Towards a Framework and Procedure for Specifying User Interfaces

Paul Chesson, Lorraine Johnston, Philip Dart

Department of Computer Science, The University of Melbourne, Parkville, Australia, 3052
{chesson,ljj,philip}@cs.mu.OZ.AU

Abstract: This paper presents a framework for specifying user interface dialogue. The framework supports the integration of a dialogue specification with external environments such as the application core. Some results are presented concerning the development of a procedure for constructing a dialogue specification within the framework. This procedure considers meaningful pieces of work in the application environment, and relates the user interface design requirements to the various layers in the framework.

Keywords: dialogue specification, user interface model, development process

1. INTRODUCTION

User interface dialogue specification is concerned with describing the structure of dialogue between an interactive system and its users. The process and result of formally specifying a user interface is aimed at facilitating the understanding and communication of the design between the participants in software development. Specifications can also enable developers to analyse features related to the usability of an interface.

When applying languages to particular specification problems, procedural guidance is rarely given. Such guidance would reduce the time needed to learn the language, and avoid the language being used inappropriately (Monk et al., 1993). There is a need for representation techniques to be closely tied to, and interleaved with, the mental design process. Control over complexity can be achieved with carefully applied methods and sound principles (for example, abstraction and stepwise refinement) which have been successfully applied in other design environments (Hartson and Boehm-Davis, 1993).

This paper presents some results concerning the development of a framework and procedure for specifying the dialogue of user interfaces. User interface specification

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35349-4_22](https://doi.org/10.1007/978-0-387-35349-4_22)

S. Chatty et al. (eds.), *Engineering for Human-Computer Interaction*

© IFIP International Federation for Information Processing 1999

languages differ widely in the types of components used, and in the way in which these components are combined to form a complete description of the interface (Gray and Johnson, 1995). For this reason, no single procedure can hope to be applicable to all languages. Although the ideas presented here have been developed for use with the user interface specification language FLUID (Chesson and Johnston, 1996) in mind, they should be applicable to other languages which possess similar characteristics in their underlying model.

Section 2 introduces the high level model and concepts used for the specification of interactive systems. Section 3 presents a framework for specifying user interface dialogue, based on the level of abstraction of its requirements. Section 4 addresses the issue of specifying the interface between user interface dialogue and the external environments needed to provide application and computational functionality. Finally, work towards the development of a procedure for specifying the behaviour of an interactive system using the given framework is outlined in section 5.

2. MODELLING INTERACTIVE SYSTEMS

FLUID (Formal Language for User Interface Dialogue) is a language for describing the interactive dialogue between a user and an application. The model upon which FLUID is based (see Figure 1) treats the user interface as a processing unit which regulates the flow of control and data between the user and application environments. The user environment consists of one or more users, the physical devices which they use, and any information presented to them. The application environment provides the underlying functionality required by the users in order to perform their tasks.

Events are initiated by changes in the user or application environments. Actions describe the effects of these changes on the corresponding environment. Events and actions are modelled using a name and a list of parameters which describe the data associated with the event or action, for example, **move_cursor(POS1, POS2)**. Distinct instances of events which have the same name (regardless of whether they have matching data values) are said to be of the same *type*. Event types may be represented by using descriptive variable names to represent its data elements. To differentiate constant data values from variable names, variable names begin with a lower case letter, for example, **move_cursor(from_position, to_position)**.

The mapping from events to actions is defined by the dialogue specification and the state of the interface at the time when it receives its events.

The conceptual model of FLUID corresponds to the Seeheim model (Pfaff, 1985) for User Interface Management Systems (UIMS) in the following manner: the set of user events and presentation actions required by a user interface defines the lexical units of the presentation component of the Seeheim model. Similarly, the set of application events and application actions correspond to the application interface component of the Seeheim model. Lastly, the FLUID dialogue specification corresponds to the dialogue control component of the Seeheim model. The dialogue control component will be further decomposed in section 3.

Since a user interface can be viewed as a unit which reacts to its environments, it seems natural to also model the internal components of a user interface in the same manner. The idea of scaling down a user interface model into its components has also

Figure 1: The conceptual model of FLUID

been applied by Palanque and Bastide who used the Seeheim model as a basis for specifying Interactive Co-operative Objects (ICOs) (Palanque and Bastide, 1994).

