstCharacter" is a single character that the user typed that is to
                        be the first character of the text.
        OUT PARAMETERS:
                KeyRead : TYPE IS character
                English Description:
                        "KeyRead" is the last keystroke read, i.e., the key that caused text
                        entry to terminate.
        IN/OUT PARAMETERS:
                form : TYPE IS form object
                English Description:
                        "form" represents the form image.

ENVIRONMENT:
        Name:   form
        Type:   form object
        English Description:
                This object holds the form to which text is being added.

TRANSITIONS:

        initial(form, FirstCharacter) CAUSES
                Reset the data value of the form's current field to "FirstCharacter".
                AND THEN DisplayFormImage(form)

        DisplayFormImage CAUSES ReadKey

        ReadKey (keystroke) CAUSES
                CASE keystroke IN
                  WHEN control-d OR control-e OR control-n OR control-P =>
                        final (form, keystroke)
        WHEN user's erase-character key =>
            Remove the last character from the current field's data value.
            AND THEN DisplayFormImage (form)
        WHEN user's line-kill key =>
            The current field's data value becomes an empty string.
            AND THEN DisplayFormImage (form)
        WHEN user's erase-word key =>
            Remove the last "word" from the current field's data value.
            AND THEN DisplayFormImage (form)
        WHEN OTHERS =>
            The current field's data value becomes the old value, with the keys-
            troke appended.
            AND THEN DisplayFormImage (form)
        END CASE

NODE ReadKey IS
    DIVERSION
        DESCRIPTION:
            This diversion component handles mis-typed editing keys.
        OCCURS WHEN
            KeyRead is one of the user's backspace, line erase, or word erase
            characters, and the cursor is positioned at the beginning of the
            current data field.
        MESSAGE
            Ring the terminal's bell.
        RECOVER BY LOOPING

    DIVERSION
        DESCRIPTION:
            This diversion component handles non-printing characters.
        OCCURS WHEN
            KeyRead is not any non-printable character aside from those given
            above.
        MESSAGE
            Ring the terminal's bell.
        RECOVER BY LOOPING

    DIVERSION
        DESCRIPTION:
            This diversion component handles data that is longer than the width
            of the data field.
        OCCURS WHEN
            The current data field is a single-line data field, the existing data in
            the current data field is the width of the field, and KeyRead is a
            character that would cause text to be added.
        MESSAGE
            Ring the terminal's bell.
        RECOVER BY LOOPING
        DIVERSION
                DESCRIPTION:
                        This diversion component handles non-digit characters in integer
                        fields.
                OCCURS WHEN
                        The current data field is a single-line integer field, and the character
                        typed is not a digit.
                MESSAGE
                        "Integer field"
                RECOVER BY LOOPING

