Software Specifications for Interactive Systems: A Multi-Level Approach

By:
Steven Wartik*
Arthur Pyster†

Computer Science Report No. TR-85-06
May 29, 1985

This paper has been submitted for publication in *IEEE Transactions on Software Engineering*.

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

† Author's address: Digital Sound Corporation, 2030 Alameda Padre Serra, Santa Barbara, CA 93102
UUCP: ucsbcs!!dscvax2!ap

Abstract

A major difficulty in the specification of contractual software is the communication gap between the customers and the designers. Both groups are active participants, but their different backgrounds can cause misunderstandings that show up later in the specification document. This paper analyzes the role of both groups in the specification process. It uses the model of DARWIN, an environment for writing software requirements and specifications, and describes how customers and designers interact through the model. Special emphasis is placed on specifications of interactive computer systems. Experience with DARWIN is presented, and research directions are discussed.

Index Terms: Ada, contractual software, customer, designer, interactive system, rapid prototyping, software specification.

This work was supported by the TRW Defense Systems Group.

1. Introduction

Software system development is split into many phases, typically including requirements, specification, design, implementation, testing, and maintenance [15]. These are logical points at which to review achievements and establish new goals. Moreover, they assist the communication between the two groups involved in contractual software development: "customers" and "designers". These groups have quite different backgrounds and experience levels. Often, customers are casual users, unfamiliar with software development techniques; furthermore, being the end users of the system to be developed, they wish to view the system as a "black box" that somehow transforms inputs into the desired outputs. Designers are sophisticated computer users, seeing the system in terms of its architecture; also, their sophistication can make it difficult for them to comprehend the needs of the simpler customer environment. These differences in experience and understanding often result in systems that are incorrect or inadequate. The customer and designer must communicate much information to each other in various forms during the life of a system. Too often information is miscommunicated or lost, leading to systems that fail to meet customer needs.

Typically, customers write requirements, and designers do design and implementation; the specification, by contrast, is usually a joint effort. However, the groups have different needs from specifications. Customers want them to state how the system will be used (interface issues), whereas designers want to read a specification as a set of structural elements that they can convert into design. Furthermore, the specification stage ends active customer involvement for some time. Customers will assume supervisory roles during design and implementation, but will not participate in day-to-day details until the system is completed, often several years later. For this reason, customers must have confidence in a specification's correctness.

The relationship between customers and designers, and their roles prior to the design

phase, has received attention in such systems as USE [26] and DCDS [2]. It has been studied with regard to the relationship between the requirements and specification stages. However, their role for customers during specification is minimal. As discussed above, this is not always desirable.

This paper studies the relation between customers and designers during specifications, presented in terms of the DARWIN specification-writing environment [23]. Section 2 describes specification issues in more detail. Section 3 presents the models customers and designers use to write specifications. Section 4 describes the relation between the models. Section 5 discusses our experience with DARWIN and its models. Section 6 presents conclusions and directions.

2. An Overview of Interactive System Specifications

A specification is a nonprocedural description of inputs, outputs, and the relationship between them. Moreover, it typically includes some degree of structure that will eventually become the basis for system design. Many techniques have been used for writing software specifications. These range from plain English to formal systems based on axiomatic methods, such as AFFIRM [10] or SPECIAL [19]. English is understandable, but often ambiguous, and cannot be machine-checked. Hence, English specifications are usually incomplete and incorrect. A formal specification is exactly the opposite: while unambiguous and provably consistent, understanding it requires training in the concepts of the specification model.

Both types of models have their place. Customers usually are unwilling to invest the time to learn a formal specification model. Designers, with better mathematical training, generally can master a formal specification model fairly quickly; also, they appreciate the advantages of a formal specification as the basis for implementation. Furthermore, some formal specification systems can produce *rapid prototypes* [21] that provide customers with an executable version of their system; this seems the only effective way known to produce

correct specifications [14]. Rapid prototypes are particularly useful for interactive systems, where the man-machine interface is of great concern. A poor user interface destroys an implementation; it should be defined before implementation begins, while the customer is still active and can assert his opinions on acceptable styles. Unhappily, what seems acceptable on paper often turns out to be clumsy in practice, and so rapid prototyping during the specification is highly desirable. Not all of the user interface need be defined; certain decisions may be left for the design phase [7], but the overall style should be clear from the specification document.

The above discussion implies that informal specifications are needed for the customer, and formal specifications are desirable for both the customer and the designer. This idea has been used successfully in USE: it lets customers and designers jointly construct transition diagrams, and it also lets customers use the BASIS methodology [13] to define entities. DARWIN uses a similar notion, except that the specification model is more tightly integrated than that of USE, and therefore can produce a specification of any entity at varying levels of formality.

