Temporal Aspects of Tasks in the User Action Notation

H. Rex Hartson  
*Virginia Polytechnic Institute and State University*

Philip D. Gray  
*Glasgow University*

**ABSTRACT**

The need for communication among a multiplicity of cooperating roles in user interface development translates into the need for a common set of interface design representation techniques. The important difference between design of the interaction part of the interface and design of the interface software calls for representation techniques with a behavioral view—a view that focuses on user interaction rather than on the software. The User Action Notation (UAN) is a user- and task-oriented notation that describes physical (and other) behavior of the user and interface as they perform a task together. The primary abstraction of the UAN is a *user task*.

The work reported here addresses the need to identify temporal relationships within user task descriptions and to express explicitly and precisely how designers view temporal relationships among those tasks. Drawing on simple temporal concepts such as events in time and preceding and overlapping of time intervals, we identify basic temporal relationships among tasks: sequence, waiting, repeated disjunction, order independence, interruptibility, one-way interleavability, mutual interleavability, and concurrency. The UAN temporal relations, through the notion of modal logic, offer an explicit

Authors' present addresses: H. Rex Hartson, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061; Philip D. Gray, Department of Computing Science, 17 Lilybank Gardens, Glasgow University, Glasgow G12 8QQ, Scotland.
CONTENTS

1. INTRODUCTION
2. THE NEED FOR BEHAVIORAL REPRESENTATION
3. RELATED WORK
   3.1. Constructional Representation Techniques
   3.2. Behavioral Representation Techniques
   3.3. Temporal Aspects
   3.4. Contributions of This Work
4. INTRODUCTION TO THE UAN
5. THE NEED FOR TEMPORAL RELATIONS
6. TIME
   6.1. Events in Time
   6.2. Time Intervals
   6.3. Preceding and Overlapping
7. TASKS AND ACTIONS
   7.1. Basic Definitions
   7.2. Instances of Tasks
8. TIME AND ACTIONS
   8.1. Actions as Happenings in Time
   8.2. Lifetimes
   8.3. The Boustrophedon Argument
   8.4. Interruption
   8.5. Idle Time
   8.6. Periods of Activity
9. TEMPORAL RELATIONS AMONG USER ACTIONS
   9.1. Sequence
   Task Names and Levels of Abstraction
   Grouping, Closure, and Composition of Relations
   9.2. Waiting
   9.3. Repeating Disjunction
   9.4. Order Independence
   9.5. Interruptibility
   Uninterruptible Tasks and Preemptive States
   Scope of Interruptibility
   9.6. One-Way Interleavability
   9.7. Mutual Interleavability
   9.8. Concurrency
10. DISCUSSION
   10.1. How the UAN Helps With Interface Development
   10.2. Conclusions
APPENDIX. MATHEMATICAL SYMBOLOGY

and precise representation of the specific kinds of temporal behavior that can occur in asynchronous user interaction without the need to detail all cases that might result.
TEMPORAL ASPECTS OF TASKS IN THE UAN

Time is nature's way of keeping everything from happening all at once.

—Unknown

1. INTRODUCTION

The great difficulty many people have in using computers is often due to a poor design of the human-computer interface. The issue is usability, and high usability stems from a good design. Good designs invariably depend on an ability to understand and evaluate (and thereby improve) interface designs during the development process. Understanding and evaluating designs depends, in part, on the methods used to represent the designs. Design and representation are very closely related; design is a creative, mental, problem-solving process, whereas representation is the physical process of capturing or recording the design. The need for effective representation techniques is especially important with new interface development methods that emphasize iterative refinement and involve a multiplicity of separate but cooperating roles for producing the interface. These roles include at least designer, implementer, evaluator, documenter, marketing, customer, and user. Each of these roles has its own, often different, needs for communicating—recording, conveying, reading, and understanding—an interface design. This communication need translates into the need for a common set of interface design representation techniques—the mechanism for completely and unambiguously capturing an interface design as it evolves through all phases of the life cycle. Development of a user-centered interface design necessitates that these techniques have a behavioral view—a view that focuses on the user rather than on the software.

The UAN has provided an answer to the need for a behavioral representation technique (Hartson, Siochi, & Hix, 1990; Siochi & Hartson, 1989). The UAN is a user- and task-oriented notation that describes physical (and other) behavior of the user and interface as they perform a task together. The primary abstraction of the UAN is a user task. An interface is represented as a quasi-hierarchical structure of asynchronous tasks, the sequencing within each task being independent of that in the others. User actions, corresponding interface feedback, and state change information are represented at the lowest level. Levels of abstraction hide these details and build the task structure.

