Int. J. Man–Machine Studies (1989) **31**, 477–494

Toward empirically derived methodologies and tools for human–computer interface development

H. Rex Hartson and Deborah Hix
Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg VA 24061, USA

(Received 4 May 1988 and in revised form 4 January 1989)

1. Background: generations of UIMS

USER INTERFACE MANAGEMENT SYSTEMS—UIMS

This term, unknown only a few years ago, now conjures up images of icons and objects, windows and words that comprise the human–computer interface. A UIMS is an interactive system composed of high-level tools that support production and execution of human–computer interfaces. UIMS have become a major topic of both academic and trade journal articles, conference technical presentations, demonstrations, and special interest sessions. Many commercial software packages and research products even tangentially related to the area of human–computer interaction now claim to be UIMS. As young and exciting as the field is, there are already signs of promises unfulfilled, due to a lack of both functionality and usability—factors that can make the difference between whether UIMS are a passing fad or a viable tool. But what does the future hold for UIMS? All indications are that they are here to stay. We perceive a trend in UIMS evolution that we have divided into generations based primarily on common characteristics and only loosely on chronology:

First generation: façade and prototype builders, predecessors to real UIMS.

Second generation: UIMS with run time support, design time tools with limited functionality, increased emphasis on rapid prototyping, and often poor usability.

Third generation: UIMS with increased functionality and flexibility through such advances as object orientation, direct manipulation, asynchronous and complex graphical dialogue; still a usability gap.

Fourth generation: UIMS with improved usability for interface developers, allowing UIMS to be put more widely into practice; empirically-based improvements in both interface development methodologies and the UIMS themselves.

Future generations: UIMS supplemented by artificial intelligence, such as expert systems for aiding interface development and knowledge-based processing.

UIMS of the first two generations have established themselves in both research and commercial arenas. Although first generation UIMS were mostly façade generators and prototypers (e.g. IDS of Hanau & Lenorovitz, 1980; ACT/1 of Mason & Carey, 1981; FLAIR of Wong & Reid, 1982), several saw significant application in commercial development environments, participating in the development of real interactive software systems. Interfaces produced by these UIMS were

0020-7373/89/040477 + 18$03.00/0

typically specified by BNF-style languages, supplemented with conventional programming.

Early second generation UIMS research often focused on support for interface execution (e.g., Guedj, ten Hagen, Hopgood, Tucker & Duce, 1980; GIIT, 1983), with little emphasis on the end-user, human factors, or interface design. The second generation produced several experimental systems that contributed greatly to the knowledge and experience base (e.g. RAPID/USE of Wasserman, 1981; TIGER of Kasik, 1982; Menulay of Buxton, Lamb, Sherman & Smith, 1983; Abstract Interaction Handler of Kamran & Feldman 1983; Syngraph of Olsen, 1983; Dialogue Management System of Hartson, Hix & Ehrich 1984; GUIDE of Granor & Badler, 1986). Second generation UIMS have been commercially available for some time (e.g. BLOX of Rubel, 1982; Domain/Dialogue of Schulert, Rogers, & Hamilton, 1985; RAPID/USE of Wasserman & Shewmake, 1985). UIMS of this generation varied greatly in their capabilities and especially in their usability, with many still requiring significant conventional programming. Thus, first and second generation UIMS were mostly tools for programmers.

Moving toward the third generation, the UIMS view has broadened (e.g. Tanner & Buxton, 1984; Olsen, Buxton, Ehrich, Kasik, Rhyne & Sibert, 1984; Hartson & Hix, 1989) to include emphasis on the end-user and the non-programming interface developer. Third generation UIMS are adding new functionality to address specific problems. For example, limitations of sequential dialogue are overcome by event-based asynchronous dialogue (e.g. Green, 1985). Object orientation offers marked improvement in implementation efforts (e.g. Sibert, Hurley & Bleser 1988) because of its close match to the event-based perspective. Dialogue techniques can be specified by demonstration (e.g. Myers, 1987). New non-UIMS software tools and toolkits for specific interface features, such as window management (e.g. X Windows, described in Scheifler & Gettys, 1986), are sometimes used as the basis for UIMS development, speeding production of the UIMS itself and enhancing its power. While the third generation is producing interesting new UIMS ideas, it has not yet seen the needed empirical refinement to solve the problems, especially usability problems, of previous generations.

By these admittedly subjective definitions, the state-of-the-art in UIMS evolution is into the third generation, and we see a need to produce yet a fourth generation, a generation *characterized by greatly improved usability and functionality of the UIMS themselves*. Despite a scarcity of supporting empirical evidence, and despite their sometimes limited scope and difficulty of use, it is a generally accepted premise that UIMS have the potential to improve user interface quality and shorten development time. Much research has addressed end-users; now it is time to address interface developers—as users of UIMS. Those who have begun work in this direction include Mantei & Teorey (1988), with incorporation of human factors into the software life cycle; Carey (1988), concerned with matching tools such as UIMS to their users and environment; and Rosson, Maass & Kellogg (1987), who used interviews with designers to generate strategies for design and requirements for tools to support these strategies.

One point is important to note here: this paper is not about the human factors issue of how to design good human–computer interfaces. Rather its purpose is to provide a *framework* into which the human factored interface design process can be

placed, along with other activities in the development process. Specific decisions such as which evaluation techniques to use and when to use them (e.g. Grudin, Ehrlich & Shriner, 1987) are a part of what is placed into such a framework. Human factors is a focus for design of the interface, while human factorability is a focus for design of interface development tools and environments. To build a specific human factors approach into a general model for a development process and environment would be limiting and counter-productive.