END ReadKey

END AddText
```

2.4 EVENT DEFINITIONS

This section defines all events used in the specification of fillin. Not all of these events are used in the customer-level specification. However, an event must be expressible at the customer's level of formality, as well as the designer's. To prove this point, we define all events in this section. To help the reader differentiate, those events that are *not* part of the customer requirements are marked with an asterisk. Not all the events are truly "understandable" by a customer, as some of them—e.g., CloseFile and OpenFile—refer to programming techniques a naive customer might not know. However, these events are well-commented and given in a format consistent with all others, which is an important aid to documentation.

The events are presented in alphabetical order, as that is deemed the easiest reference technique in lieu of automated aids. Future sections will also package events, along with types, functions, and conversations, according to purpose.

The events defined so far are:

AddText, CloseFile, DisplayFormImage, EditText, fillin, NextField, OpenFile, PrevField, ReadFDF, ReadFTF, ReadKey, WriteFDF

EVENT NAME: AddText

ENGLISH DESCRIPTION:

"AddText" is used to perform the basic operation of filling text in one of the data field areas of a form image.

PARAMETERS:

IN PARAMETERS:

FirstCharacter : TYPE IS character
English Description:
"FirstCharacter" is a single character that the user typed that is to be the first character of the text.

OUT PARAMETERS:

KeyRead : TYPE IS character
English Description:
"KeyRead" is the last keystroke read, i.e., the key that cause text entry to terminate.

IN/OUT PARAMETERS:

form : TYPE IS form object
English Description:
"form" represents the form image.

FILES:

standard_input is accessed, to read the user's data.
standard_output is accessed, to write the text being added.

SEMANTICS:

The user will type a string of text that is to be the new data for the current data field. Four restrictions apply. First, no non-printable characters are allowed. Second, the user's erase, word kill, and line kill characters are to be available for editing. Third, the text cannot be longer than the width of the field if the field is single line. Fourth, integer-typed fields may contain only integers. As each character is typed, it is to be displayed so the user sees the currently accumulated string of text.

For multi-line fields, where text size is unlimited, if the addition of text would move the cursor to the right of the data field's right margin, then the cursor is to be moved to the left margin of the next line. Moreover, a newline or carriage return character causes the current line to end in the data field and the cursor to be moved to the left margin of the next line in the screen image. If either of the above would cause the cursor to move below the bottom line of the data field, then the data field data entry area is to be scrolled one line so that the line at the top of the data field disappears and the bottom line is clear. The cursor will be placed at the data field's left margin of this line.

Text entry is to be terminated by typing any of the next-field, previous-field, or quit keys. If the data field is a multi-line field and has been scrolled, then the data in it is to be redisplayed such that the beginning of the data appears at the top line of the data field.

END AddText

EVENT NAME: CloseFile*

ENGLISH DESCRIPTION:

"CloseFile" closes an open file stream (see the OpenFile event).

PARAMETERS:

IN PARAMETERS:

file stream : TYPE IS file
English Description:
"file stream" is an open file stream.

FILES:

SEMANTICS:

The file stream referenced by the parameter is closed.

END CloseFile

EVENT NAME: DisplayFormImage

ENGLISH DESCRIPTION:

"DisplayFormImage" displays a form image on the user's screen, complete with whatever data is currently in the fields.

PARAMETERS:

IN PARAMETERS:

form : TYPE IS form object
English Description:
"form" is the form to be displayed.

FILES:

standard output

SEMANTICS:

The form is displayed as follows. Trim is displayed exactly as it appears in the form template file, and at the same position (accounting for horizontal data field offsets, as explained below). Data fields are slightly different, as they contain extra information, but the intent is still to display them at approximately the same place as they appear in the form template file. Thus, a data field that begins at position (x,y) in the form template file will be positioned approximately there. Its label position will determine its exact placement: if no explicit position is given, then the data field will be right-shifted by the width of the label (plus one space for the intervening blank). If a label position is given, however, then the data entry area's position is not affected by the label.

END DisplayFormImage

EVENT NAME: EditText

ENGLISH DESCRIPTION:
"EditText" allows the user to edit a multi-line data field.

PARAMETERS:

IN/OUT PARAMETERS:

data field : TYPE IS data field object
English Description:
"data field" is the data field to be edited.

FILES:

SEMANTICS:
The textual data in the data field is made available to a text editor. The user is then placed in this editor. When the editor finishes, FILLIN reads the new data field value. This value is then used as the value of the new data field. The new data is to be redisplayed immediately, replacing what was there before.

END EditText

EVENT NAME: fillin

ENGLISH DESCRIPTION:
This is the requirements for the fillin system.

PARAMETERS:

IN PARAMETERS:

args : TYPE IS a list, where each element is a string.
English Description:
"args" is a list of arguments for the fillin program. The list must contain two elements, the first of which is the form template file name, and the second of which is the data file name.

FILES:
The following files are required:

- Standard input (which must be a user's keyboard) to read commands.
- Standard output (which must be a user's screen) to display output.
- A readable form template file.
- A readable form data file.
- A writable form data file.

SEMANTICS:
The fillin system is a form-oriented user interface tool that collects data from the user. This data, which is most conveniently regarded as a linear list, is stored in a form data file.

END fillin

EVENT NAME: NextField

ENGLISH DESCRIPTION:
The "NextField" event moves the cursor to the next field in the form image.

PARAMETERS:

IN/OUT PARAMETERS:

form : TYPE IS form object
English Description:
"form" is the form whose image appears on the screen.

FILES:
standard_output is accessed, to move the cursor.

SEMANTICS:
The cursor is to be moved to the next data field. If the cursor is currently positioned on the last data field in the form, then it is to be moved to the first data field in the form. The form parameter is to be updated so that the current field becomes the field just moved to.

END NextField

EVENT NAME: OpenFile*

ENGLISH DESCRIPTION:
The "OpenFile" event opens a file, so that the file may be read or written.

PARAMETERS:

IN PARAMETERS:

file name : TYPE IS string
English Description:
"file name" is the name of the file to open.

OUT PARAMETERS:

file stream : TYPE IS file
English Description:
"file stream" is a data file stream.

FILES:

SEMANTICS:
The named file is opened. The "file stream" parameter becomes the standard reference means for the file.

END OpenFile

EVENT NAME: PrevField

ENGLISH DESCRIPTION:
The "PrevField" event moves the cursor to the previous field in the form image.

PARAMETERS:

IN/OUT PARAMETERS:

form : TYPE IS form object
English Description:
"form" is the form whose image appears on the screen.

FILES:
standard_output is accessed, to move the cursor.

SEMANTICS:
The cursor is to be moved to the previous data field. If the cursor is currently positioned on the first data field in the form, then it is to be moved to the last data field in the form. The form parameter is to be updated so that the current field becomes the field just moved to.

END PrevField

EVENT NAME: ReadFDF

ENGLISH DESCRIPTION:
"ReadFDF" reads a form data file from a specified file and merges it with the given form.

PARAMETERS:

IN PARAMETERS:

dfname : TYPE IS "dfname" is a string.
English Description:
"dfname" is the name of the form data file.

IN/OUT PARAMETERS:

form : TYPE IS form object
English Description:
"form" is a form object. Its data fields will, on completion of the event, contain the values of the form data file.

FILES:
The form data file is accessed.

SEMANTICS:
Each datum in the form data file is to be placed in the appropriate field of the form.

END ReadFDF

EVENT NAME: ReadFTF

ENGLISH DESCRIPTION:
The ReadFTF event reads and interprets a form template file, creating from it a form abstraction for the conversation.

PARAMETERS:

IN PARAMETERS:

tfname : TYPE IS "tfname" is a string.
English Description:
"tfname" is the name of the template file.

OUT PARAMETERS:

form : TYPE IS "form" is of type Form Object.
English Description:
"form" is the form abstraction created from reading the template file.

FILES:
The file containing the form template must be accessed.

SEMANTICS:
Read the file, according to the standard syntax (see appendix), and interpret from this a form, according to the definition of a form.

END ReadFTF

EVENT NAME: ReadKey

ENGLISH DESCRIPTION:
The ReadKey event reads a single keystroke from the user. This keystroke is not displayed.

PARAMETERS:

IN PARAMETERS:

x,y : TYPE IS integer
English Description:
(x,y) are an integer coordinate pair representing the position of the cursor when reading the character.

OUT PARAMETERS:

char : TYPE IS character
English Description:
"char" is the character read.

FILES:
standard_input is used to read the character.

SEMANTICS:
Move the cursor to the (x,y) position if it isn't there, and then (without echoing it) read a single character from the user. This character may be any ASCII character. Assign it to the "char" parameter.

END ReadKey

EVENT NAME: WriteFDF

ENGLISH DESCRIPTION:
"WriteFDF" creates a form data file from a given form.

PARAMETERS:

IN PARAMETERS:

form : TYPE IS form object
English Description:
"form" is a form abstraction.

file name : TYPE IS string
English Description:
"file name" is the name of the file in which the form data is to be placed.

FILES:
The file named by "file name" is created.

SEMANTICS:
Each data field in the form, in the order in which they appear, is written to the form data file.

END WriteFDF

3. MIDDLE-LEVEL REQUIREMENTS

The requirements in this section refine the customer's requirements. Much more formality is introduced, with greatly improved possibilities for data flow analysis and consistency checks. Enhancements include the following:

1. *Functional and non-functional terms are identified*. The beginnings of data types and functions are apparent.
2. *Packages are introduced*. Readers can see the relationship between functional and non-function terms, and the relation of terms, events, and conversations.
3. *State transformations are function-oriented*. This is the desirable intermediate form between the customer's informal and the designer's formal requirements. The correspondence between both is clear.
4. *Events are refined*. Some of the events from the customer's conversation are re-specified as conversations to demonstrate functionality more precisely. The substitution rule is still in force [9, Section 8.6], so, strictly speaking, user interface is not tied to a screen-oriented style yet. However, data flow is quite precise.

This section does not attempt to rewrite the previous one, which would require including all the text from the customer requirements in the context of the more formal model used herein. However, an important objective of DARWIN is to understand different requirements for the same object by including customer text alongside designer specifications. Therefore, portions of the customer requirements are included as comments; some parts, however, require re-reading the previous section.

3.1 TERM CLASSIFICATIONS

This section classifies each of the terms in the previous section as functional or non-functional. Doing so aids in defining packages.

Non-Functional Terms

Blank Character, Current Data Field, Cursor, Data Entry Area, Data Field, Form Data File, Form Image, Form Object, Form Template File, Form, Label Position, Label Width, Label, Multi-Line Data Field, Single-Line Data Field, Standard Input, Standard Output, Trim, Word.

Functional Terms

Echo, Editing, Filling, Merging, Scroll.

3.2 PACKAGES

This section introduces packages. The terms and events from the previous section are grouped together according to purpose. This provides yet another means of quickly locating an entity in DARWIN requirements.

3.2.1 data_field_package

```
PACKAGE data_field_package IS
        TYPE DataField IS PRIVATE ;
        TYPE DataFieldType IS PRIVATE ;
        TYPE DataFieldTypes IS (Int, Str, Mul) ;

        -- "data_field_value" retrieves the dfn'th field of the "form" parameter.
        FUNCTION data_field_value (
                form : FormObject ;
                dfn : integer
        ) RETURN DataField ;

        -- "data_field_value" retrieves the data field of the
        -- the "form" parameter named "FieldName".
        FUNCTION data_field_value (
                form : FormObject ;
                FieldName : string
        ) RETURN DataField ;

        -- "ValueOf" retrieves the value of a data field, as a string.
        FUNCTION ValueOf ( df: DataField) RETURN string ;

        -- "IsSingleLine" is true iff its argument data field is single-line.
        FUNCTION IsSingleLine ( df : DataField) RETURN boolean ;

        -- "IsMultiLine" is true iff its argument data field is multi-line.
        FUNCTION IsMultiLine ( df : DataField) RETURN boolean ;

EXCEPTIONS
        invalid_field_index ,
                -- This occurs when an attempt is made to access a field's
                -- value through:
                --      An integer index that is greater than the number of
                --      fields, or less than zero, or:
                --      A field name that is not in the form.

        invalid_field_type_access ;
                -- This occurs if an attempt is made to access a string-valued
                -- data field as an integer, or vice-versa.