The UAN has been found by many to be expressive and highly readable because of its simplicity, natural enough so that it is easily read and written with almost no training. Use within design and implementation projects has shown the UAN to be effective in conveying large and complex user interface designs from designers to implementers and evaluators. Because the UAN is task oriented, it provides a crucial articulation between task analysis and design.
In addition to the need for a behavioral view, new styles of interaction involving direct manipulation of graphical objects and icons necessitate temporal considerations. These interaction styles are more difficult to represent than the older styles of command languages and menus. User actions are asynchronous, having more complex temporal behavior than those of the old style interfaces that were largely constrained to predefined sequences. Heretofore, representation of temporal relationships has been ad hoc in the UAN. The work reported here addresses the need to identify more formally temporal relationships within task descriptions and to express explicitly and precisely how designers view temporal relationships among tasks.

We are not proposing a general theory of time or of tasks; we are merely applying some intuitive physical notions about time and the temporal properties and relationships of user tasks and computer processes. This is not a cognitive model of time (even though it refers to user's behavior), and none of our inferences or conclusions depends on features that are inconsistent with other views of time.

Out of the many logically possible temporal relationships among tasks that people carry out using computers, we have identified several that we believe to be fundamental to describing interaction (e.g., sequencing, interrupting, and interleaving). These notions are not new to designers of interactive systems, but their use has often been informal and imprecise. It is our hope that clear and precise definitions of these concepts will provide a foundation on which to reason about the temporal characteristics of interaction between users and computer systems.

A word is in order here about the structure of this article. We have used a top-down approach, often mandated by formal documentation of technical concepts. This approach has the advantage that it weaves the whole article into a connected development of concepts and definitions, defining each term before it is used. The disadvantage, however, is that this approach necessarily defers the "real content" until the end. The first five sections present introduction and motivation. The temporal relations themselves are discussed in Section 9, with Sections 6, 7, and 8 building a foundation of definitional fabric on which Section 9 is laid.

Those desiring a full logical development should read this article in the order in which it appears. A reader interested only in gaining intuitive knowledge of the temporal relations can skip Sections 6, 7, and 8; can begin with Section 9; or can read only Section 9. This reader, however, must expect to encounter many terms lacking a formal definition. As a further guide for the reader, equations in the article are numbered and set off from the text. Readers not wishing to sort out the meanings of the symbols and equations can skip all equations and still understand most of the ideas. The article is
written in a style to support this mode of reading; each equation is preceded by a prose description of what is stated in the equation.

2. THE NEED FOR BEHAVIORAL REPRESENTATION

Historically, and practically, many user interfaces have been designed by software engineers and programmers as part of the software of an interactive system. The result has been interfaces of varying quality and usability. Much work in the field of human–computer interaction has been directed toward new approaches to user interface development in hopes of improving quality and usability. From this work, it has become clear that there is an important difference between design of the interaction part of an interface and design of user interface software and that interaction design has special requirements not shared by software design. Good interaction design must be user centered. Being user centered means focusing on the behavior of a user performing tasks with the computer. To emphasize this distinction, we use the terms behavioral domain and constructional domain to refer, respectively, to the working worlds of the people who design and develop the interaction part of user interfaces and the people who design and develop the software to implement those interfaces (Hartson et al., 1990).

Most representation techniques currently used for interface software development (e.g., state transition diagrams, event-based mechanisms, window managers, software toolkits, object-oriented programming) are constructional—and properly so. Any description that can be thought of as being performed by the system is constructional. For example, a state transition diagram represents the system view, looking out at the user and waiting for an input. This diagram shows the current system state and how each input takes the system to a new state. Constructional representation techniques support the designer and implementer of the interface software but do not support design of the interaction part of the interface itself. In contrast, it is in the behavioral domain—from the user’s view—that developers of the interaction part of an interface (e.g., interaction designers and evaluators) do their work. A description performed by the user (e.g., performance of a task) is behavioral. In the behavioral domain, one gets away from the software issues and into the processes that precede software design, such as task analysis, functional analysis, task allocation, and user modeling. Consequently, there is a need for behavioral representation techniques (and supporting tools) to give a user-centered focus to interface development and to serve interface developer roles. As Richards, Boies, and Gould (1986, p. 216) stated about tools for mocking up user interface prototypes, “Few of these provide an interface specification language directly usable by behavioral specialists.”

With current emphasis on user-centered design (Norman & Draper, 1986),
the interface development process is driven heavily by user requirements and task analysis. Early evaluation of designs is based on user- and task-oriented models (e.g., see Reisner, 1981). In fact, the entire interface development life cycle is becoming centered around evaluation of users performing tasks (Hartson & Hix, 1989; Nickerson & Pew, 1990). Thus, most interface development activity that precedes constructional design and implementation is done in the behavioral domain leading to the user task as the common element among developer roles. Behavioral representation techniques are not replacements for constructional techniques; they just support a different domain. Interaction designs represented behaviorally must still be translated into the constructional domain for interface software design and implementation. Interaction designs become requirements for the design (and implementation) of user interface software. A formal representation of interaction designs is, therefore, needed to convey these requirements, and the UAN is intended for that purpose.