The work we report here is an observational approach toward what Rosson, Maass & Kellogg (1987) term "designing for designers". This paper presents some problems of the early generation UIMS (Section 2), an empirical approach to begin solving these problems (Section 3), and qualitative results (Section 4) of the observations: a human–computer interface development life cycle and representation techniques, as well as capabilities of UIMS needed to support them. It concludes (Section 5) with a summary and future directions for the further evolution of UIMS.

2. Problems with early generation UIMS

A GAP TO BE BRIDGED

Co-operation between behavioral scientists and computer scientists is necessary for successful development of interactive computer systems with quality human–computer interfaces, as discussed in Hartson (1985). With the prevailing emphasis on human–computer interfaces, a new role—that of an interface specialist called an interface developer—and a new task—that of developing an effective, usable interface—have emerged. Support for this new role and new task has logically been sought in the software engineering realm. But the conventional approaches for software development are proving to be inadequate and ineffective for human–computer interface development. Thus, there is a critical gap caused by differences between methods used by behavioral scientists (interface developers) to design and analyse interfaces and the traditional methodologies and tools provided by computer scientists to design and implement software.

Mechanical aspects of representing interface designs (especially with interactive tools such as UIMS) often hamper performance of the design task itself; the syntactic domain interferes with the semantic domain, as discussed in Shneiderman & Mayer (1979). Particularly with the increased functionality—and corresponding complexity—of third generation UIMS, an interface developer can easily lose sight of the interface development task, getting lost in the UIMS itself. Thus, the gap is manifest in the lack of, first, a *holistic development methodology* for the complete system, and, second, usable *interactive tools* for supporting activities of this broader methodology.

First, consider the methodology problem. In simple terms, a methodology includes procedures for various stages of the development process and a notational technique for representing designs. Despite advances in human–computer interface development, no accepted methodology exists to guide development of the complex structure of an interface. The few interface development approaches that do exist lack connections to approaches to development of the computational (non-interface)

software of an application system, thus widening the gap. Wasserman's User Software Engineering (USE) methodology (Wasserman, 1981) and the Dialogue Management System (DMS) methodology (Yunten & Hartson, 1985) come closest to providing a connection to software engineering.

Second, consider the tool problem. Existing UIMS are generally limited in the types of interfaces they can produce, and are difficult to use. Early generation UIMS provide examples of how the few existing tools fail to support activities of the whole life cycle. As with the methodology, most UIMS (exceptions again include USE and DMS) do not make a connection to approaches for developing the computational software. Although current UIMS support some interface development activities (such as specification, prototyping, implementation, and execution), little exists to support the crucial very early creative stages of interface development. The representational techniques of many existing tools are indirect, using such notations as BNF, state transition diagrams, and high-level dialogue programming languages. Furthermore, few observational data exist of developers performing development activities (cf. Hammond, Jorgensen, MacLean, Barnard & Long, 1983; Lammers, 1986; Rosson, Maass & Kellogg, 1987). In addition, the term "UIMS" itself is ill-defined; interpretations of what a UIMS is range from very limited tools for screen layouts or run-time procedure calls, to systems that are almost CASE†-like in their scope. In this paper we address the concept of tools in a broader sense, including their support during all interface development activities, both at design-time and at run-time.

The limited existing methodologies and tools for interface development have typically evolved based on best guesses and intuition. Although behavioral scientists and computer scientists have begun reaching toward each other, a gap still exists between their worlds. A new generation of UIMS can bridge this gap by supporting all developer roles through an integrated holistic methodology involving a more extensive life cycle concept, and by making the tools a better fit to the interface developer's needs. The next section presents an empirical approach toward evolving such a methodology and tools.

3. Approach to solving the problems

EMPIRICAL OBSERVATIONS

In an effort to begin bridging this gap, we conducted several studies, not to formally test a hypothesis, but rather to qualitatively investigate natural—i.e. without *a priori* bias from specific approaches, notations, and tools—and effective ways in which quality interfaces are developed. These observations have provided input to the procedural (life cycle) aspects and the notational (representation) aspects of an evolving holistic methodology, as well as tool requirements to support this methodology. This approach follows the principles of system design recommended by Gould & Lewis (1985): focus on end-users (of UIMS) early in the design process, interactive participation by end-users in the design process, empirical studies throughout all phases of system evolution, and iterative refinement of the system based on the empirical results.

† Computer-Aided Software Engineering.

We conducted three protocol-gathering case studies involving interface developer subjects producing different kinds of interactive systems. In the most extensive of these, three subjects developed (including full implementation) a document retrieval system over a two year period. This system was chosen because it represented a broad variety of interface styles and techniques (e.g. menus, function keys, typed text input, direct manipulation), and included both sequential and multi-thread dialogue. Subjects had several years of industrial experience in software development environments. Subjects were given a requirements document containing a functional description of application system semantics, with very little information about the human–computer interface and no specific methodological instruction about interface design.

Although details varied, what follows describes typical observations. All three subjects used similar approaches, starting with detailed pencil-and-paper scenarios of what the end-user sees and does (e.g. viewing displays, pressing keys, entering commands, using a mouse). They initially represented the design with a set of numbered sketches of screen displays. Beside each item that corresponded to an end-user input (e.g. menu choice, function key definition, prompt for a keyword), they wrote the number of the corresponding successor screen sketch. To show global flow among screens graphically, they developed representations that varied from flow charts to state diagrams. These were detailed and concrete, yet large, complex, and initially without much structure. This pencil-and-paper "prototype" was tested manually, and changes immediately fed back to the interface design.