END data_field_package ;
```

3.2.2 form_package

```
PACKAGE form_package IS
        WITH curses; USE curses ;
        WITH data_field_package ; USE data_field_package ;

        TYPE FormObject IS PRIVATE ;

        -- "ReadFDF" reads a form data file from a specified file
        -- and merges it with the given form.
        EVENT ReadFTF (
                IN dfname: string ;
                        -- "dfname" is the name of the form data file.
                IN OUT form: FormObject ;
                        -- "form" is a form object.  Its data fields
                        -- will, on completion of the event, contain the
                        -- values of the form data file.
        ) ;

        -- The ReadFTF event reads and interprets a form template file,
        -- creating from it a form abstraction for the conversation.
        EVENT ReadFDF (
                IN tfname: string ;
                        -- "tfname" is the name of the template file.
                OUT form: FormObject ;
                        -- "form" is the form abstraction created from
                        -- reading the template file.
        ) ;

        -- "WriteFDF" creates a form data file from a given form.
        EVENT WriteFDF (
                IN form: FormObject ;
                        -- "form" is a form abstraction.
                IN FileName: string ;
                        -- "file name" is the name of the file in which the form
                        -- data is to be placed.
        ) ;

        -- "AddText" is used to perform the basic operation of filling text
        -- in one of the data field areas of a form image.
        EVENT AddText (
                IN FirstCharacter: character ;
                        -- "FirstCharacter" is a single character that the
                        -- user typed that is to be the first character
                        -- of the text.
                OUT KeyRead : character ;
                        -- "KeyRead" is the last keystroke read, i.e., the
                        -- key that caused text entry to terminate.
                IN OUT form : FormObject ;
                        -- "form" represents the form image.
        ) ;
END form_package ;
```

3.2.3 form_image

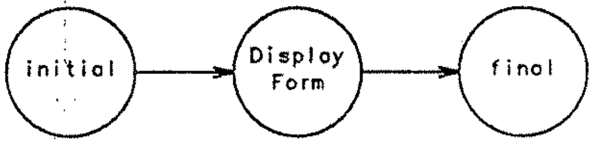
```
-- The "form_image" package includes all the operations and objects
-- associated with a form image.
PACKAGE form_image IS
        WITH form_package, data_field_package ;
        USE form_package, data_field_package ;

        -- "DisplayFormImage" displays a form image on the user's screen,
        -- complete with whatever data is currently in the fields.
        EVENT DisplayFormImage (
                IN form: FormObject ;
                        -- "form" is the form to be displayed.
        ) ;

        -- The "NextField" event moves the cursor to the next field
        -- in the form image.
        EVENT NextField (
                IN OUT form: FormObject ;
                        -- "form" is the form whose image appears on the screen.
        ) ;

        -- The "PrevField" event moves the cursor to the previous field
        -- in the form image.
        EVENT PrevField (
                IN OUT form: FormObject ;
                        -- "form" is the form whose image appears on the screen.
        ) ;

        -- "NumberOfFields" is a count of the number of fields in the
        -- "form" parameter.
        FUNCTION NumberOfFields (
                form : FormObject
        ) RETURN integer ;

        -- "CurrentDataField" is the current data field of the form.
        FUNCTION CurrentDataField (
                form : FormObject
        ) RETURN DataField ;

        -- "CurrentDataFieldNumber" is the index of the current data
        -- field within the form.
        FUNCTION CurrentDataFieldNumber (
                form : FormObject
        ) RETURN integer ;

END form_image ;
```

3.3 CONVERSATIONS

Several new conversations are presented in this section. They are refined from events of the previous section.

3.3.1 NextField

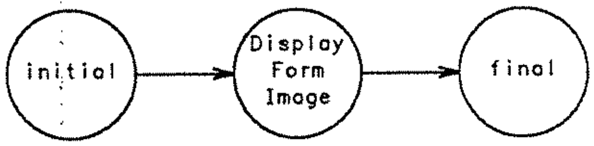
This section presents the conversation for the event NextField.



```
-- The "NextField" event moves the cursor to the next field in the form image.
CONVERSATION NextField (

        IN OUT form : FormObject ;
        -- "form" is the form whose image appears on the screen.
) IS

TRANSITIONS

        initial ( form ) CAUSES
                -- The form is changed: its current data field is reset. If the current data
                -- field is the last in the form, it becomes the first data field in the form.
                -- Otherwise, it becomes the next data field in the form.
                AND THEN DisplayFormImage ( form ) ;

        DisplayFormImage ( form ) CAUSES final ( form ) ;

END NextField ;
```

3.3.2 PrevField

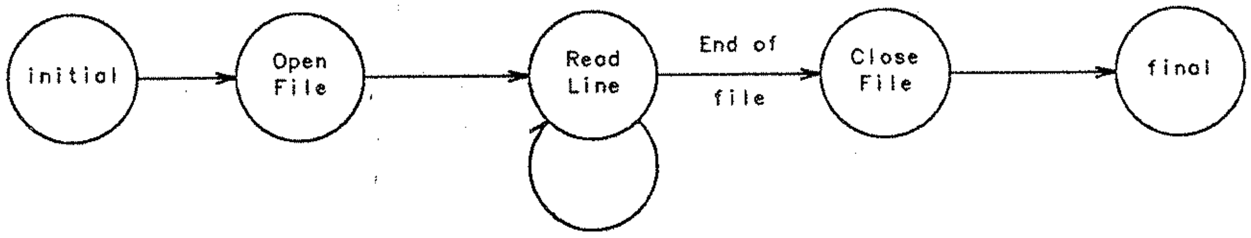
This section presents the conversation for the event PrevField.



```
-- The "PrevField" event moves the cursor to the previous field in the form image.
CONVERSATION PrevField (

        IN OUT form : FormObject ;
        -- "form" is the form whose image appears on the screen.
) IS

TRANSITIONS

        initial ( form ) CAUSES
                -- The form is changed: its current data field is reset.  If the current data
                -- field is the first in the form, it becomes the last data field in the form.
                -- Otherwise, it becomes the previous data field in the form.
                AND THEN DisplayFormImage ( form ) ;

        DisplayFormImage ( form ) CAUSES final ( form ) ;

END PrevField ;
```

3.3.3 ReadFTF

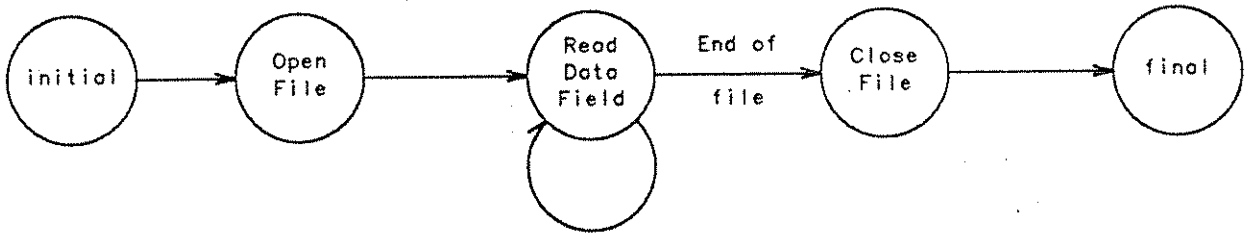
This is the conversation for reading a form template file. Although it is not shown here, the events used are are not useful elsewhere and hence would be local to the conversation.