3. RELATED WORK

3.1. Constructional Representation Techniques

In comparing the UAN with earlier techniques, we begin with state transition diagrams (STDs). Because STDs are a constructional representation technique, we are comparing apples and oranges, but STDs are a common basis for interface representation, and, in the absence of good oranges, designers have been known to try apples. Although STDs can be used to supplement UAN task descriptions to represent certain aspects of task transitions and interface state (Hartson et al., 1990), they cannot represent interface feedback or appearance, and the power of standard STDs to represent relationships among tasks is limited to single-stream sequential control flow. STDs theoretically can represent the other temporal relationships by representing explicitly all possible sequential control flow paths, but the result is unusable—overwhelmingly large and complex, obscuring the very sequencing structure that transition diagrams are good at showing. Two independent, asynchronous tasks must be cast together as a single entity in a synchronous model. To represent asynchronism and interleavability of two tasks, in addition to the regular state transitions within a task, each state of one is a next state of every state in the other and vice versa. For example, consider this small generic example. Suppose task A has subtasks B and C that are temporally interleavable (the user can move back and forth, suspending each one while working on the other). In the UAN this is represented as:

\[
\text{Task A} \\
B \leftrightarrow C
\]
where $\Leftrightarrow$ means "is interleavable with," a temporal relation explained in Section 9.7. For simplicity, let tasks B and C be composed of sequential steps, as shown here in the UAN:

**Task B**
- D
- E
- F

**Task C**
- G
- H
- I

The STD for this simplest possible example of interleaving, shown in Figure 1, would contain complicated transition conditions that depend on the real current state in each separate sequence. To include that real state information in the STD of Figure 1, one would have to replicate every possible subsequence of task B in conjunction with every subsequence of task C, producing a combinatorial explosion of states and transitions. Just as one example, if task B is really at subtask E and task C is at I, then transitions are not legal from E to G or H nor from I to D. For real tasks, such as the use of spreadsheets and text editors, the result is overwhelmingly large and complex.

It is equally important to note here that the original intention was to represent the asynchronous relationship between tasks B and C. The sequencing of B is independent of the sequencing of C, but the diagram obscures that relationship by interconnecting them.

It is clear that possible transitions between subtasks of B and subtasks of C in the just-cited example must be represented implicitly, an approach taken, for example, by Jacob (1986) and by Wellner (1989). Because direct manipulation interfaces are composed of many individual simple dialogues that interact like coroutines, Jacob divided an interface into interaction
objects, each with a separate specification based on an STD. Coroutine calls among STDs give the necessary asynchronism, suspending execution of the calling STD and remembering its current state (part of the real state discussed before). One interaction object is active at a time, and one state is current within each object.

Wellner's similar approach describes, as an example, copy-machine controls with two buttons, one for toggling through choices of paper trays and one for toggling through exposure settings. Both Jacob's and Wellner's approaches separate all states relating to paper trays from those relating to exposure. In each case, the asynchronism between the two sequences is implicit in the rules for STD operation.

It is also useful to compare the kind of concurrency that can be represented by state charts and by the UAN. The two operations in Wellner's example, selecting a paper tray and setting exposure, are represented in state charts as being concurrently available to the user. This is really interleaving of those operations and is what Lorin (1972) called "apparent concurrency" as contrasted with "real concurrency." In the UAN, interleaving is explicitly distinguished from real concurrency, which involves the ability of the user to do both operations simultaneously, something not addressed by state charts. To the designer the difference may be just an implementational detail, but to the user it is significant.

Most representation techniques used with user interface management systems are constructional, including STDs, asynchronous STDs, and state charts. There are also event handlers (Green, 1985; Hill, 1987), which describe system actions (e.g., invocation of a computational procedure) in response to events resulting from user actions. Event handlers introduce an object-oriented flavor and, therefore, are even better suited for representing asynchronism. They have more expressive power than STDs (Green, 1986) but suffer in comparison, as do most object-oriented approaches, when there is a need to visualize or trace sequences of user operations.

3.2. Behavioral Representation Techniques

All representation techniques in the previous section are constructional. The UAN is task oriented and behavioral, so it does not compete with STDs, for example. Both kinds of techniques are necessary for interface development, but behavioral methods are needed specifically for interaction design.