Next, subjects alternated from this bottom-up approach using scenarios and state diagrams to a top-down approach. To get state diagrams to fit on single sheets of paper, simple groupings were used to organize details into levels of abstraction. Subjects worked downward through the representation, analysing, structuring, and modifying the design, until they could implement a first version of the document retrieval system. After implementation, the developer subjects alternated back to a bottom-up step of end-user testing to refine the interface iteratively. Although some of the sequencing structure was modified, focus was primarily on interface details, and many design decisions (e.g. screen layout, menu format and content, graphics, consistency of wording) were made and/or changed. Developer subjects went through three complete interface revision cycles, plus many smaller iterations for fine-tuning of interface details. During each successive cycle, end-users became more satisfied with the quality and functionality of the interface.

In two other similar, but less extensive studies, subjects produced some form of scenarios and state diagrams as a starting point for interface development. Added support has been given to our observations by two years of co-operative research with IBM Federal Systems Division, as described in Hartson, Hix & Kraly (1989). We found that much of their interface development, at least informally, is scenario-driven. Often the customer provides initial scenarios; on other occasions, interface developers produce them in the early steps of the development process. In addition, Carey (1988) relates observations of numerous independent design teams; all produced pencil-and-paper versions of screen sequences early in the development process. Finally, the interface developers' life cycle and notational needs we observed are consistent with Piaget's theory (1950) that people naturally learn by starting with concrete examples and working toward the abstract.

4. Results of the observations

A LIFE CYCLE, REPRESENTATION TECHNIQUES, AND NEXT GENERATION UIMS FOR INTERFACE DEVELOPMENT

It is surely impossible to determine empirically (or otherwise) the "best" way to develop interfaces. But our observations have yielded indications of procedural (life cycle) and representational (notational) needs of interface developers, that in turn reflect tool (UIMS) requirements.

We conclude that a unified *holistic methodology* is needed that treats human–computer interface development as an integral part of the software engineering process. As shown in Fig. 1, it consists of (1) a *life cycle* for interface development and (2) a set of *representation techniques* for capturing outputs of each activity of this life cycle. A *new generation of UIMS* is also needed to support this methodology throughout all activities of the life cycle. Without this support, interfaces produced with UIMS will not necessarily be better than those produced without UIMS. In fact, early generation UIMS may serve only as a means for faster production of bad interfaces.

4.1. A LIFE CYCLE FOR INTERFACE DEVELOPEMENT

Bridging the gap in the interface development process necessitates a non-conventional life cycle. Two relevant results of our empirical observations are discussed below.

- We hypothesize, based on our observations, that human–computer interface development occurs in "alternating waves" of bottom-up (synthesizing) and top-down (analysing) activities.
- We are evolving a star configuration interface development life cycle that supports this hypothesis.

Evolution of our hypothesis. Draper & Norman (1985) draw a parallelism between human–computer interface development and software engineering. A prevailing view of software engineering is that it is based on a top-down, linear life cycle. However, some researchers (e.g. Swartout & Balzer, 1982; Ramamoorthy, Prakish, Tsai & Usuda, 1984) are beginning to see that this linear cycle is not the most appropriate for all software. Given this more flexible view of software engineering, the arguments of Draper & Norman for treating interface design analogously to software design are even more compelling. Our observations have led us to conclude that the life cycle for interface development does not "naturally" follow the

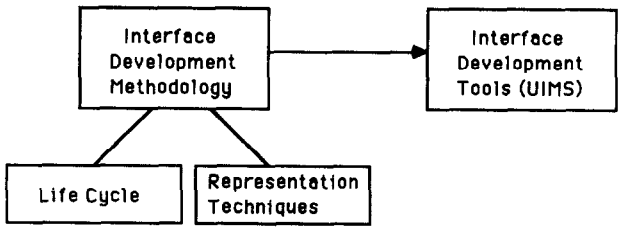

FIG. 1. Requirements for support of human–computer interface development.

traditional software development life cycle, with its top-down, linear sequence of somewhat isolated activities for requirements, design, implementation, and testing. In fact, attempts to impose the classical "waterfall' paradigm on interface development are undoubtedly the cause of many bad interfaces.

Based on (1) results of our initial studies, (2) our industrial liaison work, (3) other researchers, and (4) Piaget's classical theory, we have developed an alternative hypothesis about human–computer interface development. We hypothesize that the interface development life cycle most naturally occurs in "alternating waves" of two kinds of complementary activities. Typical early activities of interface development are bottom-up, based on concrete dialogue scenarios, often augmented with state-diagram-like representations of sequencing. Subsequent activities involve top-down, step-wise decomposition and structuring. Activities that are bottom-up, synthetic, empirical, and related to the end-user's view alternate with activities that are top-down, analytic, structuring, and related to a system view. These two kinds of development activities reflect different kinds of mental modes, which we call "analytic mode" and "synthetic mode", summarized in Fig. 2. The well-established iterative refinement approach is a step in the direction of these alternating modes, but in practice, most iterative refinement consists of going through the linear life cycle several times, each time requiring weeks or months. In contrast, an interface developer may alternate between these analytic and synthetic modes of mental activity several times within a single phase of development activities and within a short time period—hours or possibly even minutes.

While this concept of alternating waves is not new, its application within the context of software and interface development methodologies is new. In fact, the hypothesis of alternating mental modes runs counter to traditional top-down software engineering paradigms. However, it is supported by observations, practitioners, researchers, and psychological precepts, and says that humans do not naturally follow the accepted paradigm! (Our interviews with professional human–computer interface developers in fact revealed that where strict top-down methodologies were enforced, developers would often produce initial designs bottom-up but report them as having been developed top-down.)

Analytic mode	**Synthetic mode**
Top-down	Bottom-up
"Stop"	"Go"
Organizing	Free-thinking
Judicial	Creative
Structural	Behavioral
General	Specific
Abstract	Concrete
Modeling	Empirical
Formal	*Ad hoc*
Reflects system's view and work toward end-user	Reflects end-user's view and works toward system

FIG. 2. Summary of analytic mode and synthetic mode activities.