```
-- The ReadFTF event reads and interprets a form template file, creating from it a form
-- abstraction for the conversation.
CONVERSATION ReadFTF (

        IN tfname : string ;
        -- "tfname" is the name of the template file.

        OUT form : FormObject ;
        -- "form" is the form abstraction create from reading the template file.
) IS

ENVIRONMENT
        form : FormObject ; "form" is the currently-defined form.
        ftf : file of character ; "ftf" is the file stream for the form.

TRANSITIONS

        initial ( tfname ) CAUSES
                -- Make the "form" object into an empty form, i.e., one with no data fields
                -- or trim.
                AND THEN OpenFile ( tfname ) ;

        OpenFile ( f ) CAUSES
                -- "ftf" becomes the open file stream; all reading will be done through this
                -- file.
                AND THEN ReadLine ( ftf ) ;

        ReadLine ( line ) CAUSES
                CASE end_of_file_found IN
                  WHEN false =>
                        -- Add data fields and trim to the form, as appropriate.
                        AND THEN ReadLine ( ftf ) ;
                  WHEN true => CloseFile ( ftf ) ;
                END CASE ;

        CloseFile CAUSES final (form) ;
END ReadFTF ;
```

3.3.4 ReadFDF

This is the conversation for merging the contents of a form data file into a form.



```
-- "ReadFDF" reads a form data file from a specified file and merges it with the given
-- form.

CONVERSATION ReadFDF (

        IN dfname : string ;
        -- "dfname" is the name of the form data file.

        form : FormObject ;
        -- "form" is a form object.  Its data fields will, on completion of the event,
        -- contain the values of the form data file.
) IS

ENVIRONMENT
        form : FormObject ; -- "form" is the currently-defined form.

        fdf : file of character ; -- "fdf" is the file stream for the form data file.

TRANSITIONS

        initial ( dfname ) CAUSES OpenFile ( dfname ) ;

        OpenFile ( f ) CAUSES
                -- "fdf" becomes the open file stream; all reading will be done through this
                -- file.
                AND THEN ReadDataField(fdf) ;

        ReadDataField ( field ) CAUSES
                CASE end_of_file_found IN
                  WHEN false =>
                        -- Place the data in the correct field of the form.
                        AND THEN ReadDataField ( fdf ) ;
                  WHEN true => CloseFile ( fdf ) ;
                END CASE ;

        CloseFile CAUSES final (form) ;

END ReadFDF ;
```

4. DESIGNER SPECIFICATION: LEVEL I

The specifications in this section are the high-level designer specifications. The conversations are now completely formal: they are fully parseable, and all function headers are defined. Therefore, data flow analysis is precise. However, not all the functions are defined: their definitions are often pseudo-code. These definitions will be completed in the next section.

4.1 CONVERSATIONS

4.1.1 Fillin

```
CONVERSATION fillin IS

ENVIRONMENT
        form : FormObject ;          -- This object holds the form abstraction being
                                     -- manipulated.

        form_TFN : string ;          -- This object holds the name of the file containing the
                                     -- form template.

        form_DFN : string ;          -- This object holds the name of the file from where
                                     -- form data is to be read, and to where form data is
                                     -- to be written.

        bs : character ;             -- This object holds the value of the user's "erase-
                                     -- character" key.

        erase : character ;          -- This object holds the value of the user's "erase-line"
                                     -- key.

        eraseW : character ;         -- This object holds the value of the user's "erase-
                                     -- word" key.

TRANSITIONS

        initial (argumentlist) CAUSES
                -- Determine, from the argument list, the names of the form template file
                -- and the form data file. Place them in the environment.
                form_TFN := head(argumentlist),              -- First argument.
                form_DFN := (head * tail) ( argumentlist) ;  -- Second argument.
                AND THEN ReadFTF(form_TFN) ;

        ReadFTF ( form ) CAUSES ReadFDF ( form_DFN, form ) ;

        ReadFDF ( f ) CAUSES
                -- Place the form in the environment.
                form := f
                AND THEN DisplayFormImage(form) ;

        DisplayFormImage CAUSES ReadKey ;

        ReadKey(KeyRead) CAUSES
                CASE KeyRead IN
                  WHEN EOT            -- control-d
                       => WriteFDF(form) ;
                  WHEN SO             -- control-n
                       => NextField(form) ;
                  WHEN DLE            -- control-p
```

```
                        => PrevField(form) ;
                    WHEN ENQ            -- control-e
                        => EditField(form) ;
                    WHEN ' ' .. '~'     -- a printable character
                        => AddText(form, KeyRead) ;
                END CASE ;

        WriteFDF CAUSES final ;

        AddText(form, KeyRead) CAUSES
                CASE KeyRead IN
                    WHEN EOT            -- control-d
                        => WriteFDF(form) ;
                    WHEN SO             -- control-n
                        => NextField(form) ;
                    WHEN DLE            -- control-p
                        => PrevField(form) ;
                    WHEN ENQ            -- control-e
                        => EditField(form) ;
                END CASE ;

        EditText(form) CAUSES ReadKey ;

        NextField(form) CAUSES ReadKey ;

        PrevField(form) CAUSES ReadKey ;

NODE ReadKey IS
        DIVERSION
                -- This diversion component handles mis-typed editing keys.
                OCCURS WHEN
                        -- KeyRead is one of the user's backspace, line erase, or word erase
                        -- characters.
                        KeyRead = erase OR KeyRead = bs OR KeyRead = wordE ;
                CAUSES EVENT
                        Message("You have not yet entered any text.")
                RECOVER BY LOOPING

        DIVERSION
                -- This diversion component handles other non-printing characters.
                OCCURS WHEN
                        -- KeyRead is any non-printing character that is not a special key
                        -- and is also not one of the user's editing keys.
                        NOT KeyRead IN [' ' .. '~']
                                AND
                                KeyRead /= erase
                                AND KeyRead /= bs
                                AND KeyRead /= wordE ;
                CAUSES EVENT
                        Message(bell)
                RECOVER BY LOOPING
        DIVERSION
                -- This diversion component is for bad editing attempts.
                OCCURS WHEN
                        -- KeyRead is control-e and the current field is a single-line field.
                        KeyRead = ENQ AND IsSingleLine(CurrentField(form))
                CAUSES EVENT
                        Message("You may only edit multi-line fields.")
                RECOVER BY LOOPING

END ReadKey

END fillin ;
```

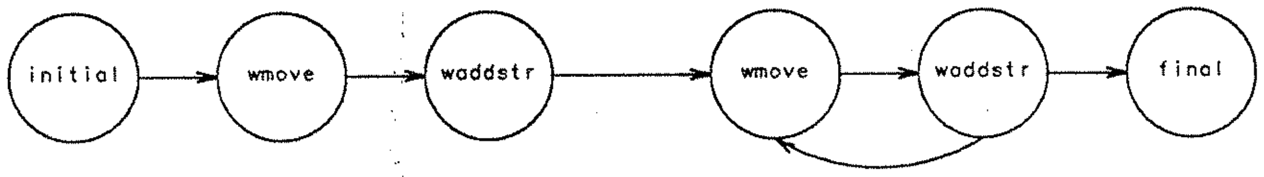
4.1.2 AddText

"AddText" is used to perform the basic operation of filling text in one of the data field areas of a form image.