Grammatical representations using Backus Naur Form (e.g., Syngraph; Olsen & Dempsey, 1983) tend to be behavioral because they describe expressions that come from the user, but they are difficult to write and understand. Also, like standard STDs, grammars typically do not represent interface feedback, do not represent the appearance of the interface, and are not suitable for asynchronism. Multiparty grammars (Shneiderman, 1982), an interesting extension to production-rule-based techniques, do support
direct association of interface feedback with user inputs. Multiparty grammars, however, are not easily adapted to the variety of user actions in direct manipulation interfaces.

One behavioral technique that has long been used both formally and intuitively involves scenarios (or storyboarding) of interface designs. This technique is effective for revealing an early picture of interface appearance and behavior. But, because a scenario is an example of the interface, it cannot represent the complete description of the user's behavior while interacting with the computer. Peridot (Myers, 1987) is based on specification of interfaces by demonstration—The user carries out the actions of the scenarios. The use of inference and confirming dialogue solves the problem of generalizing a design from specific instances of interaction. This approach is novel, but Peridot produces program code directly with no intermediate representation that can convey interface designs or behavior or that can be analyzed.

Most other behavioral techniques are generally task oriented, including the GOMS model (Card, Moran, & Newell, 1983); the Command Language Grammar (CLG; Moran, 1981); the Keystroke-Level Model (Card & Moran, 1980); the Task Action Grammar (Payne & Green, 1986); and the work by Reisner (1981), Kieras and Polson (1985), and Sharratt (1990). Design of interactive systems, as with most kinds of design, involves an alternation of analysis and synthesis activities (Hartson & Hix, 1989). Most of the models just mentioned were originally oriented toward analysis; that is, they were intended to represent an existing design in order to evaluate usability by predicting user performance, rather than to capture a design as it is being developed. On the other hand, synthesis includes activities that support the processes of creating a new interface design and capturing its representation. The UAN shares the task orientation of these other behavioral models but is more synthesis oriented, because it was created specifically to communicate interface designs to software engineers and implementers. In practice, most techniques mentioned before can be used to support synthesis as well but typically do not represent the direct association of feedback and state with user actions. Also, many of these models—the GOMS, CLG, and keystroke in particular—are models of expert error-free task performance in contiguous time (without interruption, interleaving of tasks, and without considering the interrelationships of concurrent tasks), not suitable assumptions for the synthesis-oriented aspects of concurrent tasks.

3.3. Temporal Aspects

The phenomena with which we are concerned in this article—user actions during interaction with computer systems—are similar to computer-based processes and to human cognitive behavior in that they all exhibit
sequentiality through time. That is, we can measure the amount of time taken for their execution, perhaps identify beginning and endpoints for their duration, and describe temporal relations among them. It should not be surprising, therefore, to find formalisms similar to our own for describing and reasoning about the temporal aspects of such processes and behavior.

Temporal logics have been developed for a number of applications in computer science, cognitive science, and artificial intelligence, among which are:

- reasoning about concurrent systems, including program verification (Barringer, 1985), operating systems, and very large scale integration design (Moszkowski, 1986);
- reasoning about database updates (Kowalski & Sergot, 1986);
- systems for temporal logic programming (Hale, 1987);
- building theories and automated systems to model human planning behavior (Allen, 1983, 1984; McDermott, 1982); and
- natural language understanding systems (Kahn & Gorry, 1977).

Two basic approaches to handling time are employed in these applications. One approach, employed largely for natural language understanding and temporal logic programming, uses a tense logic with modal operators that express the temporal dependencies of the truth values of propositions. However, where the goal is to describe the temporal attributes of events and processes, the truth value of propositions need not be treated as time dependent. This second approach, which we adopt in this article, models time as entities or attributes of entities that are then described using first-order predicate calculus.

Of these applications, the work closest to ours is that of Allen, which is concerned with describing and automating reasoning about human planning and conversation. Using first-order logic, Allen identifies 13 basic temporal relations among events and processes, such as "before," "during," and "overlaps," among which are the relations with which we are concerned. The main difference between Allen's theory and our own lies in his adoption of intervals of time, rather than time points, as primitive, with a consequent effect on the handling of the interruptibility of actions; this is discussed further in Section 6.2. As Allen has noted, however, it is possible to recast his theory with time points as primitives.

Constructional interface design models have not used temporal logic up to now, but that is not to say that they have failed to capture temporal aspects of interface behavior. Transition networks are capable of modeling sequential temporal ordering but are not capable of representing the temporal relation-
ships within asynchronous and concurrent interaction (Green, 1986). Production systems, based on sets of event–response rules, have been proposed as a means of capturing the more complex temporal relations among events in modern interactive systems (Duce, 1985; Hill & Hermann, 1989). However, the complex temporal relations that they are capable of capturing are hidden in the implicit semantics of rule selection, and hence these relations are neither explicitly expressed nor capable of being reasoned about.