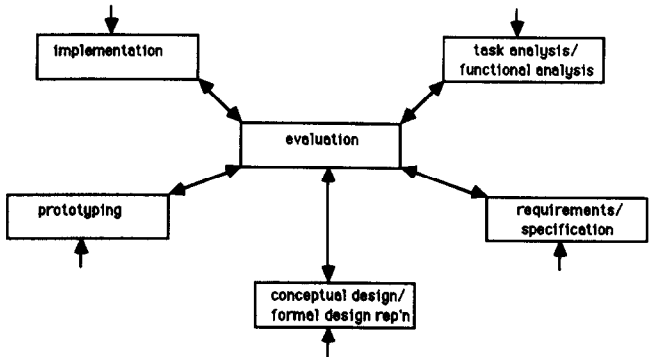

FIG. 3. A star life cycle for human–computer interface development.

As many practioners know intuitively, development of interfaces and software requires both mental modes. This emphasis of one or the other is dictated by experience, as shown in Rosson, Maass & Kellogg (1987). The top-down nature of analytic mode activity can predominate when the developer has some experience and *a priori* knowledge of target system structure. Emphasis on the bottom-up nature of synthetic mode activity is more suitable in novel design situations, where little about the target system structure is known in advance, and "trial and error" activity must augment experience and intuition to develop system structure. Interface development is newer and less well understood than software development. As interface development knowledge and experience are accumulated, top-down techniques can be used more successfully. An exciting topic for future work is a study of the nature of the contexts in which interface developers use each development mode.

A star life cycle for human–computer interface development. We have extrapolated our observations into a model of the human–computer interface development life cycle, shown in Fig. 3 as a star-shaped network configuration with evaluation at the center. Unlike the conventional linear "waterfall' life cycle, or its wheel-shaped variation with feedback for global iteration, this star life cycle of interface development activities helps bridge the interface development gap, and supports our alternating waves approach in at least three ways:

- it is highly interconnected,
- it allows almost any ordering of development activities, and
- it promotes rapid alternation among development activities.

In each pass through the center of the star—evaluation—the extent of evaluation can range from essentially none (e.g. for minor changes to requirements or design) to a large formative evaluation study (e.g. after a relatively complete prototype or implementation effort). This is the kind of life cycle we foresee being supported by the next generation of UIMS. Because of its totally interconnected nature, almost any place (except possibly implementation) is appropriate to enter this star life cycle. Thus, a traditionally "later" entry point into the cycle of Fig. 3 (e.g. a

conceptual design activity such as playing with scenarios) may be followed by a traditionally "earlier" activity (e.g. formulation of requirements) in the life cycle. This again opposes conventional linear, top-down paradigms, but supports our hypothesis of cyclic alternating waves of development.

The star life cycle offers to the interface developer the same freedom for directing the course of the development process that the new asynchronous interface styles offer to end-users for directing the interface. However, managers must recognize that this freedom requires different techniques for tracking development progress. In particular, they must assume increased responsibility for determining when sufficient evaluation and iteration have occurred.

Several seemingly disparate interface development approaches from both literature and practice can be put into perspective as parts of the life cycle in Fig. 3. For example, some developers start with user models and task analysis, others use dialogue scenarios, and yet other begin with the hierarchical structure of a command language. These approaches are just different entry points into the star life cycle of Fig. 3.

This star life cycle, while obviously very different from the traditional "waterfall" model, has some similarities with Boehm's (1988) "spiral" model for software development. There are, however, noteworthy differences between the star model and the spiral model. The spiral life cycle is for development of all software, with no particular emphasis on development of the human–computer interface, while the star life cycle is specifically for improving usability of the interface. The spiral life cycle includes testing and prototyping phases, but the evaluation-centered star life cycle revolves around continuous testing of the interface throughout the entire development process. The star life cycle easily allows small localized mini-cycles that are so important during interface development, while the spiral model features larger, relatively complete cycles.

Use of the star life cycle in a human–computer interface development project. To begin testing use of the star life cycle in a real world human–computer interface development project, we applied it during development of DMS, a UIMS we are producing within our Dialogue Management Project at Virginia Tech. The DMS development effort was sizable, at least for an academic research team, involving two full-time systems analyst/programmers, a designer, and an evaluation manager, plus two principal investigators about one-third-time, for fifteen months. Of particular interest were two major evaluation activities. In addition to numerous small evaluation cycles, we had one large formative evaluation about five months into the implementation process, and another, more summative evaluation about two months before delivery of the final product. The formative evaluation, described in Kahn, Hix, Notess & Hartson (1989) and performed by an expert evaluator from outside the Dialogue Management Project, yielded 57 "problems" with the DMS human–computer interface. For each of these problems, we developed possible revisions as well as a cost/benefit rating (cost in terms of programmer hours and benefit in terms of a severity rating for the problem). Approximately half the problems resulted in modifications to the interface; others simply could not meet the cost/benefit criteria and still be handled within the allotted time frame for modifications. The more summative evaluation, again using an expert evaluator from outside the project, resulted in identification of a smaller number of problems.

A few management problems were encountered by the principal investigator most closely in charge of DMS development. In particular, tracking revisions, especially as a result of the large formative evaluation, was somewhat time-consuming and difficult. Interestingly, the analysts/programmers and designer felt that the openness of the star life cycle was very conducive to creativity without sacrificing procedural control and structure. The human–computer interface of DMS was definitely improved as a result of use of the star life cycle. The final conclusion was that the star life cycle is very effective, leading to on- or even ahead-of-schedule deliveries of all components of DMS.