3. The DARWIN View of Specification Languages

Specification writing in DARWIN begins with the customer, who defines from the requirements a simple first version of a specification. This version is informal, with functionality described primarily in English. It categorizes system functions, as stated in the requirements, by a set of commands; therefore, it is the beginning of the user interface, and of the logical system architecture. Note that although customers will usually perform this task, designers are not forbidden to help. The division is based on expected experience levels, and on the necessity of having customer inputs into user interface style. Designers will lend their insights as needed; however, their active role comes later.

After the first version, the specification is transformed into increasingly formal versions in ways that are shown below. In each version, designer participation increases. The

process ends when a sufficient level of formality is reached. Fully formal specifications are useful but seldom cost-effective; hence, DARWIN allows many different levels of formality, even within a single specification.

3.1 The Customer View

The customer language's primary requirement is to be understandable. With his active role about to end, the customer must be certain that the specifications accurately reflect the requirements. Rapid prototyping has often been suggested as a means to accomplish this [11, 14, 16], but a rapid prototype alone is not enough. Rapid prototyping languages, while higher-level than programming languages, are still too formal for many customers. An English document is necessary, for two reasons. First, the specifications must incorporate all the functionality given in the requirements; proving this by executing a prototype is impractical. Because a system satisfies a particular set of test values does not imply that it is generally correct. This can only be determined by examining the prototype's structure. Second, prototyping languages do not as yet express concepts as clearly as English; reading them is difficult, or at least requires training time that customers generally do not wish to spend. An English description of the prototype will help customers understand the prototype in terms of the requirements, and so will help assure that the software specification satisfies the requirements.

DARWIN combines English descriptions with high-level formal specifications. Its model is based on directed graphs. Graph models are well suited to interactive system specifications. The concept was introduced (for design) in 1969 by Parnas [17], and has been used, in various forms, in many other systems since then (e.g., [1, 26]). Graphs are fairly easy to write, and to read; in addition, their structure can be adapted quite naturally to interactive systems. Each node in a DARWIN graph corresponds to a point when the system performs input or output; nodes are called *events*, because they correspond to the events that users perceive. Arcs connecting the nodes correspond to the times when the

system is "calculating," i.e., determining the next set of values to print, or what values to read.

In DARWIN, an interactive system has communication channels to one or more "worlds." A world is something external to the system that is *persistent*, meaning it exists before system execution begins, and continues after it ends. Humans are a persistent world; so are disk files, databases, etc. The idea is that a specification must describe its relation to all persistent worlds, since each changes (or at least is accessed) during execution. The specification must state the effect of executing the system on each world. All communication between a system and its worlds occurs through events. A system is presumed either to be doing internal calculations whose effect will not be apparent until communication takes place ("inside" the system, in the picture), or to be communicating with a world (at the boundary between the system and a world). An advantage of this representation is that specifications can refer to systems from each world's perspective. Thus, a system can be described from the point of view of a user; equally, it can be described from the perspective of the database, if that is desired.

Having access to all worlds in a single graph differs from other graph models of interactive systems, which are usually based on finite automata. Each node in the automaton performs both input and output, and to only one world, whereas in DARWIN nodes do either, and can talk to any of the worlds used by the system (although each node can talk to exactly one world). The DARWIN view is more flexible, and its model is self-contained; in the finite automata models, communication with other worlds is usually specified through additional model features. However, DARWIN graphs may contain more nodes, since they contact more worlds, and this can reduce readability. For most systems, the tradeoff seems worthwhile; systems that interface to a large number of worlds sometimes have confusing graphs.