```
CONVERSATION AddText (

        FirstCharacter : character ;
        -- "FirstCharacter" is a single character that the user typed that is to be the first
        -- character of the text.

        KeyRead : character ;
        -- "KeyRead" is the last keystroke read, i.e., the key that caused text entry to
        -- terminate.

        form : FormObject ;
        -- "form" represents the form image.
) IS

ENVIRONMENT
        form : FormObject ;
        -- This object holds the form to which text is being added.

TRANSITIONS

        initial(form, FirstCharacter) CAUSES
                -- Reset the data value of the form's current field to "FirstCharacter".
                form := ResetDataField(CurrentField(form), form, FirstCharacter) ;
                AND THEN DisplayFormImage(form) ;

        DisplayFormImage CAUSES ReadKey ;

        ReadKey (keystroke) CAUSES
                IF fillin_command_key(keystroke)        -- control-d   OR   control-e   OR
                                                        -- control-N OR control-P
                THEN
                        final (form, keystroke)
        ELSIF bs      -- User's erase-character key
        THEN
                -- Remove the last character from the current field's data value.
                form := ResetDataField(
                        CurrentField(form),
                        form,
                        substr(
                                ValueOf(CurrentField(form)),
                                (length * ValueOf * CurrentField)(form) - 1
                                )
                        )
                AND THEN DisplayFormImage (form)
        ELSIF erase   -- user's line-kill key
        THEN
                -- The curent field's data value becomes an empty string.
                form := ResetDataField( CurrentField(form), form, "")
                AND THEN DisplayFormImage (form)
        ELSIF wordE  -- user's erase-word key
        THEN
                -- Remove the last "word" from the current field's data value.
                form := ResetDataField(
                        CurrentField(form),
                        form,
                        EraseWord(ValueOf(CurrentField(form)))
                        )
                AND THEN DisplayFormImage (form)
        ELSE
                -- The current field's data value becomes the old value, with the
                -- keystroke appended.
                form := ResetDataField(
                        CurrentField(form),
                        form,
                        concat(ValueOf(CurrentField(form)), KeyRead)
                        )
                AND THEN DisplayFormImage (form)
        END IF ;

NODE ReadKey IS
    DIVERSION
        -- This diversion component handles mis-typed editing keys.
        OCCURS WHEN
                -- KeyRead is one of the user's backspace, line erase, or word erase
                -- characters, and the cursor is positioned at the beginning of the
                -- current data field (which is equivalent to saying the field is
                -- empty).
                fillin_command_key(KeyRead) AND
                        (length * ValueOf * CurrentField)(form) = 0
        CAUSES EVENT
                -- Ring the terminal's bell.
                Message(bell)
        RECOVER BY LOOPING
        DIVERSION
                -- This diversion component handles non-printing characters.
                OCCURS WHEN
                        -- KeyRead is any non-printable character aside from those given
                        -- above.
                        NOT KeyRead IN [' ' .. '~']
                CAUSES EVENT
                        -- Ring the terminal's bell.
                        Message(bell)
                RECOVER BY LOOPING

        DIVERSION
                -- This diversion component handles data that is longer than the width of
                -- the data field.
                OCCURS WHEN
                        -- The current data field is a single-line data field, the existing data
                        -- in the current data field is the width of the field, and KeyRead
                        -- is a character that would cause text to be added.
                        (IsSingleLine * CurrentDataField)(form) AND
                                (WidthOf * CurrentDataField)(form) =
` * CurrentDataField)(form) AND                                                        (length * Valı
                                KeyRead IN [' ' .. '~']
                CAUSES EVENT
                        -- Ring the terminal's bell.
                        Message(bell)
                RECOVER BY LOOPING

        DIVERSION
                -- This diversion component handles non-digit characters in integer fields.
                OCCURS WHEN
                        -- The current data field is a single-line integer field, and the char-
                        -- acter typed is not a digit.
                        CurrentDataField(form).dtype = int AND          NOT KeyRead IN
                        ['0' .. '9']
                CAUSES EVENT
                        -- Integer field
                        Message("Integer Field")
                RECOVER BY LOOPING

    END ReadKey

    END AddText ;
```

4.1.3 DisplayFormImage

This section presents the conversation that, the conversation that, given a form image, places it on the screen. This would be difficult were it not for the existence of the CURSES package for screen control (see [9, Appendix B]). However, the problem involves little more than just moving to the correct spot on the screen and displaying the appropriate piece of text. The conversation involves two parts: displaying the screen image, and then filling in data field values. the



```
-- "DisplayFormImage" displays a form image on the user's screen, complete with whatever
-- data is currently in the fields.

CONVERSATION DisplayFormImage (

        form : FormObject ;          -- "form" is the form to be displayed.
) IS

ENVIRONMENT
        line : integer ;             -- "line" keeps track of the current line number.
        form : FormObject ;          -- "form" is the form being displayed.
        screen : string ;            -- "screen" is a string representing the screen image.
        field : integer ;            -- "field" tracks each field to be displayed.
        fields : DataField_l ;       -- "fields" is the list of fields remaining to be displayed.

TRANSITIONS

        initial ( f ) CAUSES
                form := f, line := 1
                AND THEN wmove[1](0, 0) ;

        wmove[1] CAUSES
                -- Build a screen image (one huge string) from the screen image in the
                -- form.
                screen := join(form.screen);
                AND THEN waddstr[1]( screen ) ;

        waddstr[1] CAUSES
```

```
                -- Begin to step through each field.
                fields := form.datafields
                AND THEN wmove[2] ((head(fields)).x, (head(fields).y)) ;

        wmove[2] CAUSES
                CASE (head(fields)).dtype) IN
                  WHEN Int OR Str => waddstr[2]((head(fields)).dvalue) ;
                  WHEN Mul =>
                        -- Create from the field's value a string, where each newline except
                        -- the last is followed by head(fields).dx spaces.
                        AND THEN waddstr[2]. ("the created string") ;
                END CASE ;

        waddstr[2] CAUSES
                IF empty(fields)    -- are all fields displayed?
                THEN
                        final ;
                ELSE
                        fields := tail(fields)
                        AND THEN wmove[2] ((head(fields)).x, (head(fields).y)) ;
                END IF ;

END DisplayFormImage ;
```

4.1.4 NextField

```
-- The "NextField" event moves the cursor to the next field in the form image.

