Towards Customized User Interface Design Environments

Farah Arefi  Masoud Milani  Christian Stary

Florida International University
School of Computer Science
Miami, Florida 33199

Abstract

The advantages of declarative end user task specification and those of automatic language tool generation are combined to produce individual user interface design environments that are maximally adapted to the specific requirements of the problem on hand. The development of the target design environment proceeds in two steps. First, a visual syntax-directed editor for a declarative task-oriented specification language is generated from formal specifications. The editor is then embedded with the functionalities required for the target design environment - the result is a customized user interface design environment. The initial results from experimenting with customized user interface design environments using our visual language specification framework and its editor generator are discussed.

1 Introduction

The most crucial part of any computerized system is its user interface. The user interface of an application should present the implemented functionalities (end user tasks) to the end user in such a way, that he/she is effectively supported. In order to be effective, the user interface should not be a bottleneck - it must be easy to learn and easy to use. Despite the recent advances in concepts, design aids and specification environments, the ease of learnability and usability have remained a major issue in user interface development. Moreover, with the requirements stemming from more and more sophisticated work environments and an ever increasing variety of interaction media, the necessity for effective interfaces for application development environments becomes a key factor to their success. Therefore, the orientation of the interfaces of such environments towards the needs of the entire development team (system analysts, designers, and programmers) should result in the increased productivity of the development team as well as the success of their products. The incorporation of the end user tasks in the development environment should provide for relatively easy development of effective user interfaces for several reasons:

- The layout of system functions on the screen is not the only aspect of interface design - the problem domain data which are essentially task fragments should also be considered. Together with presentation types (interaction media) the latter are the backbones any task-oriented user interface development effort[12].

- End users are only focused on their tasks that should be handled as transparently as possible[4]. In order to achieve this transparency and also to cope with complex work environments the end user tasks and their organizations must be explicitly represented.

- Clearly, providing direct task support for end users will result in ease of use, learnability, and transparency of task accomplishment (i.e. the acceptance of a computer applications). Direct user support, however, can be realized only by correlating tasks with problem domain data as well as interaction media in a flexible way.

- The establishment of appropriate interaction modes, such as direct manipulation, requires the use of already developed interaction media such as windows, mouse interactions, etc. Thus, the user interface development team should consider all relevant interaction media and correlate them to establish individual interaction modes. Furthermore, they have to be related to the problem domain data by specifying how problem domain data can be manipulated on the screen[8].


1-1
Task-Oriented User Interface Specification. A grasp of the end user's mental models helps the designer to build systems that are tailored to user tasks [9]. Unfortunately, there is no commonly accepted method for discovering descriptive metaphors for problem domains and task representations[5]. To facilitate the mapping of tasks on system procedures, Seeheim model[11] provides an architectural framework that strictly separates presentation, application, and control components. Unfortunately, mainly because of difficulties in constructing the control component, the strict separation of problem domain data (application component) and interaction media/modes (presentation component) is not applicable to complex work systems[4].

Layering of user interfaces[10] concerns several levels of man machine communication: pragmatic, semantic, syntactic, and lexical interaction that can be specified with algebras, formal grammars, state transition diagrams, etc.[14]. Here, the problem is that the existing gap between the formal or conceptual representation and elementary system procedures prevents adequate specification. Moreover, formal frameworks, e.g. algebras, do not support task-oriented user interface specification.

User Interfaces Tool Kits. Existing environments, such as X-Windows [13] and development tool kits, such as InterViews[7] have been focused only at the implementation level of user interface development and thus, the maintainability of basic building blocks, such as interaction media or global events concepts (precondition – activity – postcondition). In order to utilize such environments for task-oriented user interface development, approaches such as UIDE[3] must be elaborated by specifying more precise types of knowledge concerning task-oriented interaction. But, because of general and therefore vague description of interactions between tasks, problem domain data and interaction media remains implicit.

