• Is there a metaphor or visualisation which would facilitate the designer's expression of design 'quality' and interaction with trade-off information generated by the algorithm?

5. REFERENCES

[1] Gupta, A. and Rankin, P.J.. Knowledge assistants for design optimisation. *1st Artificial Intelligence in Design Conf.,* Edinburgh, June 25-27, 1991.
[2] Rankin, P.J.., Siemensma, J.M., Analogue circuit optimization in a graphical environment. *Proc. IEEE ICCAD Conf.,* Santa Clara, Nov. 1989, pp. 372-375.
[3] Apperley, M., Brouwer-Janse, M.D., Kasik, D. Rankin, P.J., & Spence, R. Practical Interfaces to complex worlds (Panel), *Proc CHI.Conf,* Seattle, 1990, pp. 257-260.
[4] Colgan, L., Brouwer-Janse, M.D., An analysis of the circuit design process for a complex engineering application. *Proc. Interact Conf.,* 1990, pp. 253-258.
[5] Colgan, L., Spence, R. Cognitive models of electronic design. *I st Artificial Intelligence in Design Conf.,* Edinburgh, June 25-27, 1991.

CONTACT INFORMATION

Lynne Colgan
Imperial College
Department of Electrical Engineering
Exhibition Road
London SW7 2BT U.K.
Phone: +44 (0)715814419
Fax: +44 (0)71823 8125
E-mail: lynne@uk.ac.ic.ee.titan

THE DRUID USER INTERFACI MANAGEMENT SYSTEM

Eelco Vriezekolk
Eindhoven University of Technology

Druid is a user interface management system (UIMS) for building complex, highly interactive graphical user interfaces (UIs). Druid is based on three fundamental principles.

* Druid is platform independent.

  Druid is not restricted to any specific window-system or graphical toolkit. This guarantees that UIs developed with Druid can be used on all present and future platforms (whether they are X11-based or not).

  Interfaces adopt the look and feel of the local window-system. We call this the "chameleon behavior" of Druid.

* Simple things must be done in a simple way.

  Druid allows for concise and direct interface descriptions, because the parts of the interface that do not have the current attention of the UI-designer need not be specified in detail. This means that even barely worked-out specifications can be processed.

  This feature does not only make the Druid language easy to use, but also helps rapid prototyping, or iterative design. This means that it is possible to test prototypes of a UI with users and to change the interface according to their comments.

* Druid will be used by non-programmers.

  Persons best equipped to make well-designed user-interfaces generally are not experienced programmers. They could, for example, be human factor specialists without a computing science background. The person constructing the actual application generally is a skilled programmer with no experience with the ergonomical aspects of UIs.

  Druid allows non-programmers to specify complex UIs, while the application programmer is creating the actual application. Even if these two roles are combined in the same person, which is still often the case, Druid's easy of use will be very useful.

TOOLS OF THE DRUID SYSTEM

Druid uses an object oriented language to specify UIs. The Druid-compiler translates such a UI-description into C-code that is still platform-independent. It is only during the linking phase that a platform specific library is used. This library is called the Druid Intermediate Toolkit (DIT).

The function of the DIT is to map calls between the generated C-code and the specific graphical toolkit. The DIT will generally be only a small layer. For each supported graphical toolkit such a DIT must be constructed.

To facilitate the writing of the Druid-code an interactive graphical editor is being created. We see the editor as an essential part of the Druid system, although it is not yet finished.

A POWERFUL UIMS

In Druid, UIs are built of objects, such as buttons and menus. With these objects, both the layout and the dialogue of a UI can be specified.

Layout is specified by setting the attributes of objects (such as position, color, or label) to their desired values. Most attributes can be changed run-time, and it is possible to define extra attributes.

Dialogue is specified by event handlers of objects. With event handlers, objects can react to external mouse and keyboard events, but it is also possible to add your own events for communication between objects. Event handlers can do assignments to attributes (adjusting the layout and giving user feedback), call application procedures, and send events to other objects. Event handlers can contain conditional statements.

A set of related objects can be grouped into a single compound object, such as a file selection panel. These compound objects can be reused between different UIs.