Figure 1 shows a typical DARWIN graph. It is the specification of a simple forms

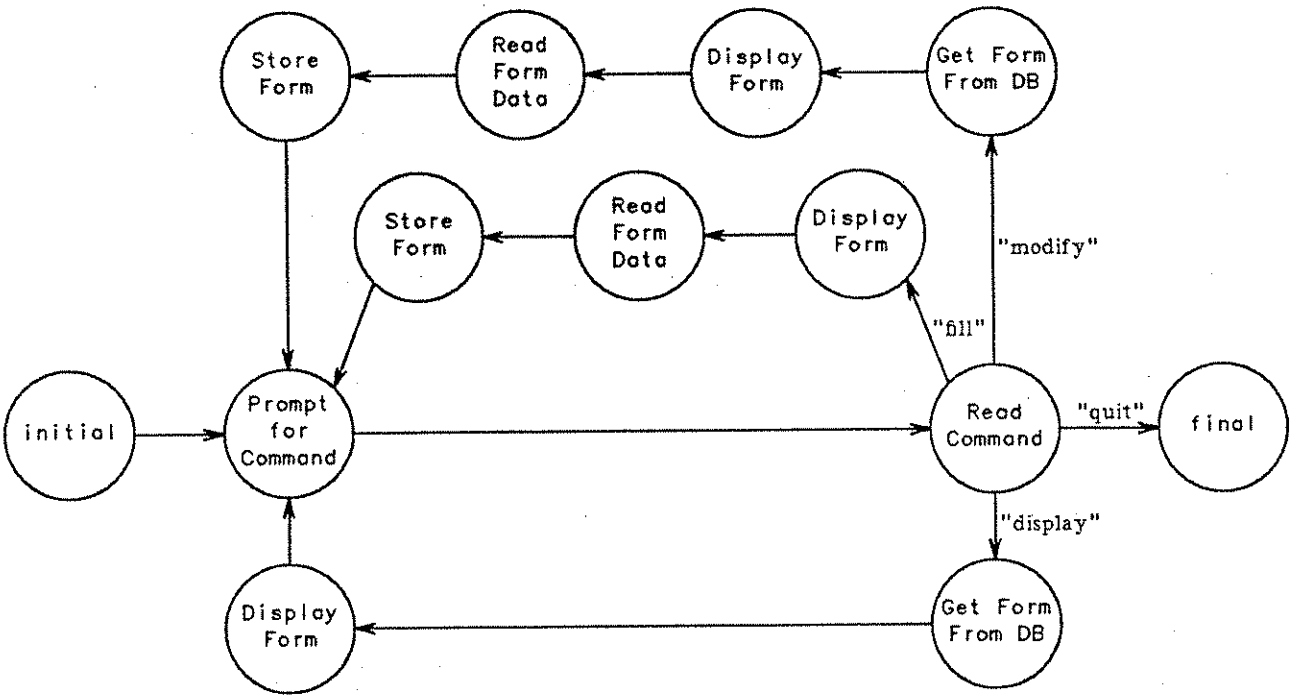
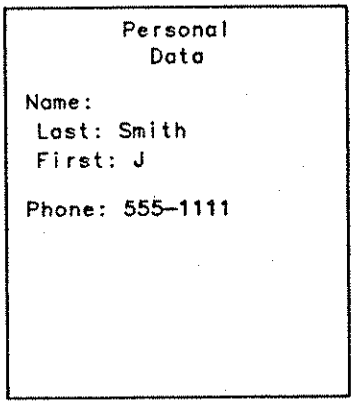


Figure 1. A Forms Management System

management system. There are four user commands: filling in (creating) a form, modifying an existing form, displaying an existing form, and quitting the system. Logically, a "form" is a set of fields. Some fields are textual, for the user's benefit; others are "data" fields with user-settable values. We represent a form as two sets of data: a "template" that describes the form layout, and the set of data entered in the data fields. The template is an ordered list of 4-tuples, each of the form:

```
[Name, type, x, y]
```

where "Name" is a string that identifies the field, "type" is either `T` or `D` for "textual" or "data", and x and y are an (x,y) coordinate pair giving the field's location on the form. This models the usual paper version of a form. To simplify the example, we consider only one type of form. It is shown in Figure 2, along with the set of data that defines it. The figure on the left is a simple personal information form consisting of three data areas: the "Last", "First", and "Phone" fields. The information on the right is the template and



Template:

```
[
    ["Personal", T, 6, 1],
    ["Data", T, 8, 2],
    ["Name:", T, 1, 4],
    ["Last:", D, 2, 5],
    ["First:", D, 2, 6],
    ["Phone:", D, 1, 8]
]
```

Data:

```
["Smith", "J", "555-1111"]
```

Figure 2. Example Form and Sample Data

data that define it. In subsequent examples, the template information will be represented using an "environment object" (see below) called `template`.

Execution starts at the "initial" event, and follows the arcs; it ends when the "final" event is reached. Command selection is indicated here by the label on the arcs emanating from the `ReadCommand` event. Thus, form filling is selected by typing "fill" when the system is reading a command, quitting by typing "quit", etc.

The Forms Management System talks to two worlds: the user, and a database that stores the forms. Since forms exist before and after the system executes, the database is a persistent world. Therefore, communication with the database is described by events, here `GetFormFromDB` and `StoreForm`, as is all communication with persistent worlds.

The graph does not show specifications of processing that is to occur on the arcs. For example, the difference between filling and modifying a form is that the former requires creating an empty form that may be displayed, while in the latter an existing form is retrieved from the database. Processing does not appear because it is too large to fit in the picture. Instead, it is specified as a separate description in a syntax that will be explained shortly.

Although the graph defines the flow of the conversation between system and worlds, more information is needed to understand this flow. In a customer specification, most of it

is in English. It takes the following forms:

1. *Event descriptions* that define what data passes in or out of the system through the event.
2. *Processing specifications* that define the processing that is to occur during transitions between nodes.
3. *Parameter-to-processing descriptions* that link data read by events to processing specifications.