Design Environments. User interface design environments provide: the entire development knowledge for the generation of the user interface; transparency throughout all design steps; and hopefully automatically generate user interfaces according to the specification. Hence, design environments should provide developers with end-user information and implementation procedures to map end-user tasks to system functions. In contrast to design tool kits, design environments provide knowledge about the design task itself.

Of course, to properly handle the knowledge required for task-oriented design environments must be based on a general and precise specification mechanism. But, most of the existing design environments are restricted to certain programming paradigms, and thus, are limited to the expressive power of such paradigms. Nevertheless, while the existing environments, such as UIDE, do not support effective task-oriented user interface specification, they provide us with the insight, that design environments have to comprise mainly visual information supporting their user group, i.e. designers, analysts, etc.[8]. So it is quite natural, that any user interface design environment should be to a visual one.

The various issues underlying User Interface Design Environments, suggest an inherent requirement for custom-made user interfaces for individual software development projects within the same design environment. Each custom-made interface, of course, beside providing a specification language, must cover all particular development requirements stemming from user tasks, work organisation, etc. Considering the fact that customizing interfaces for each development project might indeed be more difficult than the completion of development project itself, for this approach to be feasible, it must be automated. Such an automation must be based on a general formal specification framework so that restrictions imposed by predefined design environments as well as the need to utilize general but vague design entities are avoided.

Unfortunately, the current approaches to unify different conceptual representations within formal frameworks are limited. In fact, they provide particular insights on how to handle the variety of requirements and the specification of different interaction media generators within a unified framework[2]. There is no indication of how to apply such frameworks to more complex knowledge categories, such as organizational knowledge. While [6] describes how visual languages can be specified by formal grammars, no reference is made to the content handled by the rules of the grammar or the languages. A mechanism to specify User Support Environments is suggested in [15]. Here, since the paradigm of reformulation of already existing specifications is followed the results are not applicable to first-time customized environments.

Our approach to generate customized interfaces for each development project is based on a framework to specify visual languages and a generator system. The generator system accepts, as input, the specification of a visual language and produces, as output, its visual syntax-directed editor. The generation of the target
Figure 1: A part of the IMN for the appointment example. Terms, relationships and clusters are denoted by rectangles, ovals and shaded areas respectively.

development environment proceeds in two steps. In the first step, the specification of a "customized specification language" is used to automatically generate its visual editor. The editor is, in the second step, then further customized so that a complete customized design environment is obtained. That is, we specify both the design language and the functionality of the design environment, so that adequate support for the designer (who is the user of the design environment) is provided.

The remainder of this paper is organized as follows: Section 2 describes a conceptual representation scheme to specify task-oriented user interfaces. The specification framework for visual languages is described in section 3. In section 4 we discuss the integration of the findings in the previous sections, i.e. how this formal framework for language generation can be applied to map the conceptual representation scheme to a visual language, and how to specify and generate a visual environment for the specification of task-oriented user interfaces. The paper is completed with our concluding remarks in section 5.

2 Declarative Task-Oriented User Interface Specification

In this section we discuss the Interaction Management Network (IMN), a specification scheme for the representation of task-oriented design knowledge, introduced by [14]. In comprising three different categories of knowledge (problem domain, interaction media, help strategies) the IMN captures: end user tasks, problem domain data, interaction media, and system functions. Its elements and concepts are:

Terms: Since the IMN has been designed for different user groups, namely potential end users of the target application, system analysts, designers, and programmers, it has to represent all "languages" used by the addressed groups. Potential end users are able to express tasks they expect the target application to support them in terms, such as Make Appointment - see Task Layer in Figure 1. The decomposition into subtasks, such as Make Proposal, leads to the representation of organizational constraints (see has subtask and before relationships) as well as to the identification of problem domain data and interaction media (modes), which are processed by system functions.

Relationships: Regardless of the level of abstraction, terms are correlated. The semantics of relationships is expressed along the links among terms. For example, if tasks are refined, based-on and has-subtask relationships are instantiated. Moreover, for the representation of tasks, all data which have to be visualized have to be related to interaction media, by applying is-represented relationships.