Cardelli and Pike's (1985) Squeak, based on cooperating sequential processes, is a language for describing interfaces that exhibit concurrency. Thus, an interface is described in terms of a set of processes, each of which accepts events as input and generates events as output. Processes communicate with one another by the transmission of an output event from one process serving as the input to another. Unlike other constructional interface models, a formal semantics for Squeak has been defined in which there is explicit reference to the passage of time, which is used to express control flow among processes in terms of null actions during which time units elapse. However, no attempt is made, as with Allen and others, to model the temporal aspects of the system based on a theory of time and temporal relations.

The models of interaction described earlier are all constructional, in the sense that they represent interaction from the system's viewpoint. Mechanisms for handling control and communication from a programming point of view are not likely to capture all the temporal relations that exist among actions from a user's point of view. For example, the difference between interleaved processes and concurrent processes may be an implementational detail for a constructional description of the interface and hence, as in Squeak, does not appear in the abstractions of the language. As mentioned in Section 3.1, however, interleaved and concurrent actions are significantly different from a user's point of view. A behavioral description of the interaction must be able to express the difference and should be built on a theory of temporal relations among user actions that explicates the difference.

It should be emphasized that we are concerned here with the temporal aspects of user activity, not with the user's perception of temporal relations among these actions. Thus, recent work on the influence of the perception of time and the efficacy of human reasoning about temporal relations among processes (Decortis & De Keyser, 1988) is not relevant to our concerns.

### 3.4. Contributions of This Work

The need for synthesis-oriented behavioral techniques for interaction design representation was motivated in Section 2. In addition, designers need a precise framework in which to think about, discuss, and represent constraints on relative timing among asynchronous tasks. More motivation for
temporal relations is presented in Section 5. Sections 3.1 through 3.3 show that nothing already exists to fill these needs.

The UAN, as described in this article, does meet the need for a synthesis-oriented behavioral representation technique with temporal relations. The UAN is the only representation technique that provides synthesis-oriented, behavioral representation of tasks in interface designs, independent of implementation concerns, and with the temporal relations necessary to represent today's asynchronous interface designs. Further, design representation is not about actions a user makes so much as it is about actions a user can make. In an asynchronous environment, it is especially important to be able to represent specific kinds of behavior that can occur without having to detail all the cases that might result. The UAN temporal relations, through the notion of modal logic, offer an explicit and precise representation of what tasks can be interrupted, interleaved, and performed concurrently.

4. INTRODUCTION TO THE UAN

Use of the basic UAN, without emphasis on temporal aspects, is introduced briefly here by way of example. Figure 2, adapted from (Hartson et al., 1990), summarizes many of the UAN symbols, with only the temporal relations needed for the examples in this section. These symbols for the basic physical user actions are at the lowest level of abstraction and are suggested symbols in the sense that the UAN is an open notation, often adapted and extended by interface designers. Tasks composed of these actions are named, and the names are used as references to the task descriptions in order to build up levels of abstraction in a task structure as described in Section 9.1.

As an example, consider a hypothetical Calendar Management System (CMS) that maintains appointments in a small database. The main interface object is the display of a calendar with views for day, week, and month (as shown in Figure 3) through which the user can navigate. The paradigm for adding, modifying, or deleting an appointment is simple and analogous to the paper calendar: Find the correct day (via day, week, and/or month views) and hour and type into the appointment spaces. There are also commands for searching the calendar and for help information. The highest level UAN task description for using CMS might be as shown in Figure 4. Each time CMS is used, the user makes one choice from among its basic functions: access_appointment, add_appointment, update_appointment, delete_appointment, or establish_alarm. This choice is represented in Figure 4 by the disjunction symbol (|). The task of selecting and executing one basic function can be performed any number of times, represented in Figure 4 by the * symbol. Task analysis is the process that reveals the need for the basic tasks in Figure 4, but details of the methods for performing those tasks may not be
**Figure 2.** State transitions diagram for task A, a simple example of interleaving.