In the case of FLUID, a specification is divided into units called *sub-dialogues* which receive incoming events and can produce outgoing events and actions. This abstraction results in components that resemble instances of Paternó's LOTOS interactor model (Paternó and Faconti, 1992) which reacts to events via its input and output gates. In Paternó's model, each interactor represents a user interface element which transfers data across multiple levels of abstraction. However, the FLUID sub-dialogue represents a simple design requirement describing part of the behaviour of the system. Such a requirement is modelled by a constraint or rule, similar to those used in ERL (Hill, 1986) or PPS (Olsen, 1990), without any explicit object modelling.

Sub-dialogues are expressed using a combination of tables and simple Coloured Petri nets, allowing for more expressive rules to be formulated than could otherwise be obtained by using simple production rules. The sub-dialogue components process events according to particular aspects of the state of the user interface relating to the requirement being specified. As a result of processing, a sub-dialogue may produce new events called *contextual events*. In some cases, contextual events can be thought of as the same occurrences in a different context. For example, if a cursor is inside an icon, then the event of pressing a mouse button produces a new event indicating an icon is selected. In other cases, the new events being produced can be thought of as new occurrences. For example, if an icon is selected, then the event of selecting another icon produces a new event indicating that the first icon becomes unselected.

The method by which requirements are specified as sub-dialogues is an important issue, but is not addressed in this paper. Instead, these details are abstracted by treating sub-dialogues as black boxes which react to, and produce, events. The rule governing each sub-dialogue are expressed here using natural language.

3. THE THREE-LAYER DIALOGUE SPECIFICATION MODEL

Requirements for interactive systems can cover a wide range of conceptual levels from low-level user events such as mouse movements and key presses, to high-level application tasks. Sub-dialogues provide a way of abstracting the details of individual requirements, but sub-dialogues themselves may need to be hidden when dealing with

other requirements at a higher level. The three-layer model proposed in this section aims to provide a framework for the classification and organisation of sub-dialogues. This framework promotes internal consistency by restricting the types of events and actions, and the ways in which they are used.

The use of layers representing different levels of abstraction is similar to the language stage concept (Hoffner et al., 1989). Hoffner et al. claims that the use of language stages, if carried out wisely, offers a number of benefits of modular design. In their model, the divisions (called language stages) are specifically defined for each system, rather than having precise boundaries which are applicable to all systems. In our three-layer model, sub-dialogues within the layer are responsible for the translation and interpretation steps that would be needed between the language stages.

3.1 Overview

Figure 2 presents the division of the user interface component in figure 1 into three layers: symbolic, semantic, and pragmatic. In addition, each layer can be divided into input and output streams. All sub-dialogues must be classifiable into one of these streams, in one of the layers.

Each layer can be thought of as a scaled-down version of the model of the user interface in figure 1. Actions are treated in the same manner as events, and are only differentiated from events because they happen to terminate at either the user or application environments and are not processed by the user interface.

Each layer in the model consists of sub-dialogues which are responsible for processing and producing events between two levels of abstraction. Consider the semantic layer which specifies the transformations required between symbolic events and semantic events. Sub-dialogues in the input stream take symbolic and/or semantic events as inputs, and can produce semantic events as outputs. Sub-dialogues in the output stream take semantic events as inputs and can produce semantic and/or symbolic events as outputs.

3.2 Symbolic layer

The symbolic layer deals exclusively with the specification of the user interface behaviour, without any reference to the domain of the application. The layer has two

Figure 2: The three-layer dialogue specification model

main purposes. The first purpose is to provide a means of establishing a well-defined set of events which can be used to describe interaction with the user interface. This has a similar role to the set of subroutines used to describe the programming interface to a user interface toolkit.

The second purpose is to allow the specification of the behaviour of the user interface platform, itself. For example, the behaviour of a scrollbar widget can be specified in terms of clicking on its arrow buttons and dragging a slider. This level of specification is similar to that provided by Interaction Object Graphs (IOG) (Carr et al., 1994). In fact, since IOGs interact with external environments via event and data arcs and FLUID events have a data value attached to them, it is possible to incorporate the IOG formalism into the three-layer model by treating IOGs as one large sub-dialogue (with no separation of input and output streams).

In cases where a standard user interface platform has been specified, the symbolic layer will not need to be redefined when the same platform is re-used in other systems. In other circumstances, it may be acceptable to abstract the entire layer to a set of symbolic events, without detailed consideration of their underlying behaviour.