The resulting network has been layered hierarchically to support the decomposition of end user tasks according to the mentioned knowledge categories. Both problem domain data as well as interaction media are located at the intermediate layer. Hence, the IMN contains three levels of abstraction: user tasks) at the top, data/interaction activities (the structure and facilities concerning problem domain data interaction media, and help strategies), and system functions, such as transaction management and window handling, at the bottom.

The top level is accessed by system analysts and potential end users whereas the bottom level contains all information for implementation (accessed by programmers). Designers handle knowledge of all three layers, since their task is the transformation of end user tasks into system functions. In order to maintain transparency throughout the development, the intermediate layer provides this refinement knowledge.

Moreover, if required, the IMN may contain sets of nodes and relationships to cover particular views on the development knowledge. These sets are called clusters. For instance, particular interaction modes
(like direct manipulation), which are constituted by different media, can be represented by clusters.

To exemplify the features of the IMN, let us assume the following appointment scheduling example: Making an appointment consists of the following subtasks: Make Proposal, Discuss Proposal, and Fix Proposal. The communication among appointment partners occurs via sending and receiving e-mails containing appointment data. Thus, the data model of the problem domain has to cover e-mail facilities and the management of appointment proposals which may lead to entries in individual time tables.

Figure 1 shows a part of the IMN for our example. The global activity (Make Appointment) consists of three subtasks, namely Make Proposal, Discuss Proposal, and Fix Proposal. All of these subtasks are handled by problem domain operations, as well as operations on interaction media. For instance, Make Proposal leads to Send Mail (problem domain data operation) and Open Mail Form (operation on interaction medium Form). The intermediate level (problem/interaction domain layer) does not only provide insights into an adequate data model (e.g. a mail consists of a Head and a Body), but also correlates interaction media (e.g. a Form and a Menu are of type Window) to each other. The latter also provides means to specify particular modes.

Once all semantic entities and relationships have been specified in the intermediate layer, the required elementary procedures (like Open Form, Send Body) have to be listed in the bottom layer (system function layer). For the sake of readability we have summarized these elementary procedures in Figure 1 by denoting their global functionality: Window Manager consists of Open Window, Close Window, Front, Rear, etc.; E-mail Facility comprises Send Head, Receive Body, etc.; and so forth. The identified cluster in Figure 1 refers to the interaction domain of the target application, in order to differentiate the interaction media from the problem domain data in the intermediate layer.

3 Visual Language Specification

We specify visual languages and their permitted manipulations with Graph Transformation Systems[1]. Any visual language specified with Graph Transformation Systems is a dynamic object that, as the result of applying editing operations, assumes different forms. Each such form is then defined to be a member of the language. A Graph Transformation System is a 8-tuple

\[ T = \langle LV, LB, LD, S, F, A, \text{CURSOR}, R \rangle \]

where:

- \( LV \) and \( LB \) are sets of symbols called the node label set and the edge label set, respectively,
- \( LD \) is a set of symbols called the don't care node label set,
- \( S \), the starting graph, is a labeled directed graph over \( \langle LV, LB \rangle \),
- \( F \) and \( A \) are non-intersecting subsets of \( LV \) called the menu function node set, and auxiliary function node set, respectively
- \( \text{CURSOR} \in F \) is a distinguished node label,
- \( R \) is a collection of graph transformation rules, \( \langle P_l, P_r \rangle \), where:
  1. \( P_l \) and \( P_r \) are graph patterns over \( \langle LV, LB \rangle \),
  2. No node in \( P_l \) is labeled with the symbol \( \text{CURSOR} \),
  3. \( P_l \) and \( P_r \) each have exactly one node labeled with a symbol in \( F \cup A \).

Once a language is specified in terms of a graph transformation system, its syntax-directed editor is automatically generated. The editor has an internal memory which initially contains the starting graph. At each step during the editing process, the editor presents the user with the list of all transformation rules that can be applied to the internal graph. By definition, no node in the left-part of any transformation rule is labeled "CURSOR", and therefore no rule can be applied to any graph that contains the "CURSOR". Also, there is exactly one node in the right-part of any transformation rule that labeled with a symbol in \( F \cup A \). This label represents the editing operation that is specified by the rule. The user requests the application of a particular graph transformation rule, "E", (i.e. delete, insert, etc.) by changing the label "CURSOR" to "E", thereby facilitating the application of that rule.