<table>
<thead>
<tr>
<th>Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Move the cursor</td>
</tr>
<tr>
<td>[X]</td>
<td>The context of object X, the &quot;handle&quot; by which X is manipulated</td>
</tr>
<tr>
<td>![X]</td>
<td>Move cursor into context of object X</td>
</tr>
<tr>
<td>![X,Y]</td>
<td>Move the cursor to (arbitrary) point X,Y outside any object</td>
</tr>
<tr>
<td>![X,Y in A]</td>
<td>Move the cursor to (arbitrary) a point within (relative to) object A</td>
</tr>
<tr>
<td>![X in Y]</td>
<td>Move to object X within object Y (e.g., [OK_icon in dialogue_box])</td>
</tr>
<tr>
<td>![X]</td>
<td>Move cursor out of context of object X</td>
</tr>
<tr>
<td>v</td>
<td>Depress</td>
</tr>
<tr>
<td>^</td>
<td>Release</td>
</tr>
<tr>
<td>X$V</td>
<td>Depress button, key, or switch called X</td>
</tr>
<tr>
<td>X$^</td>
<td>Release button, key, or switch X</td>
</tr>
<tr>
<td>X&quot;abc&quot;</td>
<td>Idiom for clicking button, key, or switch X</td>
</tr>
<tr>
<td>X(xyz)</td>
<td>Enter value for variable xyz via device X</td>
</tr>
<tr>
<td>()</td>
<td>Grouping mechanism</td>
</tr>
<tr>
<td>*</td>
<td>Iterative closure, task is performed zero or more times</td>
</tr>
<tr>
<td>+</td>
<td>Task is performed one or more times</td>
</tr>
<tr>
<td>{ }</td>
<td>Enclosed task is optional (performed zero or one time)</td>
</tr>
<tr>
<td>OR,</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>Separator between condition and action or feedback</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feedback</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Highlight object</td>
</tr>
<tr>
<td>-!</td>
<td>Dehighlight object</td>
</tr>
<tr>
<td>!!</td>
<td>Same as !, but use an alternative highlight</td>
</tr>
<tr>
<td>!-!</td>
<td>Blink highlight</td>
</tr>
<tr>
<td>!(-!)$^n</td>
<td>Blink highlight $n$ times</td>
</tr>
<tr>
<td>@X,Y</td>
<td>At point X,Y</td>
</tr>
<tr>
<td>@X</td>
<td>At object X</td>
</tr>
<tr>
<td>@X,Y in X</td>
<td>At point X,Y in (relative to) object X</td>
</tr>
<tr>
<td>Display(X)</td>
<td>Display object X</td>
</tr>
<tr>
<td>Erase(X)</td>
<td>Erase object X</td>
</tr>
<tr>
<td>X-&gt;</td>
<td>Object X follows (is dragged by) cursor</td>
</tr>
<tr>
<td>X-&gt; -&gt;</td>
<td>Object X is rubber banded as its follows cursor</td>
</tr>
<tr>
<td>Outline(X)</td>
<td>Outline of object X</td>
</tr>
</tbody>
</table>

known at first. (The UAN supports development of the design in any direction of abstraction—top down, bottom up, and inside out.) Later, the subtasks of the access_appointment task might be described in the UAN as shown in Figure 5.

To access an appointment, this task description specifies that the user does any number of search, access_month, access_week, and access_day
Figure 3. Typical user's view of the CMS.

Figure 4. Manage_calendar task description.

Task: manage_calendar

(access_appointment
| add_appointment
| update_appointment
| delete_appointment
| establish_alarm)*

tasks followed by a single access_time_slot task (time slots being containers of appointments). Figure 6 shows further details of the access_month task. The access_week and access_day tasks are very similar.

The first subtask, select(any_month), allows the user to make the month level the current view level and is instantiated by substituting a specific month
Figure 5. **Access_appointment** task description.

```
Task: access_appointment

(search
 | access_month
 | access_week
 | access_day)*
access_time_slot
```

Figure 6. **Access_month** task description.

```
Task: access_month

(select(any_month)
 | move_forward_by_month
 | move_backward_by_month)*
```

on the screen for "any month" and using a parameterized task description\(^1\) for select (see Figure 7). Because the **select**(object) task description is composed of primitive user actions, it is more detailed and contains (among other possibilities) columns for user actions, interface feedback, and interface state.

The symbols are explained here in approximately the order of their appearance. In the first column, the - means to move the cursor, and square brackets, [ and ], around an object denote the context of that object. Thus, -[X] means to move the cursor to the context of X. The context of an object is that by which the object is manipulated, which is often the object itself, or it can be, for example, a circumscribed rectangle or "grab handles" such as those used to manipulate line objects in a drawing application. In Figure 7, the item contained in square brackets denotes any arbitrary object icon, but the modifying condition (¬!) further specifies that the object icon must not be already highlighted. Therefore, the first line in the task reads: Move the cursor to an unhighlighted object icon and depress the mouse button (Mv). The corresponding feedback, shown in the middle column, is highlighting of the object icon (object_icon!). For this task, selection is defined (elsewhere) to be from a mutually exclusive set of object icons. The feedback also indicates that any other object icon already highlighted is now unhighlighted.

---
\(^1\) We have used an interaction style similar to that of the Macintosh in our examples. Macintosh is a registered trademark of Macintosh Laboratories. The UAN is not limited to the Macintosh, and it is not oriented toward any one specific graphical direct manipulation style. However, we have taken advantage of the popularity of the Macintosh desktop concept to illustrate use of the UAN.
Figure 7. Select (object) parameterized task description.