Each is described below.

3.1.1 Event Descriptions

Each event has several attributes that together describe its function. These include:

1. *Parameters*, which specify the ways through which the event can communicate with the system (much as procedure parameters define the data that can be passed in and out of a procedure.) There are three parameter modes: IN, OUT, and IN OUT. The parameter modes are taken from Ada[1] [3], and have the same semantics as Ada procedure parameters.
2. A *semantics* description, which defines the relationship between parameters and data that is read or written.
3. *Files*, to define which worlds are accessed.

Figure 3 shows an example event. This event appears in the graph as the ReadCommand node, and performs the function of reading a user's command. The event has a single OUT parameter, used to pass the line read to the system. The SEMANTICS description specifies more precisely what is expected, and from where.

1. Ada is a registered trademark of the U.S. Government, Ada Joint Projects Office.

```
EVENT ReadCommand IS
     PARAMETERS:
          NAME:   Line_Read
          MODE:   OUT
          DESCRIPTION:
               "Line_Read" is a string that represents the
               line of data read.

     FILES:  standard_input
     SEMANTICS:
          A single line of text is read from the standard input file
          (the user's terminal).  This line must be one of "display",
          "fill", "modify", or "quit".  If it is, it is assigned
          to the "Line_Read" parameter.
END ReadCommand ;
```

Figure 3. The ReadCommand Event

```
EVENT DisplayForm IS
     PARAMETERS:
          NAME:   form
          MODE:   in
          DESCRIPTION:
               "form" is a form to be displayed.
     FILES:  standard_output
     SEMANTICS:
          Display each "field" of the form on the standard output
          file (user's terminal screen), left-justified at its
          associated (x,y) coordinate.
END DisplayForm
```

Figure 4. The DisplayForm Event

Figure 4 shows DisplayForm, a more complex event. This event, used to display a form on the user's terminal screen, appears three times in the graph. Its IN parameter is a form, sometimes empty (in the "fill" command), sometimes full (in the "display" and "modify" commands). Displaying a form is more complex than indicated here, but the SEMANTICS description is accurate enough for the first iteration. In a full specification, the event would be "refined" (see below) by a designer, giving it more formal semantics.

3.1.2 Processing Specifications

Between the times when a system communicates with its worlds, it is performing "processing". Processing is a conceptual state transformation. A user perceives a certain change of state between the time when data is read and the output next appears on his screen. Each node in a DARWIN graph corresponds to a possible user-perceived state. For example, at the ReadCommand node, the user sees the system as containing a set of forms,

ready to be operated on in one of three ways. Just after the ReadFormData node, the system is perceived as being in a slightly different state, for it now contains new information (the new form data); and on the transition following this node, the system is perceived as calculating a new database state, containing the new form data. Calculations of this nature are what are described on state transitions. Figure 5 shows how the arc between ReadCommand and DisplayForm (on the "fill" path) might be specified. This is not a procedural description of how to build a form; that comes later, during design. Rather, it corresponds to the user's view of how the data just entered is being used.

The above description specifies a transformation to a particular part of the entire system "state". This state is composed not only of a single form, but of all forms currently stored in the database. The user is aware of the existence of this set of forms as well. Thus, a DARWIN specification includes, for each graph, an *environment*. The environment consists of a set of objects, each of which is an object used by a world. Together, these objects comprise the system state. A graph combined with an environment defines a current state and the possible states reachable from the current state; hence, for an interactive system, it is a specification. A graph and its environment are called a *conversation* in DARWIN.

3.1.3 Parameter-to-Processing Descriptions

One more descriptive item is necessary. The processing description is informal, and its relation to event parameters is not always obvious. In the processing specification given above, no explicit mention is made of what is done with the form that is built. A parameter-to-processing description would therefore be included to say:

```
The form built is assigned to the "form" parameter of the DisplayForm event.
```

A parameter-to-parameter description is required to contain the parameter and event names. This is a simple but useful consistency check that helps assure its relevance.

```
Build an empty form.  The form will contain only the field attributes
(identifier and coordinates), but no data.  Therefore, the fourth element of
each tuple will be an empty string.
```

Figure 5. Processing Specification for Building an Empty Form

Customer model specification have two important advantages over plain English: they are better structured (due to the graphs, which help partition the system into its logical components), and they permit simple experiments with user interface issues. Often, however, they are not sufficiently formal. Formality is achieved using the designer's model, discussed below.