3.3 Semantic layer

The semantic layer deals with the relationship between the user interface and the meaning of its components and dynamic behaviour. Sub-dialogues in the input stream of the semantic layer typically take symbolic events representing user interactions with the interface, and describe their meaning in terms of the application domain. However, semantic events still convey some meaning in relation to the interface itself. Two alternative interfaces for the same application may vary in how their tasks are performed, or how they present information. Each interface is likely to employ a different set of semantic events. For example, a text editor may allow a user to delete a number of consecutive lines in a number of ways. A command line interface may use a semantic event such as **delete_lines(first_line, last_line)**. A graphical interface may use a set of semantic commands to delete an arbitrary block of text, such as: **mark_start_block(pos)**, **mark_end_block(pos)**, and **delete_marked_block()**.

Sub-dialogues in the output stream of the semantic layer are typically responsible for rendering the information conveyed by semantic events to the user. These semantic events may be generated by sub-dialogues in the input stream of the semantic layer (for example, highlighting the marked text when a block is selected), or from the results produced at a higher level (for example, deleting a block of marked text after it has been established that the document can be changed).

3.4 Pragmatic layer

The pragmatic layer deals with the relationship between the meaning of the information conveyed in the user interface, and the work being performed in the application domain, i.e. the practical significance of the information. Sub-dialogues in the input stream of the pragmatic layer typically take semantic events representing operations performed by the user, and specify how they are used to perform meaningful operations (pragmatic events) and tasks (application actions) in the application domain. Pragmatic events and application actions express interaction purely in terms of the application domain, independent of the user interface.

The set of application events and actions provides a similar role to the programming interface to the core functionality of the application. The definition of these events and actions should be exactly the same for any user interface which provides access to the same functionality and presents the same information to the user.

Sub-dialogues in the output stream of the pragmatic layer are typically responsible for determining how high level output from an application is visualised with respect to the user interface components. For example, the application event used to indicate that the request to move a block of text was successful, may produce two semantic events to handle this: `copy_document_text(start_pos, end_pos, new_pos)` copies a block of text in the document presented in the user interface, while `delete_document_text(start_pos, end_pos)` is used to delete the original block of text.

4. MODELLING INTERFACES WITH THEIR ENVIRONMENTS

There are a number of environments which are external to the user interface component of the dialogue specification model. In addition to the user and application environments in the conceptual model of FLUID, a computational environment may be required to perform generic computations which are not possible using a particular dialogue specification language. An overview of the interactions between the user interface and its external environments is presented in section 4.1.

Although the full details concerning the behaviour of external environments are not part of the user interface dialogue specification, the boundary between the two should be well defined. Each piece of functionality (typically, a single computation or procedure) supported by an environment is specified as a distinct unit called an *environment interface*. An environment interface is defined by the set of event types passed between the interface and the environment, and the constraints between them.

The nature of the data passed from an environment interface can be classified according to a number of characteristics which it may possess. Section 4.2 discusses some common features found in such data and suggests how the transfer of this data can be modelled by discrete events. Finally, section 4.3 examines the issue of specifying constraints in the sequencing of events in an environment interface.

4.1 Types of interface environments

Dialogue specification languages are typically designed to be expressive enough to accommodate a significant proportion of situations that may be encountered when describing the behaviour of user interfaces. There is usually some tradeoff between the expressiveness (or computational power) of a language, its simplicity of use, its readability, and its formality. For situations when the specification language is unable to describe the computational processing required by the interface, there needs to be some way of abstracting this functionality. Even in cases where the language can express such functionality, it may still be desirable to abstract these details.

Computations may be performed in any of the three layers in the dialogue specification model. The processing of computations is modelled by interaction with a computational environment. The computational environment represents some external entity

which responds to queries. Although the computational environment may be implemented as part of the application code, unlike the application environment, it has no perceivable state and any interaction should be purely functional. In most cases, this can be thought of as occurring by the sub-dialogue sending a *computational query event* to the computational environment, and the computational environment returning one or more *computation response events*. The exchange of data involved in a computation is localised to a single sub-dialogue and as such, is not shown in figure 2.

In addition to generic computations, semantic and pragmatic sub-dialogues may require application-specific information. If this information is not required to perform work directly, but is used within the interface to assist the user in completing tasks, then an *application query* is required. An example of an application query is obtaining a list of files currently open in an application so that one of them may be selected for some purpose. This information is contained within the state of the application, and thus requires the application environment to be queried. Such a query is modelled in the same fashion as a computation, using *application query events* and *application response events*.

Finally, application actions which indicate requests for some action to occur in the application environment, are sent from the pragmatic input sub-dialogues to the application environment. Any subsequently related responses from the application environment are sent to pragmatic output sub-dialogues.