4.2. REPRESENTATION TECHNIQUES FOR INTERFACE DEVELOPMENT

An important part of bridging the gap in the human–computer interface development process is *communication*. The key to this communication is *representation techniques*—easy-to-use mechanisms that convey completely and unambiguously the specifications and designs of the evolving interface to all developer roles. Two relevant results of our empirical observations are discussed below:

- Based on our observations, we know something about the nature of the interface representation problem (*what* to represent).
- We are developing representation techniques to address those aspects of interfaces we know must be represented (*how* to represent).

Nature of the representation problem. At least three issues are involved in assessing the nature of the representation process. First, our observations showed that different interface developer roles need a common representation for communicating, but have different needs for representation in their individual development work. One solution is techniques that allow a common view for sharing, but to which *filters* can be applied to produce different working views for each role. Maintaining multiple views manually (e.g. pencil and paper) is difficult, but computer-based tools such as UIMS can maintain a single representation from which views are derived on demand.

The second issue relates to the need for matching the mental and physical processes of interface developers. In software engineering, the textual specifications, design documentation, and implementation code are all various representations of the target system. Producing such representations of the interface is a physical task, regardless of whether interactive tools or manual methods are used. A corresponding mental task to create the specification, design, or implementation necessarily precedes the physical task of representing it. This combination of mental and physical activity is illustrated by the box labeled "conceptual design/formal design representation" in Fig. 3. With current UIMS, the mental and physical tasks tend to be quite separate, causing discontinuity in an interface developer's work. Thus, another conclusion of our observations is that mental and physical interface development tasks must be brought closer together by providing effective interactive tools (i.e. UIMS) that closely match the mental processes they physically support.

The third issue is the magnitude of the unsolved problem of providing a complete, consistent, and readable representation of an interface design. Our observations made us acutely aware that the complete description of an interface requires massive amounts of complex details, including behavioral and structural, visible and

non-visible aspects. Developers must make and capture (using representation techniques) design decisions about end-user actions, system feedback, state information, interaction styles and techniques, visual attributes, lexical and pragmatic details, graphics, input validation and error handling, help information, and so forth. Representation techniques for use in human–computer interface development—whether manual or automated (e.g. UIMS)—must be able to capture this voluminous amount of detail.

A set of representation techniques. Based on our observations, we are developing a set of representation techniques to support our alternating waves approach. The techniques include *scenarios,* which support the bottom-up, behavioral, end-user-oriented mode of development; and *supervised flow diagrams,* which support the top-down, structuring, system-oriented mode of development and provide filters for different views of the design, as discussed in Hartson, Hix & Kraly (1989).

The term *scenario* is often used to refer to "storyboard" sketches of ideas for various screens of an interface. These incomplete sketches can be used informally to convey an approximation of the design to the implementers, who must fill in the details. Our use of the term scenario here is more precise, referring to a set of screen representations that begin as rough sketches but evolve into precise detailed representations of interface screens. Descriptions of end-user actions for selecting, moving, and otherwise manipulating interface objects (e.g. pop-up menus, icons, graphical application objects) are given by an easy-to-understand notation overlaid upon a visual representation of each screen. Descriptions of changes in the system representation of objects, in response to those inputs, are included as well. Screens are also labeled with rules to define all screen transitions based on end-user inputs. From this kind of scenario, a formal process can be used to derive corresponding state diagrams that augment the scenario by providing a direct representation of logical sequencing for end-user navigation among screens. In addition, various modalities within a screen (e.g. "add symbol" mode in a graphics editor) are represented by a set of concurrent state diagrams, each asynchronous with respect to the others. This kind of scenario is a prototype of the evolving interface, and offers a rather complete representation of the *behavioral domain of the end-user* of the evolving target system.

Scenarios and state diagrams are translated into *supervised flow diagrams,* the main representation technique used for developing and representing the *constructional domain of the developer* of the evolving target system. While state diagrams are useful for representing sequencing behavior, they suffer from several drawbacks as constructional representations: they lack a representation of data flow, they do not distinguish between dialogue and non-dialogue states, and they do not offer a discipline for controlling hierarchical abstraction. A supervised flow diagram is an extension of the state diagram concept, showing both control flow and data flow. Dialogue functions are distinguished from computational functions, and a model of human–computer interaction is used to constrain the way a design is organized into levels of abstraction. Because they are machine-readable, supervised flow diagrams can also provide filtered views of a design by displaying for the interface developer, for example, a design representation that emphasizes dialogue functions and hides details of computational functions. Supervised flow diagrams are directly interpretable for rapid prototyping. They can be translated into program-

ming language code for efficient execution of the target system.

Our representational techniques are variations of the formal concept of a state machine, giving our approach a firm theoretical basis. To help bridge the gap, we are also producing formal methods for mapping from one technique to another. For further elaboration and detailed examples of these techniques, see Hartson, Hix & Kraly (1989). Examples of other techniques oriented toward representation of asynchronous interfaces are being developed by Foley, Gibbs, Kim & Kovacevic (1988); Jacob (1986); and Siochi & Hartson (1989).

4.3. INTERACTIVE TOOLS TO SUPPORT HUMAN–COMPUTER INTERFACE DEVELOPMENT: THE NEXT GENERATION OF UIMS

In addition to a more natural life cycle and representation techniques, a key to bridging the human–computer interface development gap is interactive tools—UIMS—that support this life cycle and these representation techniques. Important characteristics of these tools include *functionality* and, especially, *usability*. Two relevant results of our empirical observations are discussed below:

- Based on our observations, interactive tools must support both bottom-up and top-down activities of human–computer interface development.
- To achieve high usability, interactive tools should be empirically derived, formally evaluated, and iteratively refined.

Support for alternating waves. It is evident from our observations that next generation UIMS must support a new life cycle structure and allow new kinds of representation techniques for all interface development activities. For example, UIMS to support the alternating waves approach to interface development must allow scenarios as a (bottom-up) starting point for development. Also, most existing tools tend to emphasize support of the traditionally "later" development activities such as design representation and prototyping. Much more tool support is needed for "earlier" activities, including task analysis, requirements, and conceptual design activities—that tight mental loop of synthesis and analysis involving dialogue scenarios, creative doodling, serendipity, and experimentation with ideas.