<table>
<thead>
<tr>
<th>Task: select(object)</th>
<th>Interface Feedback</th>
<th>Interface State</th>
</tr>
</thead>
<tbody>
<tr>
<td>[object_icon-] Mv</td>
<td>object_icon-, object_icon-'.</td>
<td>selected = object</td>
</tr>
<tr>
<td>∀ object_icon'!:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>object_icon'!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(object_icon'!: object_icon-!). The resulting interface state (selected = object), shown in the third column, defines the set of selected items to contain exactly the one object whose representing icon is highlighted. This implies that any previously selected objects are now unselected and makes explicit the difference between selection of an object and highlighting of the icon that represents the object. The select task is completed in the last line of the task description by releasing the mouse button (MA).

Returning to the access_month task description in Figure 6, the first step to select(any_month) makes the month level the current level in navigating the calendar, and the user can then move forward or backward by month. When the user desires to navigate at the week (or day) level, this is accomplished by performing the access_week (or access_day) task that starts with select(any_week), or select(any_day), causing the current level to be the week (or day) level. The overall design for navigation requires access to all levels, supported by a design decision to keep at least one instance (default is current instance) of month, week, and day on the screen at all times.

The task of accessing a time slot is shown in Figure 8. The access_time_slot task in Figure 8 begins with a precondition, called a condition of viability, that means the view level for navigation must be at the day level or the user cannot perform this task. This precondition is met by performing the access_day task, either by itself or as part of the access_appointment task that precedes the access_time_slot subtask, as shown in Figure 5.

The task of adding a new appointment is described in Figure 9. Task transaction diagrams (Hartson et al., 1990), STDs among tasks as states, are a useful representation technique to supplement the UAN. Navigation within the CMS provides a good example where clarity is added by a task transition diagram, as shown in Figure 10.

2 For simplicity, we ignore the more complex reality of the CMS that requires consideration of a containment relation. For example, a month can be selected without a week or day, but selecting a week also selects the containing month and so on.


Figure 8. Access_time_slot task description.

---

Task: access_time_slot

view_level = day:
((scroll_up | scroll_down)*
 select(any_time_slot)

---

Figure 9. Add_appointment task description.

---

Task: add_appointment

access_appointment
edit_appointment

---

The task of establishing the alarm, to notify the user later when an appointment is impending, is described in Figure 11. The condition of viability in the first line ensures that there is a current appointment (or at least a specific time) with which to associate the alarm. The second line establishes the association of an alarm with the appointment. The third line is a matter of using a dialogue box to set parameters such as alarm lead time (how long in advance of an appointment to sound the alarm). This dialogue box is also the means to express standing orders for alarms (such as every week at this day and time).

The set_alarm task to associate an alarm with an appointment (invoked in the second line of Figure 11) is accomplished by dragging a copy of the alarm icon from the upper left-hand corner of the screen (see Figure 3) to the time slot of the appointment. The set_alarm task is detailed in Figure 12.

The first line of Figure 12 contains a condition of viability for the whole task. The first line of feedback (opposite MV) shows that the alarm icon is to be highlighted, if it is not already so. The next line of feedback shows the alarm icon to be an element of a mutually exclusive set of command icons, causing any other already selected icon in the set to be unhighlighted (and unselected) when the mouse button is depressed over the alarm icon. The user action \([x,y]^{*}\) describes movement of the cursor to various arbitrary points over the screen, on the way to the appointment. Feedback for this action shows that the icon itself stays in place at the top of the screen while the outline of a copy of the icon gets dragged away. The feedback for the action of releasing the mouse button (MA) indicates that the copy of the alarm icon is affixed to the appointment display at a specific point \((x', y)\) relative to the appointment itself.

An interesting part of the temporal nature of a task is the phrasing or chunking that occurs among user actions (Buxton, 1983). For example, the
Figure 10. Task transition diagram depicting navigational possibilities among some CMS tasks.

**VIEW LEVEL NAVIGATION**

(EXAMPLE OF NEED FOR S.T.D. AS PART OF REPRESENTATION)

---

task description of Figure 12 clearly and visually delineates the part of the task performed while the mouse button is depressed as everything that occurs in the task description between MV and MA.

As another example of phrasing, consider the task of multiple icon selection with the Shift key, as shown in Figure 13. Here the interval over which the Shift key is depressed is a "phrase" that spans the selection (and/or deselection) of as many icons as desired and can easily be identified visually in the task description.

5. THE NEED FOR TEMPORAL RELATIONS

Temporal relations were not emphasized in Section 4, which introduced the basic UAN. We now begin to discuss the introduction of temporal relations,
Figure 11. Establish_alarm task description.