4 Generating Visual User Interface Design Environments

In the first step towards the generation of customised design environments the conceptual framework for task-oriented user interface specification (section 2) is specified as a visual language. The specification is then applied to the visual language generator.
so that a visual syntax-directed editor for the specification language is generated. The following illustrates how the visual language is defined. Due to the space limitations only a subset of transformation rules is presented.

\[ T = \langle L_V, L_E, L_D, S, F, A, CURSOR, R \rangle \]

\[ L_V = \{ \text{system-function, task, subtask, problem-domain-date, interaction-medium, add-subtask, add-before, add-problem-domain-date, add-interaction-medium, add-is-based-on, add-is-represented, add-has-part, add-is-a, add-is-handled-by, add-system-function, delete-sub-task, delete-is-a, delete-problem-domain-date, add-after, delete-interaction-medium, delete-system-function, delete-is-based-on, delete-has-part, delete-is-represented, delete-is-handled-by, cursor} \}

\[ L_E = \{ \text{apply, before, has-subtask, is-based-on, is-represented, has-part, is-a, is-handled-by} \} \]

\[ L_D = \{ 1, 2 \} \]

\[ F = \{ \text{add}^2, \text{delete}^*, \text{cursor} \} \]

\[ A = \emptyset \]

\[ CURSOR = \text{Cursor} \]

\[ R = \{ \text{add}^2, \text{delete}^*, \text{cursor} \} \]

The generated editor is considered to be the core of any design environment which is based on the introduced specification language. Producing a customised design environment proceeds by augmenting the editor with the environment's special requirements. Such requirements will include, for example, the following principles:

- For each operation of the specification language, such as \text{insert-problem-domain-date} we need an item in a menu of the visual design environment.
- There is a set of generic screen and file handling operations which concern the layout and the data management of the resulting specification. For example, \text{move} is a screen operation to position elements of the specification language.

It turns out, that the formal framework used for the describing the specification language can be applied to define the requirements of the visual design environments. Once the formal specification of the “ingredients” for the generation of visual task-oriented design environments, namely the specification of a task-oriented specification language as well as the specification of the elements and operations of visual design environments are completed, they are integrated and consequently result in the desired customized design environment. The design environment will contain screen handling operations, a work area to manipulate elements of the target system specification, and a menu which contains all the operations which the designer can perform on the highlighted elements in the work area.

Figure 2 illustrates the finally achieved environment. The work area is centered on the screen. The visual design operations, such as \text{select}, are located on top of the work area window, and the operations which may be executed in the next design step according to the position of the cursor are displayed on the right hand of the work area. The sample screen shows the initial state for a visual user interface specification based on the IMN (section 2). This technique...
is based on the stepwise refinement of end user tasks until appropriate system functions can be identified.

In the beginning the cursor is located at a particular task (highlighted in the figure) whose accomplishment should be supported. In order to achieve the decomposition of user tasks the operational part of the specification provides a set of design activities. Add SubTask provides for the refinement of tasks at the top level of abstractions, whereas Add InteractionMedium and Add ProblemDomainDate allows the designer to identify elements of the intermediate layer of the IMN, before he/she can add system function to handle problem domain data as well as interaction media.

5 Conclusions

In this paper we presented a novel approach to generate customized design environments from formal specifications. Using this approach, the design process can be individually adapted to user tasks by generating specific environments for the accurate representation of specific domains.

We summarized how an environment for an existing declarative task-oriented specification technique is formally specified. To further establish the feasibility of this approach, our effort is continued by specifying environments for a wide variety of specification languages. This, we hope, will assist in fine-tuning our approach to accommodate particular requirements that various projects might have.

References


The Argos Language: Graphical Representation of Automata and Description of Reactive Systems

F. Maraninchi

(Paper not received in time for press.)