Based on our observations, we have determined some requirements for next generation UIMS for the alternating waves of each activity in the star life cycle configuration of Fig. 3:

- *Task analysis/functional analysis*—Task analysis, primarily a top-down activity, produces a hierarchical organization of end-user/system task activities. The top of the hierarchy is independent of interface design; the bottom part, however, typically has direct correspondence to the interface, containing interface-specific details of how the end-user performs each task. Task analysis feeds functional analysis, providing a hierarchical structure of computational functions to carry out the system's part of the task. Functional analysis constitutes the connection between task analysis and the later process of computational software development. An interface developer cannot complete task and functional analysis before beginning early interface design. As a result, UIMS support for task and

functional analysis must allow rapid alternation between task analysis and design activities. In fact, access to on-line documentation produced during task and functional analysis is needed in virtually all other development activities of the star life cycle, especially evaluation. We know of no UIMS that provides explicit support for these "early" development activities.

- *Requirements/specifications*—The requirements process is strongly both top-down and bottom-up, since both structure and details are important at this point in the development process. Although the conventional life cycle concept demands that requirements and specifications be complete before design begins, we observed a large amount of interaction between these activities. In most cases, structuring of requirements and specifications strongly influences structure of the subsequent design. We also observed the surfacing of design issues during the requirements process; for example, resolution of questions about feasibility of design facilitated some practical aspects of requirements decisions. The conventional life cycle espouses relatively independent performance of each activity; the alternating waves concept arises precisely because of the *interdependent*—but still separate—nature of the star life cycle activities. Just as for task and functional analysis, UIMS to support requirements and specification activities must provide on-line availability of requirements and specification documentation throughout all the star life cycle activities. For traceability and other project management needs, UIMS should help relate design decisions to the fulfillment of specific requirements. Also, an effective information storage and retrieval mechanism is needed to locate appropriate requirements for specific design situations.
- *Conceptual design/formal design representation*—As already discussed (see Section 4.2), a gap exists between the mental task of conceptual design and the physical task of representing that design. These activities are both top-down and bottom-up. Most formal design representation techniques are based on top-down, hierarchical structure, not allowing development of details without a pre-established structure into which they fit. This is a poor match to the bottom-up process of experimentation with details, followed by development of structure. Thus, UIMS must allow development of small detailed parts of the interface (e.g. a single disconnected menu) in isolation from the rest of the system, and allow these designs to be later connected into the interface structure. Emphasis in early generation UIMS has been on formal encoding of designs using various representation techniques (e.g. state transition diagrams, BNF grammars, high-level dialogue programming languages). In contrast, tools are need that allow, for example, "trial and error", blackboard-type "sketching", and that are as easy to use as doodling on the back of an envelope. An interface developer must be able to create interface objects and see them immediately, exactly as the end-user will see them. Object representations are manipulated by selecting, dragging, orienting, and changing attributes (e.g. shape, color, texture) by visual selection. With this type of support for the conceptual design process, formal representation can be automatic; the design is captured at the source with no encoding into a representation. That is, tools that match cognitive needs for conceptual design can all but eliminate the

problem of capturing a formal representation. To determine specific UIMS requirements for supporting conceptual design, much more observational work, of the type reported in this paper (in Section 2), is needed. "By-example" and other kinds of direct manipulation paradigms, for instance, may prove to be effective for conceptual design.

- *Prototyping*—Use of prototypes is primarily a bottom-up activity, in which the end-user sees and evaluates details of the interface. Rapid prototyping should allow the interface developer immediately to observe any part of interface form, content, and sequencing behavior already defined, without being hindered by incomplete and missing parts (a severe problem with most existing computer-based tools). Rapid prototyping is a key to support of evaluation and iterative refinement. The nature of iteration is such that it can occur in very short cycles (e.g. to resolve consistency problems over many screens of the interface). To support rapid alternation between prototyping and other development activities, especially design, UIMS need to minimize deep modality in the prototyping tool. This means that instead of closing down the design tools and bringing up the prototyper, the user should be able to view them as essentially two facets of the same tool. This view allows a smooth sequencing among design, execution (of prototype), halting, modification, and continuing with execution.
- *Implementation*—Output of the UIMS design tools is used during this activity to produce executable program code or interpretable descriptions. The interface development environment can be coupled with programming environments to produce the computational component of the target system.
- *Evaluation*—This is the very heart of the star life cycle, and necessitates different kinds of support tools within the UIMS at different times during development activities. For example, support needed to evaluate designs is different from that needed to evaluate the real application system after implementation. UIMS support for evaluation of application system design might include project management support, context-specific metering, traceability to relate the design to the requirements, and even some very early prototypes to test out various interaction styles. To support evaluation efforts after application system implementation, tools to capture log data files of end-user activities during interaction with the system are needed, as well as tools to expedite the collapse and analysis of these data.

In sum, UIMS must make visible an end-user-oriented view of interface behavior and then provide rapid modifiability to accommodate easy and natural evaluation and sometimes massive design changes throughout the life cycle. UIMS must allow the interface developer to see immediate results of changes and, to the extent possible, all effects on the rest of the system.

Evolution of next generation UIMS. To improve both UIMS functionality and usability, the interface to the UIMS itself must be given considerable attention, so that it provides the best possible support for an interface developer. This is achievable only through empirical evaluation and iterative refinement of existing UIMS, a much neglected aspect of UIMS development thus far. Approaches for empirically-based UIMS development are suggested in the following section.