However, designers cannot understand what the customer has not explained; in the phrase "erase the last word," the concept of a word is clear enough in English but does not extend to the full ASCII character set. For this reason, the customer's model also includes an entity called a "term." A term consists of a name (a word or phrase) and an accompanying English description. Terms give the specification a glossary; furthermore, they are important in DARWIN's methodology. Noun terms correspond to the external objects that the system manipulates, and verb terms correspond to the user-defined system operations. Hence, they will tell the designer what objects and operations to specify.

DARWIN specifications may at first appear to be design specifications, or at least to imply a design. They do identify a system's logical divisions, and in fact specify much of what while become flow of control in the implementation. This is unavoidable when defining user interfaces, because much of a user interface consists of prompting for and reading data, in a certain order, and with no intermediate calculations. However, the specifications of functionality are non-procedural; what appears to be design is specification of user interface, and there are clear advantages to doing this prior to the design phase [6].

3.2 The Designer View

The designer's model, with respect to the customer's, serves to aid in determining the consistency and correctness of the customer's "readable" (English) specifications. The

designer is also producing something to guide design, but designer specifications in DARWIN have important feedback into customer work. Through completeness and consistency checks not possible with English, designer specifications are analyzed and corrected; and, through prototyping, customers are able to test a mock-up of their system. These corrections and prototypes are reflected back onto customer specifications to improve their correctness and readability.

The reader will observe some similarity between DARWIN's designer language and Ada. This is done because DARWIN is intended for specifying systems that are to be implemented in Ada. A full treatment of this subject is outside the scope of this paper, but it should be noted that many concepts are drawn from Ada, the meaning of part of a DARWIN specification can often be inferred from the corresponding Ada entity. Aside from syntactic similarities, DARWIN uses the Ada package concept, as well as Ada's data types. Therefore, someone who is not familiar with DARWIN can still understand much of a specification if he knows Ada. This is discussed further in [23].

A close relationship exists between customer and designer specifications. Software requirements often say little about the system-world interfaces, but software specifications must define what data is read, and when. For an interactive system, this means giving a precise specification of the user interface. The conversation that occurs between system and user should be well-defined. Its style (window vs. line-oriented, e.g.), the order in which data is collected and displayed, and the available help and error processing and recovery must be included as part of the specification.

Because of the close relationship, the DARWIN designer's model resembles the customer's model. The intent is that designer specifications be derivable from an entity in customer specifications, and that a designer entity be traceable back to its origin in the customer's specification. To this end, the designer's model is also graph-based, and uses events instantiated as nodes in the same way as the customer's model. However, the

designer's model uses a formal, unambiguous syntax. For example, Figure 6 shows the ReadCommand event from Figure 3 rewritten in the designer's language. Note that the event includes (as Ada-style comments) all the text from the customer's event, accompanying the now formal description of semantics. This enhances readability, adds useful redundancy, and furthermore is automatically performed by the tools that the designer uses to enter the specifications.

Rewriting a more complex event, such as DisplayForm, in the designer's language is often not possible using the simple I/O primitives shown above. Displaying a form depends on many factors, such as how many fields it contains or whether a field has data. This must be expressed without a sequence of statements, containing loops; such a specification would be too procedure-oriented. It is, however, convenient to express such an event as a DARWIN graph. This is done using a process called "refinement" of an event. The event is replaced, conceptually, by a graph, with subsequent effects analogous to a procedure call in a programming language: executing the event is equivalent to executing the graph refined from it. Refinement keeps graphs small, and also helps when writing specifications top-down or bottom-up.

Processing specifications are written using a functional notation, loosely based on Backus' language FP [5]. Functional languages are well suited to requirements and specifications, and have been applied in PAISLey [27]. Recall the perception of transitions between events as state transformations, transformations that are considered indivisible operations. A functional expression, associated with an arc in a DARWIN graph, models this exactly. For example, the specification from Figure 5 may be rewritten as:

```
-- Build an empty form.  The form will contain only the field attributes
-- (identifier and coordinates), but no data.  Therefore, the fourth element
-- of each tuple will be an empty string.
form := CreateForm(template, ["", "", ""])
```

Figure 7. Converting English to Formal Specifications

where CreateForm is a function whose first parameter is a form template, and whose second

```
EVENT ReadCommand (
     Line_Read: OUT string
          -- "Line_Read" is a string that represents the
          -- line of data read.
)
IS
     FILES   standard_input;
     SEMANTICS
          -- A single line of text is read from the standard input file
          -- (the user's terminal).  This line must be one of "display",
          -- "fill", "modify", or "quit".  If it is, it is assigned
          -- to the "Line_Read" parameter.
          get_line(Line_Read);
END ReadCommand ;
```

Figure 6. Designer's Version of ReadCommand

parameter is a list of values to fill the template's fields. This is a single, indivisible transformation of state, combining two objects into one.