For the purposes of specifying the interface between a sub-dialogue and an environment, all three cases will be modelled in the same manner. Computation query events, application query events, and application actions, will be classified as instances of *stimulus events*. Computation response events, application response events, and application events responding to application actions, will be classified as instances of *response events*.

4.2 Characteristics of environment interface responses

After a stimulus event has been sent to an environment interface, the response of the environment depends on the data sent with the stimulus event and the state of the environment. In many cases, the response can be modelled using a single response event to represent the final result. However, in other cases, a sequence of several response events may be required to model the response accurately. In this section, three such cases will be examined. No claim is made that these cases cover all instances where the transfer of data cannot be represented by a single event, but it is expected that the solutions offered will serve as a useful guide for handling a number of common cases in a consistent manner. Lastly, the proposed use of standardised event types will be briefly discussed.

4.2.1 Case 1: Delayed response

In many interactive systems, some operations will involve periods where the user will be waiting for the application to provide a response. This delay can be caused by factors such as a complex computation being performed, or information having to be transferred over a network. In such cases, a response event should be returned by the environment to indicate that the subsequent response may be delayed. This initial response event can be handled by the dialogue specification which indicates to the user that a certain piece of data is awaited.

For example, the application environment of a web browser may not be able to respond with the contents of a requested page immediately, so a response event **start_page_contents()** should precede the delayed event **send_page_contents(data)**. The **start_page_contents()** event may be subsequently rendered to indicate to the user that a page contents transfer is in progress.

4.2.2 Case 2: Segmented response

In addition to being delayed, the actual response from the application environment can be spread over a period of time. This can occur when a large unit of data is progressively transferred, and it is useful to specify the behaviour of the interface as each segment of data is received, or to allow pre-emptive interaction to be modelled. Each segment of data is represented by a separate event, along with additional information regarding the state of the response (for example, time elapsed, time remaining, or percentage complete). The complete set of data segment events should occur between start and end events to indicate when the data is being sent, and when all the data has been sent. Once again, these events can be used to provide cues to the user regarding the progress.

Continuing with the web browser example, the progressive transfer of a web page can be treated as a segmented response using the set of event types: **start_page_contents()**, **send_page_contents(data, percent_trans)**, and **end_page_contents()**, where **data** refers to the segment of the page contents being transferred, and **percent_trans** refers to the total percentage of the page which has been transferred including the current segment.

4.2.3 Case 3: Multiple response

An environment may respond with several pieces of data, each of which are independent from each another. In such cases, start and end events are useful in indicating the commencement and termination of the complete response. If the instances of data themselves are delayed or segmented, then each instance will have its own start and end (in the case of segmented data) events.

When instances of data are of the same type, their corresponding events will also be of the same type. If each instance of data needs to be referred to at a later time, then it requires a unique identifier. One case where this is necessary is when the data from each instance is segmented. A unique identifier is required by the user interface to be able to associate the segment with the correct piece of data.

Once again using the web browser example, a request for a web page may result in a number of components being transmitted such as images, sounds and applets. Using the URL of each of these components as a unique identifier, the set event of event types needed to model multiple, segmented response is: **start_page_contents()** and **end_page_contents()** for the response, and **start_url_contents(url)**, **send_url_contents(url, data, percent_trans)**, and **end_url_contents(url)** for each of the components.

4.2.4 Standardised event types

The use of standardised event types appears to be a useful approach in dealing with particular classes of specification problems. When modelling response characteristics for environment interfaces, events representing the transfer of data can be enclosed within “start” and “end” events to model delays in response and the termination of seg-

mented data. Similar uses of standard events for dealing with specification problems has been found in other ongoing work related to the specification of sub-dialogues. Different types of states (for example, single values, sets, and mappings) may have their values changed via “change”, “set” and “unset” event types, whose behaviours depend on the particular types of states to which they are applied.

4.3 Temporal constraints

In addition to defining the events required to formulate an environment interface, it is necessary to define the constraints between response events in cases where a sequence of more than one event may be used in response to a stimulus event. These constraints enable software developers to understand the possible behaviours which can be expected of an environment. In turn, this information can be used to devise scenarios for test cases, and verify properties of the interface based on the specified behaviour of its environment.

The possible set of responses in an environment interface can be defined by a grammar. This grammar may be represented using any number of formalisms, including BNF, LOTOS, temporal logic, or Petri nets.