CONVERSATION NextField (

        IN OUT form : FormObject ;
        -- "form" is the form whose image appears on the screen.
) IS

TRANSITIONS

        initial ( form ) CAUSES
                -- The form is changed: its current data field is reset.  If the current data
                -- field is the last in the form, it becomes the first data field in the form.
                -- Otherwise, it becomes the next data field in the form.
                form.cf := NextField(form)
                AND THEN DisplayFormImage ( form ) ;

        DisplayFormImage ( form ) CAUSES final ( form ) ;

END NextField ;
```

4.1.5 PrevField

```
-- The "PrevField" event moves the cursor to the previous field in the form image.
CONVERSATION PrevField (

        IN OUT form : FormObject ;
        -- "form" is the form whose image appears on the screen.
) IS

TRANSITIONS

        initial ( form ) CAUSES
                -- The form is changed: its current data field is reset.  If the current data
                -- field is the first in the form, it becomes the last data field in the form.
                -- Otherwise, it becomes the previous data field in the form.
                form.cf := PrevField(form)
                AND THEN DisplayFormImage ( form ) ;

        DisplayFormImage ( form ) CAUSES final ( form ) ;

END PrevField ;
```

4.1.6 ReadFDF
This is the conversation for merging the contents of a form data file into a form.

```
-- "ReadFDF" reads a form data file from a specified file and merges it with the given
-- form.

CONVERSATION ReadFDF (

        IN dfname : string ;
        -- "dfname" is the name of the form data file.

        form : FormObject ;
        -- "form" is a form object.  Its data fields will, on completion of the event,
        -- contain the values of the form data file.
) IS

ENVIRONMENT
        form : FormObject ; -- "form" is the currently-defined form.

        fdf : file of character ; -- "fdf" is the file stream for the form data file.

TRANSITIONS

        initial ( dfname ) CAUSES OpenFile ( dfname ) ;

        OpenFile ( f ) CAUSES
                -- "fdf" becomes the open file stream; all reading will be done through this
                -- file.
                fdf := f
                AND THEN ReadDataField(fdf) ;

        ReadDataField ( field ) CAUSES
                CASE end_of_file_found IN
                  WHEN false =>
                        -- Place the data in the correct field of the form.
                        form := ResetDataField(
                                        FieldNumber(field),
                                        form,
                                        ValueOf(field)
                                )
                        AND THEN ReadDataField ( fdf ) ;
                  WHEN true => CloseFile ( fdf ) ;
                END CASE ;

        CloseFile CAUSES final (form) ;

END ReadFDF ;
```

4.1.7 ReadFTF

```
-- The ReadFTF event reads and interprets a form template file, creating from it a form
-- abstraction for the conversation.
CONVERSATION ReadFTF (

        IN tfname : string ;
        -- "tfname" is the name of the template file.

        OUT form : FormObject ;
        -- "form" is the form abstraction create from reading the template file.
) IS

ENVIRONMENT
        form : FormObject ; "form" is the currently-defined form.
        ftf : file of character ; "ftf" is the file stream for the form.

TRANSITIONS

        initial ( tfname ) CAUSES
                -- Make the "form" object into an empty form, i.e., one with no data fields
                -- or trim.
                form := NewForm
                AND THEN OpenFile ( tfname ) ;

        OpenFile ( f ) CAUSES
                -- "ftf" becomes the open file stream; all reading will be done through this
                -- file.
                ftf := f
                AND THEN ReadLine ( ftf ) ;

        ReadLine ( line ) CAUSES
                CASE end_of_file_found IN
                  WHEN false =>
                        -- Add data fields and trim to the form, as appropriate.
                        form := AddLine (form, line)
                        AND THEN ReadLine ( ftf ) ;
                  WHEN true => CloseFile ( ftf ) ;
                END CASE ;

        CloseFile CAUSES final (form) ;

END ReadFTF ;
```

4.2 FUNCTIONS

This section presents the functions introduced on the transitions for the designer-level specifications. It is, however, worth noting that many are useful as terms. Therefore, the comments preceding the functions would be available to the customer as DARWIN terms. This point is important: it explains why the comment header does not reference a function's name directly, but rather talks in terms of the function's behavior.

Some of the functions are fully defined; others have only the header. This depends largely on need, and on complexity. Those functions whose functionality has been omitted here will be completed in the next section.

```
-- The "next field" of a form is either:
-- •      The data field immediately following the current data field in the form, if the
--      current data field is not the last data field in the form, or:
-- •      The first data field in the form, if the current data field is the last data field
--      in the form.
FUNCTION NextField (
     form: FormObject
         -- "form" is the form whose next field is to be determined.
) RETURN integer
IS
BEGIN
     IF (CurrentDataFieldNumber * CurrentField)(form) = NumberOfFields(form)
     THEN  1
     ELSE  1 + CurrentDataFieldNumber(form)
     END IF
END NextField ;
```

```
-- The "previous field" of a form is either:
-- •      The data field immediately preceding the current data field in the form, if the
--      current data field is not the first data field in the form, or:
-- •      The last data field in the form, if the current data field is the first data field
--      in the form.
FUNCTION PrevField (
     form: FormObject
         -- "form" is the form whose previous field is to be determined.
) RETURN integer
IS
BEGIN
     IF CurrentDataFieldNumber(form) = 1
     THEN  NumberOfFields(form)
     ELSE  CurrentDataFieldNumber(form) - 1
     END IF
END PrevField ;
```

```
-- A "new form" is not really a form, but rather an object that is to be used in preparing
-- a form.  It is not a form because it has no associated screen image containing trim and
-- data, nor does it have any information on the type of data that may be entered.  How-
-- ever, it does have the ability to be made into a form.
FUNCTION NewForm
RETURN FormObject ;


-- "Adding a line to a form" means to add a new specification of a data field, or of trim,
-- to a form.  That is, the form will include extra trim (extending the screen image) or
-- will contain more data fields.  Because of the format of a form template file, adding a
-- single "line" may involve adding several data fields and trim items, or (if the line is
-- empty) just extending the form image.
FUNCTION AddLine (
     form: FormObject ;
         -- "form" is the form to which a line is to be added.
     line : string
                     -- "line" is a form template file line, containing zero or more instances
                     -- of trim and data fields.
) RETURN FormObject ;


-- To "reset a data field" means to replace the value of a data field in a form object with
-- any legal value.
FUNCTION ResetDataField (
     field: DataField ;
         -- "field" is the data field whose value is to be replaced.
     form : FormObject ;
         -- "form" is the form that contains the data field.
     value: string
         -- "value" is the replacement value for the data field.
) RETURN FormObject ;
```

```
-- To "erase a word" is to take a string, and, starting at its end, remove all text that con-
-- stitutes the last "word" in the string; or, if the string is empty, to return simply the
-- word.
FUNCTION EraseWord (
    value : string
) RETURN string
IS
BEGIN
    CASE length ( string ) IN
      WHEN 0
        => string ;
      WHEN OTHERS
        => substr(
               word,
               length(word) - index(reverse(word), " ")
           ) ;
    END CASE ;
END EraseWord ;


-- To "join" a screen image is to take it and produce a large string that consists of all
-- lines in the screen joined together, with each separated by a newline.
FUNCTION join (
    screen: screen_type
) RETURN string
IS
BEGIN
    IF empty(l)
    THEN  nil ;
    ELSE
        (construct (
            (concat * construct ( head * head, tail)),
            (join * tail * head)
            )
            ([screen, "\n"]) ;
    END IF ;
END join ;
```

4.3 DATA TYPES

This section presents data type definitions for fillin. The reader will note that these types were first given as private in the packages of section 3.

The first type describes the information required for a data field. This aggregate is defined most easily as a record.