The assignment statement may appear out of place in what is an applicative language. In fact, it is an assignment to the environment object form. That is, the designer also declares objects for each object in the system environment. The specification says that for a particular transition, the conceptual state change is to build an empty form, and that users perceive this as a change to the form area of the environment. The assignment symbol is a familiar notation for this.

Parameter-to-processing specifications are unnecessary in the designer's language; instead, a formal notation expresses the transitions. For example:

```
ReadCommand(line) CAUSES
     CASE line IN
       WHEN "fill" =>
         -- Build an empty form.  The form will contain only the field attributes
         -- (identifier and coordinates), but no data.  Therefore, the fourth element
         -- of each tuple will be an empty string.
         form := CreateForm(template, ["", "", ""])
         AND THEN DisplayForm(form);
       WHEN "display" => GetFormFromDB;
       WHEN "modify" =>  GetFormFromDB;
       WHEN "quit" =>    final;
     END CASE;
```

This describes the transitions emanating from ReadCommand. Only the transition between ReadCommand and DisplayForm has associated processing, shown by the assignment to the "form" environment variable. Parameters are linked to processing through the variables;

here, the `line` parameter is used in the `CASE`, and the newly created `form` is passed to `DisplayForm`. This information is identical to that specified by the customer, but is stated with formal semantics that can be interpreted and executed.

4. Relating the Models: A Rapid Prototyping Approach

The purpose of rapid prototyping is to build, quickly, a functional mock-up of a system. For an interactive system, experimenting with different man-machine interface styles is also important. The man-machine interface plays an important role in an interactive system. If it is delineated by the specification document, with the customer actively participating in its development, it will reflect the customer's rather than the designer's preferences in menus, use of windows, availability of help, etc.

Rapid prototyping in DARWIN is therefore a two-part process: defining user interface, and defining system functionality. This section discusses their role in specifications.

4.1 Defining the User Interface Through Rapid Prototyping

Each node in a DARWIN graph corresponds to a point when the system is to communicate with an outside world. Therefore, much of the information pertinent to the user interface—as opposed to system functionality—can be represented via nodes of a DARWIN graph. System functionality (specified on the arcs) is largely independent of user interface in this model. For this reson, defining the user interface is straightforward, and can be done by customers as well as designers (see [20,26], e.g.). Also, user interface can change without affecting functionality; indeed, little functionality need be present. When the objective is to test user interface styles, a set of built-in responses suffice to simulate functionality.

In this way, customers can build graphs that specify user interface and leave functional definitions in English, to be completed later by designers. Consider Figure 1 again. This graph defines user interface, in the sense that it fixes the order in which data is col-

lected from and displayed to the user. However, it says nothing about how data is collected or displayed. It does not state whether filling a form is window-oriented or line-oriented, for example. Furthermore, the style of form filling has no effect on the system functionality. Either style would suffice, provided the event delivers, through an OUT parameter, a "form" that can be stored in the database. This demonstrates the independence of functionality and user interface. While functionality and user interface are not always so independent, our experience has shown that events can often be switched to effect different interface styles, provided their parameter schema match. Thus a line-oriented forms management system can be changed to a window-oriented one by replacing line-oriented events with window-oriented ones. Packages of such events have been defined; for example, one modeled after the CURSES window management library [4] provides screen-management capabilities that simulate CRT-type terminals.

4.2 Defining Functionality Through Rapid Prototyping

Designers can use the functional language of DARWIN to produce rapid prototypes that specify functionality as well as user interface. Increased functionality also increases confidence in the correctness of the system, and the ability to automatically convert from specification to design. As discussed earlier, there is a point of diminishing cost-effectiveness. Therefore, another equally important goal of DARWIN is that it support specifications containing semi-formal entities. This does not mean allowing incomplete specifications, but rather that the exact wording of each help message is not required; or, for the forms management system, that the functionality `CreateForm` can be defined in English, even though its calling sequence is formally defined.

A language such as DARWIN's is helpful in this respect. Work from the SREM project [1] indicates that many errors occur from interface mistakes between a entities of a specification, and that a strongly-typed language can help alleviate this problem. The designer's language is strongly typed; it uses the rules from Ada, and furthermore uses an

Ada-like syntax for function declaration. The definition for CreateForm might be:

```
FUNCTION CreateForm(
     formtempl: template_type;
     fields: list of string
) RETURN form
```

with an accompanying English comment. This is simple to write, yet facilitates powerful consistency checks; the transition between ReadCommand and DisplayForm in the forms management system now has a well-defined interface between event parameters.

Because customer terms describe external objects and operations, the designer will usually want to convert them into formal objects. A term that describes an external object is analogous to a data type, and a term that describes an operation is analogous to a function. This natural categorization is an important aid to the designer in formalizing the specifications.