5. Summary and future directions in UIMS research

In this paper, we have presented several generations of user interface management systems—UIMS—and discussed problems with the early generations. Most notable of these are limited functionality and poor usability, and a gap that exists between needs of the behavioral activities of the interface developer role and the methodology and tools provided for constructional activities by conventional software engineering techniques. We then related some empirical observations conducted to help us better understand the interface development process. Qualitative results of the observations include:

- a *star life cycle for human–computer interface development,* that facilitates an "alternating waves" approach of bottom-up and top-down development activities,
- a set of *reresentation techniques for human–computer interface development,* that represents both the end-user's behavioral view and the developer's constructional view of an application system, and
- *requirements for next generation* UIMS to support the life cycle and representation techniques.

Our work reported here is some of the first in which experimental studies and their qualitative results have been used to drive the development of methodology and tools for human–computer interface management. We see the need for long term, formal observations in "real world" software engineering environments, alternating with methodological and tool development driven by these observations, as shown in Fig. 4. Within our Dialogue Management Project we are continuing the initial work presented in this paper. This includes extensive protocol-gathering of interface devlopers at work on benchmark interface tasks, with special emphasis on "early" life cycle phases such as task analysis, requirements, specifications, and design representation. Analysis of these observations will increase our understanding of representation needs in these "early" activities and allow us to refine our corresponding methodological and tool support. For example, more work is required in the area of natural representation techniques for direct manipulation style interfaces, not just for details about use of a particular device or screen but for the overall end-user and interface behavior. It is our goal that the interface, and indeed the whole system, development process become a smooth, continuous,

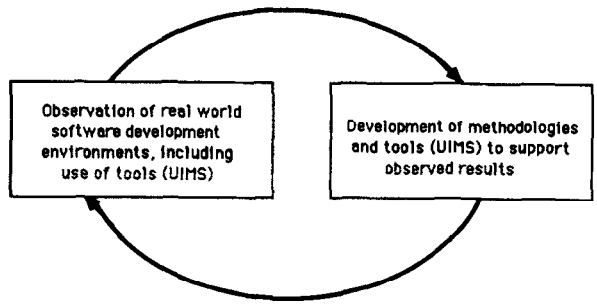

FIG. 4. Empirical evolution of interface development methodologies and UIMS: "Designing for designers".

tool-supported process from the beginning, using a natural means of representation that makes it easy to capture the design, easy to experiment with and evaluate it, easy to visualize interface behavior, and easy to change it.

Based on our observations, we conclude that such an empirical approach to "designing for designers" constitutes a profitable avenue for researchers in this area of human–computer interface management. For far too long, the need for life cycles and representation techniques tailored to human–computer interface development has been ignored, and tools have been produced based solely on their creators' intuition and best guesses. It is time to go to the source—the interface developers themselves—to gather empirical evidence about their needs for producing interfaces. Such activities will result in interface development techniques and tools that are empirically, rather than blindly, derived. This, in turn, will help bridge the gap among all interface developer roles, especially between behavioral scientists and computer scientists, bringing them together into the same interactive system development process. Building such bridges will take us to the next generation of UIMS and ultimately help us to build better interfaces.

We wish to acknowledge the participation in our Dialogue Management Project research by our colleague and friend, Dr Roger W. Ehrich, as well as research assistants Eric Smith, Antonio Siochi, Matt Humphrey, and Jeff Brandenburg. Mr Tom Kraly of IBM FSD contributed to the interface representation techniques research. This research is funded by the National Science Foundation, IBM Federal Systems Division, the Software Productivity Consortium, and the Virginia Center for Innovative Technology.

References

BOEHM, B. (1988). A spiral model of software development and enhancement. *IEEE Computer,* **21**(3), 61–72.

BUXTON, W. A., LAMB, M. R., SHERMAN, D. & SMITH, K. C. (1983). Towards a comprehensive user interface management system. *Computer Graphics,* **17**(3), 35–42.

CAREY, T. (1988). The gift of good design tools. In H. R. HARTSON & D. HIX, Eds. *Advances in Human–Computer Interaction,* Vol. 2. Ablex: Norwood, NJ. pp 159–174.

DRAPER, S. W. & NORMAN, D. A. (1985). Software engineering for user interfaces. *IEEE Transactions on Software Engineering,* **SE-11,** 252–258.

FOLEY, J. D., GIBBS, C., KIM, W. C. & KOVACEVIC, S. (1988). A knowledge-based user interface management system. In *Proceedings of CHI '88 Conference on Human Factors in Computing Systems, Washington, DC (May),* pp 67–72.

(GIIT) Graphical input interaction technique workshop summary. (1983). *Computer Graphics,* **17**(1), 5–30.

GOULD, J. D. & LEWIS, C. (1985). Designing for usability: key principles and what designers think. *Communications of the ACM,* **28**(3), 300–311.

GRANOR, T. E. & BADLER, N. I. (1986). GUIDE: Graphical user interface development environment. In *Proceedings of Trends and Applications.* pp. 37–41. Maryland: Silver Spring.

GREEN, M. (1985). The University of Alberta user interface management system. *Computer Graphics* **19**(3), 205–213.

GRUDIN, J., EHRLICH, S. & SHRINER, R. (1987). Positioning human factors in the user interface development chain. *In Proceedings of CHI + GI 1987 Conference. Toronto (April),* 125–131.

GUEDJ, R. A., TEN HAGEN, P. J. W., HOPGOOD, F. R. A., TUCKER, H. A. & DUCE, D. A., Eds. (1980). *Methodology of Interaction: Seillac II. Seillac, France, 1979.* Amsterdam, North-Holland.

HAMMOND, N., JORGENSEN, A., MACLEAN, A., BARNARD, P. & LONG, J. (1983). Design practice and interface usability: evidence from interviews with designers. In *Proceedings of CHI '83 Conference on Human Factors in Computing Systems, Boston (December)*, pp. 40–44.