Consider the example of a web browser with an environment interface for the process of requesting and serving the contents of a web page, using a multiple, segmented response. The stimulus event type representing the request for a page by the user is **request_page(url)**. In addition to the response events introduced in section 4.2.3, two error event types will be introduced: **connection_error(url)** to represent errors which occur before data from a URL is sent, and **transfer_error(url)** to represent errors which occur while data from a URL is being sent.

Figure 3 specifies the temporal constraints on the responses from the application environment using a Coloured Petri net (Jensen, 1992). When the stimulus event is received (1), the start event for the transfer of page contents is produced (2). A token is created to represent the URL of the page to be retrieved (3). Next, either a connection error occurs (4), or the contents of the URL start being retrieved (5). Then, the application can either indicate that a transfer error has occurred (6) or that the contents of the URL have been completely sent (7), or it can send some of the contents (8). The variables **data** and **percent_trans** represent data which is produced by the application and passed to the user interface. If the application sends some contents, additional URLs may be found in the new content which will commence being retrieved (9). These URLs are handled in the same way as the initial one (10). The double-headed arc indicates that multiple tokens may be created, each representing a reference to a new URL. Lastly, the application indicates that the retrieval of the contents of the page has been completed when no further URLs are pending (11).

5. A SPECIFICATION PROCEDURE

The purpose of this section is to present work undertaken towards a structured approach to specifying user interfaces within the framework presented in section 3. The procedure to be presented focuses only on the activity of dialogue specification (i.e. the three layers of the model in section 3), and as such, requires that the developer

Figure 3: Temporal constraints for a web page environment interface

specifying the user-interface has an understanding of the section of the design being represented. However, this does not mean that the entire design needs to be completed before dialogue specification can commence, nor does it mean those parts that have been specified cannot be changed. Specification is organised around pieces of functionality called *work units* which are progressively refined and related to the user environment. The developer may choose to postpone specification of part or all of a work unit at any point, and then resume or begin specification of another work unit.

Two constraints are imposed on the development activity. First, any new requirement must either correspond to a new work unit, or be needed as a consequence of the requirements already in the specification. Second, any requirements which depend on a modified requirement and are no longer correct or consistent, must either be immediately updated or deleted.

5.1 Work decomposition

Since the proposed framework uses rule-based sub-dialogues as its primary unit of abstraction, it is more appropriate to approach the specification process in terms of functionality rather than objects. A starting point for dialogue specification which is commonly recommended is task analysis. One example of this is Dialogue Specification Notation (DSN) (Curry and Monk, 1995) which advocates the use of a task model to identify those parts of the model suitable for dialogue specification. While task ana-

lysis techniques can be useful in driving the high level dialogue of an interface, we will begin at an even higher level by examining the pieces of work which need to be performed by the system. In other words, we begin with the question, “What is the user supposed to be able to do with the system?”, rather than, “How is the user supposed to use the system to do it?” This is similar to the distinction between the goal and task levels of Nielsen’s virtual protocol model for HCI (Nielsen, 1986).

Specification commences with the designation of a piece of work which the system is required to perform. This piece of work is referred to as a work unit. The motivation in using work units is to promote a well-defined separation between the user interface and the application, and to help determine whether support for the users’ access to the underlying functionality has subsequently been specified.

The functionality represented by a work unit should require some action to be performed in the application environment, and be independent of the user interface used to perform or control the work. The performance of a work unit will not necessarily have some perceivable effect on the user interface. This is similar to the distinction between the Result and Display components of the PIE model (Dix, 1991). Work unit functionality can be modelled by one or more stimulus and response events at the application environment interface (section 5.2).

In the web browser example, work units include: retrieving a web page, printing a web page, and locating a text pattern in a web page. Clearing and adjusting the cache would also be included if the browser makes these facilities available to the user, since they affect the application.

Examples of functionality which would not be considered work units include changing preferences, scrolling around a page, and interrupting the loading of a page. Changing preferences (such as whether images should be displayed) affect the work of retrieving and viewing a page, but do not achieve any results by themselves. Scrolling around a page is another way of controlling the results of work – in this case, how it is viewed. Interrupting the loading of a page controls the work of retrieving the page, and can be thought of as a *user exception* (Curry and Monk, 1995) which would be treated as a stimulus event for the page retrieval work unit.