```
TYPE DataField IS
    RECORD
        dname: string ; -- "dname" is the name of the data field,
                        -- or empty if the data field has no name.
        dtype: DataFieldTypes ;
            -- "dtype" is the field's type (integer, string, multi-line).
        dvalue: string ;
            -- "dvalue" stores the value.
        dwidth: integer ;
            -- "dwidth" is the width of the data field.
        dheight: integer ;
            -- "dheight" is the height of the data field.
        dx, dy: integer;
            -- "dx" and "dy" are the data field's location on
            -- the screen.
        fieldno: integer ;
            -- "fieldno" is the ordinal number of the data field
            -- within the form object.
    END RECORD ;
```

The next data type defines a form object. This is an ordered set of data fields, here defined most conveniently as a list, plus some information describing the status of the form image. Two new packages are needed: list instantiations of the screen (represented as a list of character arrays) and the data fields.

```
PACKAGE DataField_lp IS NEW listp ( item => df_rec );
TYPE DataField_l IS NEW DataField_lp'list ;

PACKAGE screen_p IS NEW listp ( item => ARRAY(<>) OF character ) ;
TYPE screen_type IS NEW screen_p'list ;

TYPE FormObject (width, height: integer) IS
    RECORD
        cf: integer ;-- "cf" indicates the current field.
        nf: integer ;-- "nf" is the total number of fields in the form.
        screen: screen_type ;
            -- "screen" is the screen image.
        datafields:  DataField_l ;
            -- "datafields" is the list of fields.
    END RECORD ;
```

5. DESIGNER SPECIFICATIONS: LEVEL 2

This section contains fully formal designer specifications. Only functions are included here; due to the way in which the specification was developed, conversations are as formal as necessary. Actually, certain conversations are not yet complete, but this was a deliberate choice, for reasons that are explained later.

5.1 FUNCTIONS

This section defines most functions used in the specification that have not been defined already.

The first group of functions are for the package **data_field_package** from section 3.2.1.

```
-- "data_field_value" retrieves the dfn'th field of the "form" parameter.
FUNCTION data_field_value (
     form : FormObject ;
     dfn : integer
) RETURN DataField
IS
BEGIN
     IF dfn = 1
     THEN  head (form.datafields) ;
     ELSE  data_field_value (
           [form.cf, form.nf, form.screen, tail(form.datafields)],
           dfn-1
        ) ;
     END IF ;
END data_field_value ;
```

```
-- "data_field_value" retrieves the data field of the the "form" parameter named "Field-
-- Name".
FUNCTION data_field_value (
     form : FormObject ;
     FieldName : string
) RETURN DataField
IS
BEGIN
     IF FieldName = head (form.datafields.dname)
     THEN  head (form.datafields) ;
     ELSE  data_field_value (
           [form.cf, form.nf, form.screen, tail(form.datafields)],
           FieldName
        ) ;
     END IF ;

END data_field_value ;
```

```
-- The "value of" a data field refers to its string value, since from the user's perspective
-- filling a data field means entering characters (even if the field is integer-typed).
FUNCTION ValueOf ( df: DataField ) RETURN string
IS
BEGIN
    df.dvalue ;
END ValueOf ;
```

```
-- A data field "is single line" if it is exactly one line high.
FUNCTION IsSingleLine ( df : DataField) RETURN boolean
IS
BEGIN
    df.dheight = 1;
END IsSingleLine ;
```

```
-- A data field "is multi line" if it is more than one line high.
FUNCTION IsMultiLine ( df : DataField) RETURN boolean
IS
BEGIN
    df.dheight > 1;
END IsMultiLine ;
```

Some functions from section 4.2 were left undefined. Their definitions are given here.

```
-- A "new form" is not really a form, but rather an object that is to be used in preparing
-- a form. It is not a form because it has no associated screen image containing trim and
-- data, nor does it have any information on the type of data that may be entered. How-
-- ever, it does have the ability to be made into a form.
FUNCTION NewForm
RETURN FormObject
IS
BEGIN
    [
        1,      -- Current field is initially the first field.
        0,      -- Number of fields is zero.
        [],     -- There is no screen image yet.
        []      -- There are no data fields either.
    ] ;
END NewForm ;
```

```
-- To "reset a data field" means to replace the value of a data field in a form object with
-- any legal value.
FUNCTION ResetDataField (
    field: DataField ;
        -- "field" is the data field whose value is to be replaced.
    form : FormObject ;
        -- "form" is the form that contains the data field.
    value: string
        -- "value" is the replacement value for the data field.
) RETURN FormObject
IS
BEGIN
        -- The form is constructed from the original form's values, with indicated field's
        -- value replaced.
    form := [
        form.cf, form,nf, form.screen,
        [
            FieldsBefore(form, field),
            [
                field.dname, field.dtype,
                value,      -- This replaces the value.
                field.dx, field.dy, fieldno
            ],
            FieldsAfter(form, field)
        ]
        ] ;
END ResetDataField ;

-- Two functions support the above.  They build lists consisting of, respectively, all ele-
-- ments up to and after the indicated field.
FUNCTION FieldsBefore(
    form: FormObject ;
    field : DataField
) IS
BEGIN
    PartOfList(form.datafields, 1, field.fieldno - 1) ;
END FieldsBefore ;

FUNCTION FieldsAfter(
    form: FormObject ;
    field : DataField
) IS
BEGIN
    PartOfList(form.datafields, field.fieldno + 1, form.nf) ;
END FieldsAfter ;

-- Finally, the function "PartOfList" takes its argument list and produces the slice of that
-- list specified by its "from" and "to" arguments.
FUNCTION PartOfList (
    l: DataField_l ;
    from, to: integer
) RETURN DataField_l
IS
BEGIN
```