All objects can be created and destroyed run-time. In particular, dynamically creating compound objects can be very powerful.

The strength of event handlers is an important feature of the Druid system. Also, Druid UIs can be used on any hardware and software. Although the exact layout will differ from system to system (according to the chameleon behavior), the functionality will always be the same. The Druid system will be finished in September 1991.

CONTACT INFORMATION

E. Vriezekolk
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven
The Netherlands
telefax: +31 40 436685
email: uims@win.tue.nl

TASK PROTOCOL SPECIFICATION: A WORKSTATION INDEPENDENT SPECIFICATION TECHNIQUE FOR HUMAN-COMPUTER INTERACTION

Paul M. Mullins
Youngstown State University
Siegfried Treu
University of Pittsburgh

Methods of specification for human-computer interfaces (HCI) are diverse. Commonly used methods include (augmented) transition networks, BNF grammars, event-based descriptions, and the command language grammar [Moran81]. The latter is especially useful for its task-based description of the conceptual model as well as the dialogue. The Command Language Grammar (CLG) begins with a task analysis, and proceeds to model the HCI structurally through a process of step-wise refinement. The result is a detailed specification or representation of the system.

Although useful for the design, analysis, and specification of HCIs, the CLG has not gained the kind of widespread acceptance that might be expected for so versatile a tool. This is at least partially due to the level of detail required for the specification. The task domain must be fully analyzed and each task then described in increasing detail, down to a keystroke level. Once such a specification has been achieved, it is cumbersome, at best, to work with. Finally, the representation is specific to a particular platform, i.e. interaction devices, styles and techniques become an integral part of the specification.

The proposed specification technique, the Task Protocol Specification (TPS), is designed to avoid the problems identified for the CLG. Like the CLG, the TPS is intended to model the task domain and structure of the HCI, and to provide a representation of the HCI suitable for analysis and as an implementation guide. It also shares the capacity of the CLG to describe the conceptual model for the HCI. This technique provides the three most prevalent kinds of models used for HCIs: task, structure, and representation. However, the TPS avoids the detail provided by the CLG by designing for an Abstract WorkStation (AWS) [Mullins & Treu91, Mullins91] rather than a particular environment. This results in an intermediate-level specification which is more suitable for human consumption.

The extended-TPS has five levels which are similar in content and development to the six levels of the CLG. The initial specification, the *task* level (1) description, and the subsequent refinement to a *semantic* level (2) description are essentially the same in the two methods. Except the TPS results in a specification based on the AWS concept, hence providing a description for a family of HCIs, whereas the CLG provides a description for a particular HCI.

The lowest layer (4) of the TPS, the *interaction* level, deviates substantially from its counterpart in the CLG method. This TPS level is used to describe a basic set of system messages and interaction tasks, while the CLG is concerned with dialogue structure and physical actions of the user and the system. The TPS *interaction* level provides a generic description (actually a vehicle for description) of interactions intended to support various applications and user interfaces. By analogy to compiler technology, the CLG completely specifies the "task" in machine language, while the TPS specifies the system in terms of an intermediate language. The intermediate language describes an abstract machine. Design for this machine increases portability since it represents a family of HCIs on multiple platforms and promotes *conceptual consistency* among HCI instantiations.

The TPS *interaction* level description details the kinds of interactions that take place between the application, HCI, the host system(s), and the components of the HCI. The description includes any special functionality, the communication protocol, and the general form of messages used to interconnect the modules of the HCI. This part of the specification is referred to as the Task Interaction Protocol (TIP). The TIP provides a workstation-independent language representation of the tasks and describes a means of communication between the modules which form the HCI.

The difference in form and purpose for the TPS *interaction* level specification affects the specification of the *syntactic* level (3) which is otherwise a refinement of the *semantic* level, as in the CLG. The *AWS-physical* level (5), an extension of the TPS, describes an instantiation of a TPS specification on some platform. The *spatial* level of the CLG has no corresponding level in the TPS, since the type of information needed to determine spatial orientation and layout issues is only available at the higher levels.