Functionality related to bookmarks and history is less clear, as it may not be apparent whether it should be treated as part of the interface or the application. We judge that these mechanisms are a means of performing the work or retrieving a page, and as such, should be treated as part of the interface. Events representing the use of such mechanisms to retrieve pages would be classified as semantic events, and not part of a work unit. Dialogue relating to the manipulation of bookmarks would be specified in the semantic layer. The possible use of an external environment to support the loading and saving of the bookmarks without using the application environment, is briefly discussed in section 6. Nevertheless, even if a different division between the interface and application is chosen, the method is still applicable as long as consistency is maintained.

5.2 Application environment interfaces

After a work unit has been identified, an environment interface is used to specify the communication between the pragmatic layer of the user interface and the application environment. The developer firstly determines what information is required to be returned to the user upon successful completion of the work. For example, the

result of locating a pattern in a web page can be represented by the application event **found_pattern(start_pos, end_pos, num)** where **start_pos** and **end_pos** represent the start and end locations of the pattern found in the web page, and **num** indicates the total number of times the pattern is found.

Next, each piece of information required to be provided by the user in order to complete the work, needs to be identified. Typically, only one set of information is needed to complete the work, and this can be represented by a single application event type which combines pieces of information. However, in some cases where the information is significantly disjoint, it may be more convenient to use multiple application events. For example, the act of locating a text pattern in a web page may be initiated by the application event **find_external_pattern(pattern, case, pos)** or **find_internal_pattern(start_pos, end_pos)**. In the first case, the search for **pattern** is commenced from the position **pos** in the web page. In the second case, the pattern to be searched for is defined by the text in the web page itself between **start_pos** and **end_pos**.

Next, any possible system errors or exceptions resulting from the work being performed are considered and represented by application events. Consideration should be paid to the complete set of application actions. Any irregular inputs should either be handled directly, or assumptions that they will not occur should be explicitly documented for later verification. For example, one assumption relating to **find_internal_pattern** might be that **start_pos** occurs before **end_pos** in the web page. The final set of application events are derived by considering their characteristics, and any temporal constraints between the application events should be documented (see section 4).

5.3 Pragmatic layer specification

Having defined the application environment interface, specification of the related pragmatic layer sub-dialogues is commenced in order to describe in abstract terms how the data associated with the work unit will be sent to the application and presented to the user.

The process of specifying the input stream of the pragmatic layer will involve defining how the user is able to provide the information required by the application actions and then invoke them. The developer will need to consider the final state that the user interface needs to invoke the application event, and then specify how each of its sub-states is reached. This refinement continues until the events involved are no longer completely independent of the user interface, and further consideration of the user interface platform is required (section 5.4). In the internal pattern searching example, the following pragmatic input sub-dialogue may be defined:

Example 1

If a request to find a pattern is made (semantic event **find_pattern()**) by designating a pattern on the web page from **start_pos** to **end_pos**, a request to find the pattern is sent (application action **find_internal_pattern(start_pos, end_pos)**).

The sub-state regarding a pattern being designated is defined as follows:

Example 2

If a pattern is designated from **start_pos** to **end_pos** in the web page (semantic event **designate_pattern(start_pos, end_pos)**), then a pattern has been designated until the pattern is cleared (semantic event **clear_pattern()**).

The process of specifying the output stream of the pragmatic layer will involve defining the operations to be performed by the user interface to present the information provided by the application environment to the user. In doing this, the developer should begin with the application events and refine them until the operations required to be performed are no longer completely independent of the user interface.

In the pattern searching example, the following pragmatic output sub-dialogue may be defined to specify how text patterns found in a web page are indicated:

Example 3

If a text pattern is found (application event **found_pattern(start_pos, end_pos, num)**) then indicate the pattern to the user (semantic event **indicate_pattern(VIEWING_WINDOW, start_pos, end_pos)**) and indicate the total number of times the pattern occurs (semantic event **indicate_num_patterns(num)**).

5.4 Symbolic layer specification

Before associating semantic events with their presentation components, it is helpful to define these components without reference to the application domain. Each type of presentation component should be identified along with the operations which can be performed on them. The specification process for the symbolic layer is similar in method to that for the pragmatic layer. Each operation on a presentation component should be classified into either the input or the output stream, where it can be progressively refined in a similar manner to application events and actions. Unless a developer is attempting to specify an entire user environment, the operations required can be specified as needed. The level of detail to which the symbolic layer should be specified (in other words, what constitutes a user event or presentation action) can be decided by the specification writer (see section 3.2).

An example of the operation of selecting text with a mouse is shown below:

Example 4

If the user releases the left mouse button (symbolic event **release_left_mouse()**) at position **pos2** while the mouse is being dragged from position **pos1**, then the text from **pos1** to **pos2** is selected (symbolic event **select_text(pos1, pos2)**).