However, the designer may deviate from customer specifications, resulting in a prototype that performs as expected, but no longer matches the customer's (incorrect) specifications. In the DARWIN environment, this danger is avoided by the relationship between customer and designer entities. A designer works from descriptions written by a customer, and tools help the designer develop functional expressions from the customer's English descriptions, and also record the derivations. Therefore, when the designer changes an expression, he is requested to change the accompanying English description as well. This process has two desirable effects. First, it helps keep the formal and informal specifications consistent. Second, it helps identify which areas of the customer's specifications are unclear (those that designers cannot understand, or translate into incorrect prototypes). This produces the feedback effect mentioned earlier. Because the entities in the versions of the specifications are carefully related, the designers are helping customers document their work, and the customers are helping designers to better understand the specifications.

5. Experience with DARWIN

Although DARWIN is still relatively new, we have concentrated on writing specifications that test the model and the tools we have implemented. Examples have included a small relational database management system with an interface similar to INGRES [22], and CUSTOMER, one of the tools in the DARWIN environment. These, however, were unsatisfactory because they were after-the-fact specifications of existing systems, and because they concentrated on facets of the DARWIN model, instead of being a complete specification.

Accordingly, we undertook to write a complete set of specifications that would be used for an implementation. Our choice was a form-filling tool called FILLIN. This tool is described elsewhere [8,18]; briefly, it takes a "form template file" and presents the user with a form image that is an electronic equivalent of a paper form, which may be filled, edited, stored, or retrieved in the same manner as a paper form. FILLIN is not a stand-alone tool, but rather a user interface that allows other tools to collect information in a form-like manner. FILLIN had actually been implemented several years earlier (the forms management system in this paper is a simplified version) but we planned a new version that included significant user interface modifications and performance enhancements. It was an excellent choice for several reasons:

1. It is a highly interactive tool with a sophisticated user interface, and hence is well suited to DARWIN's model.
2. It is a non-trivial tool, but it is also not too large and so is easily controlled. The current implementation now contains about 5000 executable lines of C [12] code, most of which handles the user interface, plus another 5000 lines of support library code. We anticipate that the new version will be about the same size.
3. Its user community includes both customers and designers. Hence, it must be understood by both groups, which tests DARWIN's capacities in that area.

4. It exists and is well understood, yet enough changes are being introduced to make a realistic evaluation of the ability to conceive of system concepts and express them within DARWIN's model.

The specification of FILLIN was undertaken with the goal of producing a document that would be both readable by customers and useful to developers. The result [25] was a four-part specification document: one set of for the customer, two sets for the designer, and one intermediate-level set. This section summarizes the results. We were especially interested in evaluating the following:

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.

5.1 Customer-Designer Relationships

In most cases, it was 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.

5.2 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 7: 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 `CreateForm`), 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.

5.3 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" [24]. 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.

5.4 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. Conclusions and Directions

The objective of this paper has been to extend the usual notions of customer-designer relationships from the requirements to the specification stage. The role of the customer in specifications has traditionally been minimal, but will expand as improvements in rapid prototyping techniques make pre-implementation experiments practical. Misunderstanding of what the specifications really state, on the part of both customers and designers, has always been a major concern. It has produced innumerable systems that are "correct," in the sense that they satisfy the specifications, but do not do what the customer originally wanted.

Most research has concentrated on improving the quality of the requirements, and on making them "understandable." Good requirements are certainly desirable, but it is during the specification phase that many important aspects of a system are decided, aspects that greatly influence its usefulness. Some of these decisions should be made by the customer (e.g., user interface); others should be made by the designer (those with design impact); and the decisions from each are seldom independent. Customers and designers interact often during the specification phase and need a coherent methodology to produce specifications.

This paper has presented a multi-level specification model that can help in the information exchange process. A carefully-related set of entities at various levels of formality let customers and designers write the specifications together. Entities can be both formal and informal: described in English, yet with an underlying formal definition. There is no guarantee that a formal object, converted by a designer from a customer description, matches its informal counterpart, nor that the English description will be understandable, but such an object can be tested. If the customer's description was correctly translated by the designer (something the customer can test), then the description was clear enough for at least one person to understand. Similarly, a designer that cannot understand a customer description has uncovered a problem. While this may seem self-evident, such problems do not usually surface until the design phase; using the DARWIN model relationships helps in uncovering such problems.

Another problem with specifications is knowing when the "right" level of formality has been reached; the potential advantages of complete formality must be weighed against the time commitment required. Each project has its individual needs, but DARWIN helps by allowing exactly as much formality as is needed, and by permitting mixtures of formality within a specification. An entity deemed to need careful study may be formally defined and used along with informal ones whose definitions are more obvious.