```
          IF from > 1
          THEN  PartOfList(tail(l), from - 1, to - 1);
          ELSE IF from >= to
          THEN  [head(l), PartOfList(tail(l), from+1, to)];
          ELSE  [];
          END IF;
     END PartOfList ;
```

6. CONCLUSIONS

The specification of fillin was undertaken with the goal of producing something both readable by customers and useful to developers. Several valuable lessons have been learned in the process. Particular attention was paid to the following during the course of the specification:

1. The relationship between customer and designer specifications.
2. The feasibility of converting from English descriptions of functionality to DARWIN's formal notation.
3. How much prototyping a customer could accomplish, without extensive help from a designer.
4. The optimum amount of formality in the specifications.

Each of these is discussed below.

6.1 Customer-Designer Relationships

In most cases, it is both possible and advantageous to preserve relationships between entities in the customer and designer specifications. Concepts such as "form," and operations such as "filling," were quickly recognized as necessary customer terms and converted into formal types and functions. Conversely, certain operations and objects introduced by the designer were successfully converted into customer terms that described user-level concepts. For example, when entering text, the fillin user may "erase the last word." A precise definition of what this means was first entered by a designer, as a functional expression operating on a string and resulting in that string without the last word. Because it is also something that is useful for a customer to know, it was defined as a term. However, although the customer had originally defined the term "word," the designer did not make "word" a formal data type: form data was better viewed as a string than as a list of words. Instead, operations that dealt with words were packaged into a set of functions that map onto the string type.

6.2 Converting English to Functional Forms

Converting English to an operational specification is non-trivial, but the DARWIN approach did aid in the process. We had anticipated that the transformations on arcs would often be large, complex expressions, but instead they tended to resemble the one seen in Figure 0: a single function, or a small number of functions, operating on the relevant data and performing a transformation whose exact nature is defined elsewhere. This keeps the transformations easy to read, and, as mentioned earlier, helps eliminate a major source of errors in specifications by allowing data flow checking.

Of course, the formal specification of the transformation was not always simple. The usual approach was to first create a function header (such as that show earlier for along with an English comment describing the nature of the function. Writing the formal expression that denoted the function was postponed until it was actually needed. As explained below, this kept the specifications at a certain level of formality.

6.3 Prototyping

Creating prototypes requires some formality, a degree of formality not found in the customer's model. However, because requirements written by the designer may be expressed in the customer's model and used by the customer, customers may use formal events without referencing the formal definition; instead, they need only read the comments. Therefore, customers may piece together events from the CURSES package to form window-oriented dialogues, or they may use events that read and write lines of data to form a prompt-response dialogue. Furthermore, DARWIN supports a concept called the "diversion" [8]. Diversions model user input errors, help requests, and other important components of a user interface that are "non-functional;" that is, they do not produce functional outputs, only messages that inform the user on how to properly enter data. Representing such information is awkward in a graph-based model, because it significantly increases the number of nodes in a graph and consequently reduces the graph's readability; the diversion concept alleviates the problem, making experimentation with non-functional information far more practical.

The customer can build a realistic mock-up of a system in this fashion, using a fixed set of inputs that produce canned values. Producing something with more functionality requires more effort, and in DARWIN requires using the designer's model. However, in an interactive system specification, prototyping user interface is often all that matters. The DARWIN approach—allow full mock-up capabilities, but support user interface capabilities best—thus seems eminently practical.

6.4 Degree of Formality

Deciding how formal to make the specifications is always a difficult question. Complete formality is desirable but, due to the amount of detail required, may take too much time to be cost-effective. For the fillin specification, our goal was to evaluate the advantages of the various possible levels of formality.

There were four distinct levels to the specifications:

1. *The customer's specifications.* This level is clearly necessary for the customer's benefit, and is also the logical starting point.

2. *An intermediate-level set of specifications*, essentially customer specifications with more refined events added information on transitions, and with terms, events, and conversations organized as packages (in the Ada sense of the word). Few additional automated checks were possible at this level, but the additional conversations permitted more accurate prototyping than was possible at level 1. Moreover, the extra organization made the specifications significantly easier to read.

3. *Designer-level, using the formal syntax of the designer's language* but with most functionality expressed in pseudo-formal notation. Here, data flow could be checked, and completeness of type, function, and event definitions could be verified. Prototyping possibilities were improved, as in some places functionality could be expressed simply and concisely. However, the specifications were still too informal to facilitate full prototypes.

4. *Designer-level, with formally-specified functionality.* At this level, fully functional prototypes can be constructed, and sophisticated analysis techniques can be applied based on the mathematical properties of a functional language.

The fillin specifications were written down to level 3, and in certain places to level 4. There were clear advantages in using the analyzable syntax, and in being able to construct mock-ups. However, it was not deemed important to have a complete mock-up (which after all is the nature of prototyping); instead, we only wrote specifications at level 4 where we felt possible ambiguities existed. This appears likely to be the guiding principle for future DARWIN projects: complete the specifications for levels 1-3, and then write level 4 specifications for those parts where prototyping or formal analysis is useful or necessary.

6.5 Other Issues

Several other points emerged from the specification that are worth noting. First, one should concentrate on certain parts of the specification depending on the information one wishes to extract. The designer's section is not easy reading, but then, formal specifications never are; the information it contains is, however, precise and unambiguous, and should prove useful to designers. To understand fillin, customers and designers alike should turn first to the customer's specification. On reflection, this is not surprising, since English remains a more expressive language than any mathematical notations developed to date.

Second, there is a clear need for better (and automated) text formatting. The specification was produced using the *troff* package using the *-MM* macro package [6], with added macro support for DARWIN constructs. Moreover, conversations and events were formatted using a preprocessor, indicating the feasibility of automated processing. However, we did not spend too much time trying to determine the optimal text formatting style, nor was it considered worthwhile to produce an index. This was a mistake, and should be corrected in future specifications of any size. Another improvement would be to use a metanotation for word categories in text, such as that found in [3]; much of the specification is English text, and using a different type face for term names, event names, etc. would make them easier to locate.

Third, the reader has noted that the specification is not fully formal. Areas deliberately left informally specified are as follows:

1. The conversations for reading and writing form data files.
2. The conversation for reading form template files.
3. Certain parts dealing with the environment.

In each case, this results from an issue that is not yet resolved. The reason for 1) is that the format of the data files is not yet fixed; hence, a conversation to read one cannot be defined, and so the specification leaves it as an event. The others are left unspecified because there is as yet no environment interface from DARWIN specifications. The environment, and the functions one might wish to perform on it, is a hazy and not well understood area; the goal for DARWIN is to copy those functions found in an APSE [2], but since no APSE exists yet, the goal is unachievable.

Fourth, the designer's model requires automated consistency checking, for types, data flow, etc. Once again, this is not surprising for a specification of significant size; syntax errors, missing type definitions, etc., are hard to spot through manual inspections, just as in programming languages. However, a parser/analyzer should be written soon to make DARWIN more useful for the designer.

Finally, we note that there appears to be an extra classf of terms: internal terms, generally written by the designer. Internal terms are intermediate functions or objects that do not correspond to external objects or user-level functions. They are helpful in understanding the specification, but should not appear along with the regular term list. This suggests that a new section might be desirable.

REFERENCES

[1] K. Arnold, *Screen Updating and Cursor Movement Optimization: A Library Package*, Unix Programmer's Manual (4.2 Berkeley Software Distribution), Volume II-D, Berkeley, CA, 1983.

[2] T. Buxton, *Department of Defense Requirements for a Common Programming Environment (Stoneman)*, U.S. Department of Defense, 1980.

[3] K. Henninger, J. Kallander, D. Parnas and J. Shore, *Software Requirements for the A-E7 Aircraft*, Report 3876, Naval Research Labs, Washington, D.C., Nov. 1978.

[4] B. Kernighan and J. Mashey, "The UNIX Programming Environment," *Computer 14*, 4 (Apr. 1981), pp. 12-24.

[5] M. Penedo and S. Wartik, "Reusable Tools for Software Engineering Environments," *Proc. IFIP Working Group 8.1 Conf. on Environments to Support Information System Design Methodologies*, Bretton Woods, NH, Sep. 1985 (to appear).

[6] D. Smith, J. Mashey and E. Pariser, *PWB/MM Programmer's Workbench Memorandum Macros*, Bell Telephone Laboratories, Murray Hill, NJ, Jan. 1980.

[7] M. Stonebraker, E. Wong, P. Kreps and G. Held, "The Design and Implementation of INGRES," *ACM Trans. Database Systems 1*, 3 (Sep. 1975), pp. 189-222.

[8] S. Wartik and A. Pyster, "The Diversion Concept in Interactive System Specifications," *Proc. COMPSAC'83*, Chicago, IL, Nov. 1983, pp. 281-286.

[9] S. Wartik, *A Multi-Level Approach to the Production of Requirements for Interactive Computer Systems*, Ph.D. Thesis, University of California, Santa Barbara, CA, 1983.