5.5 Semantic layer specification

The semantic layer is defined last of all, linking the high level tasks and application information from the pragmatic layer, to the low level user interface operations and presentation data from the symbolic layer. Specification of the input stream requires the consideration of the semantic events defined in section 5.3, and defining the conditions under which such events can occur. The following example demonstrates the mapping between the selection of text and its use in designating a search pattern:

Example 5

If a segment of text from **start_pos** to **end_pos** in the web page **VIEWING_WINDOW** is selected (symbolic event **select_text(start_pos, end_pos, VIEWING_WINDOW)**), then a pattern is designated between these two positions (semantic event **designate_pattern(start_pos, end_pos)**).

In the output stream, similar mappings can be specified to show how semantic events are mapped to presentation concepts. For example, **indicate_pattern** maps to highlighting of text, and **send_url_contents** maps to cumulatively creating and displaying new presentation objects.

6. CONCLUSIONS AND FURTHER WORK

This paper presented a model which provides a framework for the specification of user interface dialogue. The model consists of three layers (symbolic, semantic, and pragmatic) which constrain requirements to well-defined levels of abstraction. Environment interfaces are used to specify communication with the application core, and to support generic computations.

Work towards the development of a procedure for constructing a user interface dialogue specification within this model was presented. Use of the model aims to promote the construction of a specification which possesses a consistent structure across multiple levels of abstraction. The procedure aims to allow flexibility in the level of detail to be specified for various sections, and allow the writer to postpone and resume specification of any section, at any time.

The procedure should serve as a useful guide as to how to progress, to ensure that the writer knows how to begin or what to do next. While providing guidance, it should remain flexible enough to allow refinement to progress in the manner chosen by the writer. For each step performed, the amount of information required from outside the scope of what is being immediately specified should be minimised.

The procedure outlined is driven by work units which represent meaningful pieces of work in the application environment. The interface for each work unit is defined, and related requirements are specified for the three layers, in the following order: pragmatic, symbolic, then semantic.

Specification of individual requirements is outside the scope of the procedure presented and is dependent on the actual language used. Two omissions from the procedure are noted here. First, not all user interface functionality can be derived from work units. Operations which relate purely to the user interface, such as changing its configuration, must be specified separately in the semantic layer. Data related to this functionality which is stored or retrieved from external sources (for example, files) may need to be modelled using an additional environment accessible to the semantic layer. We are currently considering the use of work units for user interface tasks.

The second recognised omission is that the process does not have provision for semantic feedback for events which are not part of the dialogue required to perform a work unit. The strategy we are currently using to detect likely instances of such cases is to check where symbolic events are used to change the state of a sub-dialogue in the semantic layer, and to consider if additional semantic feedback is appropriate.

Acknowledgements

The authors wish to thank the anonymous referees for their valuable comments on an earlier draft of this paper.

REFERENCES

- Carr, D., Jog, N., Kumar, H., Teittinen, M., and Alberg, C. (1994). Using Interaction Object Graphs to specify and develop graphical widgets. Technical Report CAR-TR-734, University of Maryland, College Park, MD.
- Chesson, P. and Johnston, L. (1996). FLUID: Specifying data flow and control for user interfaces. In Yong, L., Herman, L., Leung, Y., and Moyes, J., editors, *Proceedings of the First Asia-Pacific Conference on Computer Human Interaction*, pages 171–180, Singapore. Information Technology Institute.
- Curry, M. and Monk, A. (1995). Dialogue modelling of graphical user interfaces with a production system. *Behaviour and Information Technology*, 14(1):41–55.
- Dix, A. (1991). *Formal Methods for Interactive Systems*. Academic Press, London.
- Gray, P. and Johnson, C. (1995). Requirements for the next generation of user interface specification languages. In Palanque, P. and Bastide, R., editors, *Proceedings of The Design, Specification And Verification Of Interactive Systems*, pages 113–133, Toulouse. Springer-Verlag.
- Hartson, H. and Boehm-Davis, D. (1993). User interface development processes and methodologies. *Behaviour and Information Technology*, 12(2):98–114.
- Hill, R. (1986). Supporting concurrency, communication, and synchronization in human-computer interaction - the Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179–210.
- Hoffner, Y., Dobson, J., and Iggulden, D. (1989). A new user interface architecture. In Sutcliffe, A. and Macaulay, L., editors, *People and Computers V, Proceedings of HCI'89*, pages 169–189, Nottingham. Cambridge University Press.
- Jensen, K. (1992). *Coloured Petri Nets, Volume 1: Basic Concepts*. Springer-Verlag, Berlin.
- Monk, A., Curry, M., and Wright, P. (1993). Why industry doesn't use the wonderful notations we researchers have given them to reason about their designs. In Gilmore, D., editor, *User-centred requirements for software engineering*, pages 253–261, Berlin. Springer-Verlag.
- Nielsen, J. (1986). A virtual protocol for computer-human interaction. *International Journal of Man-Machine Studies*, 24:301–312.
- Olsen, D. (1990). Propositional production systems for dialog description. In Chew, J. and Whiteside, J., editors, *ACM Annual conference on Human Factors in Computing Systems*, pages 57–63, New York.
- Palanque, P. and Bastide, R. (1994). Petri net based design of user-driven interfaces using the interactive cooperative objects formalism. In Paternó, F., editor, *Design, Specification and Verification of Interactive Systems*. Springer-Verlag.
- Paternó, F. and Faconti, G. (1992). On the use of LOTOS to describe graphical interaction. In Monk, A., Diaper, D., and Harrison, M., editors, *HCI'92, People and Computers VII*, pages 155–173, London. Cambridge University Press.