Specifications are intended as the precursor for design, so the near-term goal for

DARWIN is to design the new version of FILLIN from its specification. There are two objectives: first, to evaluate the utility of the DARWIN model for developers (as distinct from the designers involved in the specification effort), and second, to determine how much of the transformation can be automated. Entities in a specification that model external-world objects are often realized as data types in the implementation; because DARWIN uses Ada syntax, some specification objects could be used directly in the implementation. This may not be desirable, due to efficiency considerations; nevertheless, FILLIN will present an opportunity to observe how types from a DARWIN specification are used in an implementation. FILLIN is to be written in C, not Ada, due to the limited Ada support available, but the data types used should be similar in either case.

The long-term goal of DARWIN is to merge it into an APSE [9]. The DARWIN model, and accompanying tools, would support the specification phase. DARWIN is intended to model interactive systems; because more comprehensive support is necessary, we are investigating allowing a more general specification model. Two directions in this area are being studied: extending DARWIN's model, or having it complement a model such as SREM's for embedded systems. In this way general support for specifications would be achieved.

7. Acknowledgements

We wish to thank Maria Penedo and Frank Belz for their insights into customers and designers. DARWIN was funded by TRW Inc., in particular through the Software Productivity Project, originally led by Barry Boehm and now Don Stuckle, and we thank them for their support.

REFERENCES

[1] M. Alford, *The Software Requirements Engineering Methodology: An Overview*, TRW Software Series TRW-SS-80-03, TRW Defense Systems Group, Redondo Beach, CA, May 1980.

[2] M. Alford, "Software Requirements in the 80s: From Alchemy to Science," *Proc. ACM'80*, Nashville, TN, Oct. 1980.

[3] *Ada Language Reference Manual*, ANSI/MIL-STD-1815A, American National Standards Institute, Inc., 1983.

[4] 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.

[5] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Comm. ACM 21*, 8 (Aug. 1979), pp. 613-641.

[6] B. Boehm and A. Pyster, *Rapid Prototyping: A Position Paper*, ACM SIGSOFT Rapid Prototyping Workshop, Columbia, MD, Apr. 1982.

[7] B. Boehm, T. Gray and T. Seewald, *Prototyping vs. Specifying: A Multi-Project Experiment*, UCLA Technical Report, Computer Science Dept., University of California, Los Angeles, CA, 1982.

[8] B. Boehm, M. Penedo, A. Pyster, E. Stuckle and R. Williams, "A Software Development Environment for Improving Productivity," *Computer 17*, 6 (June 1984), pp. 30-44.

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

[10] S. Gerhart *et. al.*, *An Overview of AFFIRM: A Specification and Verification System*, USC Information Sciences Institute, Marina Del Rey, CA, 1980.

[11] G. Gladden, "Stop the Life Cycle, I Want To Get Off," *Software Eng. Notes 7*, 2 (Apr. 1982), pp. 35-39.

[12] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1978.

[13] N. Leveson, A. Wasserman and D. Berry, *BASIS: A Behavioral Approach to the Specification of Information Systems*, Technical Report, Computer Science Dept., University of California, Irvine, CA, 1981.

[14] D. McCracken and M. Jackson, "Life Cycle Concept Considered Harmful," *Software Eng. Notes 7*, 2 (Apr. 1982), pp. 29-32.

[15] B. Meyer, "On Formalism in Specifications," *IEEE Software 2*, 1 (Jan. 1985), pp. 6-26.

[16] D. Nelson, A Software Development Environment Emphasizing Rapid Prototyping, in *Approaches to Prototyping*, R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Zullighoven (ed.), Springer Verlag, New York, NY, 1984, 136-151.

[17] D. Parnas, "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," *Proc. IFIPS*, 1969, pp. 379-385.

[18] 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).

[19] L. Robinson, *The HDM Handbook*, SRI Project 4828, SRI International, Menlo Park, CA, 1979.

[20] D. Shewmake and A. Wasserman, *RAPID User Manual*, Laboratory of Medical Information Science, University of California, San Francisco, CA, 1980.

[21] S. Squires (ed), "Special Issue on Rapid Prototyping," *Software Eng. Notes 7*, 5 (Dec. 1982).

[22] 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.

[23] 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.

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

[25] S. Wartik, *A Specification of FILLIN Using The Darwin Requirements Model*, Research Memorandum, Dept. of Computer Science, University of Virginia, Charlottesville, VA, 1985.

[26] A. Wasserman, *The User Software Engineering Methodology: An Overview*, Technical Report 56, Laboratory of Medical Information Science, University of California, San Francisco, CA, 1981.

[27] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Trans. on Software Eng. SE-8*, 5 (May 1982), pp. 250-259.