HANAU, P. R. & LENOROVITZ, D. R. (1980). Prototyping and simulation tools for user/computer dialogue design. *Computer Graphics*, **14**(3), 271–278.

HARTSON, H. R., Ed. (1985). *Advances in Human–Computer Interaction*, Vol. 1. Ablex: Norwood, NJ.

HARTSON, H. R. & HIX, D. (1989). Human–Computer interface development: concepts and systems for its management. *ACM Computing Surveys* **21**(1), 5–92.

HARTSON, H. R., (JOHNSON) HIX, D. & EHRICH, R. W. (1984). A human–computer dialogue management system. In *Proceedings of INTERACT '84, First IFIP Conference on Human–Computer Interaction. London (September)*, pp. 57–61.

HARTSON, H. R., HIX, D. & KRALY, T. M. (1989). Developing human–computer interface models and recording techniques. To appear in *Journal of Software—Practice and Experience.*

JACOB, R. J. K. (1986). A specification language for direct manipulation interfaces, *ACM Transactions on Graphics*, **5**(4), 283–317.

KAHN, M., HIX, D., NOTESS, M. & HARTSON, H. (1989). Use of an outside expert evaluator in formative evaluation. Submitted to *Conference on Engineering for Human–Computer Interaction, Napa, CA (April)*, 20 pp. ms.

KAMRAN, A. & FELDMAN, M. B. (1983). Graphics programming independent of interaction techniques and styles. *Computer Graphics*, **17**(1), 58–66.

KASIK, D. J. (1982). A user interface management system. *Computer Graphics*, **16**(3), 99–106.

LAMMERS, S. (1986). *Programmers at Work.* Microsoft Press: U.S.A.

MANTEI, M. & TEOREY, T. (1988). Cost/benefit for incorporating human factors in the software lifecycle. *Communications of the ACM*, **31**(4), 428–439.

MASON, R. E. A. & CAREY, T. T. (1981). Productivity experiences with a scenario tool. In *Proceedings of the IEEE COMPCON. Washington, D.C. (September)*, pp. 106–111.

MYERS, B. (1987). Creating dynamic interaction techniques by demonstration. In *Proceedings of CHI + GI '87 Conference. Toronto (April)*, pp. 271–277.

OLSEN, D. R., Jr. (1983). Automatic generation of interactive systems. *Computer Graphics*, **17**(1), 53–57.

OLSEN, D. R., Jr., BUXTON, W., EHRICH, R. W., KASIK, D. J., RHYNE, J. R. & SIBERT, J. (1984). A context for user interface management. *IEEE Computer* (December), pp. 33–42.

PIAGET, J. (1950). *The Pschology of Intelligence.* Orlando: Harcourt Brace Jovanovich.

RAMAMOORTHY, C. V., PRAKISH, A., TSAI, W. T. & USUDA, Y. (1984). Software engineering: problems and perspectives. *IEEE Computer*, **17**(10), 191–209.

ROSSON, M. B., MAASS, S. & KELLOGG, W. A. (1987). Designing for designers: an analysis of design practice in the real world. In *Proceedings of CHI + GI '87 Conference. Toronto (April)*, pp. 137–142.

RUBEL, A. (1982). Graphic based applications—tools to fill the software gap. *Digital Design*, (July).

SCHULERT, A. J., ROGERS, G. T. & HAMILTON, J. A. (1985). ADM—a dialogue manager. In *Proceedings of CHI '85 Conference on Human Factors in Computing Systems. San Francisco (April)*, pp. 177–183.

SCHEIFLER, R. W. & GETTYS, J. (1986). The X Windows System. *ACM Transactions on Graphics*, **5**(3), 79–109.

SHNEIDERMAN, B. & MAYER, R. (1979). Syntactic/semantic interactions in programming behavior: a model and experimental results. *International Journal of Computer and Information Sciences*, **8**(3), 219–239.

SIBERT, J. L., HURLEY, W. D. & BLESER, T. W. (1988). Design and implementation of an object-oriented user interface management system. In H. R. HARTSON & D. HIX, Eds. *Advances in Human–Computer Interaction*, Vol. 2. Ablex: Norwood, NJ. pp. 175–213.

SIOCHI, A. & HARTSON, H. R. (1989). Task-oriented representation of asynchronous interfaces. In *Proceedings of CHI '89 Conference on Human Factors in Computing Systems, Austin, TX (May)*, pp. 183–188.

SWARTOUT, W. & BALZER, R. (1982). On the inevitable intertwining of specification and implementation. *Communications of the ACM*, **25**(7), 438–445.

TANNER, P. P. & BUXTON, W. A. (1984). Some issues in future user interface management system (UIMS) development. In *Seeheim Workshop of User Interface Management Systems*. New York: Eurographics-Springer.

WASSERMAN, A. I. (1981). User software engineering and the design of interactive systems. In *Proceedings of the Fifth International Conference on Software Engineering*. pp. 387–393.

WASSERMAN, A. I. & SHEWMAKE, D. T. (1985). The role of prototypes in the user software engineering methodology. In H. REX HARTSON, Ed. *Advances in Human–Computer Interaction*, Vol. 1. Ablex: Norwood, NJ. pp. 191–210.

WONG, P. C. S. & REID, E. R. (1982). FLAIR—user interface dialog design tool. *Computer Graphics*, **16**(3), 87–98.

YUNTEN, T. & HARTSON, H. R. (1985). A SUPERvisory methodology and notation (SUPERMAN) for human–computer system development. In H. REX HARTSON, Ed. *Advances in Human–Computer Interaction*, Vol. 1. Ablex: Norwood, NJ. pp. 243–281.