Pfaff, G., editor (1985). *User Interface Management Systems, Proceedings of IFIP/EG Workshop on UIMS*, Seeheim, Federal Republic of Germany. Springer-Verlag.

Biographies

Paul Chesson is completing a Ph.D. at the University of Melbourne, Australia, in the area of formal methods in HCI. His research involves the development of a dialogue specification language for interactive systems, and a supporting method and tools for its use. He holds a B.E. from Melbourne in software engineering.

Lorraine Johnston is a Lecturer at the University of Melbourne, Australia. Her research interests include usability engineering and software engineering education. She holds a BSc(Hons) and a PhD in quantum chemistry from the University of Queensland, and a DipComp from the University of Melbourne.

Philip Dart is a Senior Lecturer and the software engineering program coordinator at the University of Melbourne, Australia. His research interests include requirements engineering and formal methods. He received a BSc(Hons) from the University of Queensland and a PhD from the University of Melbourne.

Discussion

Gilbert Cockton: Is the line in the middle of the diagram a bus?

Lorraine Johnston: Yes

Len Bass: How does this notation “enable” usability evaluation of specified interfaces?

Lorraine Johnston: Some properties of usability such as deadlock detection or reachability can be analyzed via this approach. The model itself does not specifically support the production of a usable system, but a clear understanding of the design is more likely to minimize defects.

Ken Fishkin: Can you help me understand what information is known at what level of your pipeline? E.g. if I have a CAD application, and the user says, “show me parts of the circuit that aren’t grounded,” what does the symbolic layer know about this request? Does it know about “ground” and “circuits”

Lorraine Johnston: No, it knows about buttons and coloured rectangles. The input event is passed on to the semantics layer, which is responsible for mapping user’s actions into meaningful input to the application.

Christian Gram: Is your work-unit the same as a task? How does one identify work units/tasks?

Lorraine Johnston: No. A work unit is a sub-task, a small task requiring only little interaction, being short and simple. One can choose whatever level of work unit seems appropriate for the functionality required.

Laurence Nigay: Your approach is pragmatic to symbolic to semantic. Why not pragmatic to semantic to symbolic, or vice versa?

Lorraine Johnston: The pragmatic layer can easily be defined, as we know what the application requires. The user interface elements can easily be defined, or may already have been defined (reuse). The semantic layer does the mapping between the two, so it seems reasonable and logical to treat them in this order.

Morten Harning: How does this scale to large complex systems? It seems very detailed.

Lorraine Johnston: It would not be used for all systems. However, it does promote reuse of user interface components. One would normally expect to use it only in mission-critical situation.

John Grundy: How does this architecture compare to MAMP? Is semantics mapping, or mapping and abstraction?

Lorraine Johnston: Maybe there is some similarity: pragmatic = model, semantics = abstraction/mapping, symbolic = presentation. MAMP’s basis is objects; the basis of this architecture is requirements or rules.

Prasun Dewan: Could there not be an arbitrary number of semantic layers between the symbolic and pragmatic layers?

Lorraine Johnston: Three layers are found to be natural and sufficient. It would be possible to provide more structure within the layers, but three are necessary and sufficient.