Task: establish_alarm
view_level = time_slot:
set_alarm
set_alarm_parameters

Figure 12. Set_alarm task description.

Task: set_alarm

<table>
<thead>
<tr>
<th>User Action</th>
<th>Interface Feedback</th>
<th>Interface State</th>
</tr>
</thead>
<tbody>
<tr>
<td>view_level = time_slot:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>![alarm_icon]</td>
<td>alarm_icon!; alarm_icon!, selected = alarm_command</td>
<td></td>
</tr>
<tr>
<td>Mv</td>
<td>∀cmd_icon!; cmd_icon'!</td>
<td></td>
</tr>
<tr>
<td>![x,y]</td>
<td>outline(copy(alarm_icon)) &gt; -</td>
<td></td>
</tr>
<tr>
<td>![appointment_icon]</td>
<td>outline(copy(alarm_icon)) &gt; -; appointment_icon!</td>
<td></td>
</tr>
<tr>
<td>M(\lambda)</td>
<td>display(copy(alarm_icon))</td>
<td>@x',y' in appointment icon</td>
</tr>
</tbody>
</table>

summarized in Figure 14 for reference in this section, into the UAN for use in task descriptions. Formal definitions of the temporal relations are given in Section 9.

The question of temporal aspects enters into the user interface design process when the relative timing of tasks is considered. The easiest case for the designer is often the most constraining for the user. For example, the designer of a sequence requires completion of one task before another is begun. The CMS task description in Figure 9 illustrates a sequence. The user must complete the access_appointment task before beginning the edit_appointment task. The two tasks cannot be active at the same time. However, users often wish to interrupt a task and, while they are thinking of it, perform another task, later resuming the original one. A major purpose of asynchronous direct manipulation interaction styles is to support this kind of interleaved user task behavior. It follows that there is a need for a behavioral way to represent the possibility of interleaving on the part of the user. This need is met by the interleavability relation, which is used to connect these kinds of tasks in UAN task descriptions.

Most design representations leave this question of intertask temporal relationships implicit, if not ambiguous or undefined. Such specifications often lead to arbitrary design on the part of the interface software designer or implementer. For example, in designing for the task of adding a new appointment to the calendar, a designer may look to the interface toolkit for
Figure 13. **Multiple_icon_selection** task description.

<table>
<thead>
<tr>
<th>User Action</th>
<th>Interface Feedback</th>
<th>Interface State</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Sv ![file_icon]</td>
<td>file_icon-!: file_icon!, selected = selected U file</td>
<td></td>
</tr>
<tr>
<td>Mv ![file_icon]</td>
<td>file_icon!: file_icon -! selected = selected - file</td>
<td></td>
</tr>
<tr>
<td>S^+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14. **Summary of UAN temporal relation symbols.**

<table>
<thead>
<tr>
<th>Temporal Relation</th>
<th>UAN Symbology</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>A B</td>
<td>Tasks A and B are performed in order left to right, or top to bottom</td>
</tr>
<tr>
<td>Waiting</td>
<td>A (t &gt; n) B</td>
<td>Task B is performed after a delay of more than n units of time following task A</td>
</tr>
<tr>
<td>Repeating disjunction</td>
<td>(A</td>
<td>B)*</td>
</tr>
<tr>
<td>Order independence</td>
<td>A &amp; B</td>
<td>Tasks A and B are order independent (order of performance is immaterial)</td>
</tr>
<tr>
<td>Interruptibility</td>
<td>A \rightarrow B</td>
<td>Task A can interrupt task B</td>
</tr>
<tr>
<td>One-way interleavability</td>
<td>A \rightarrow B</td>
<td>Task A is one-way interleavable with B (A can interrupt B and execute, but not vice versa)</td>
</tr>
<tr>
<td>Mutual interleavability</td>
<td>A \leftrightarrow B</td>
<td>Task A and task B are (mutually) interleavable</td>
</tr>
<tr>
<td>Concurrency</td>
<td>A + B</td>
<td>Task A and task B can be performed concurrently</td>
</tr>
</tbody>
</table>

an appropriate “widget.” It could be reasonable to the designer to use a preemptive style dialogue box (Thimbleby, 1990), requiring the user to enter information for the appointment before moving on to the next task. In contrast, a user may seek information from an existing appointment while in the midst of creating a new appointment. Or a user may wish to create two or more related appointments at once, or to set the alarm while still creating an appointment. Good interface design suggests that the designer will at least allow the user to close the dialogue box without completing the associated data
entry task and that any information entered so far will be retained. A good
design might also provide copy-and-paste operations for moving information
from one appointment to the other. But the user might still be left with the
responsibility of closing one task and opening the other and often must use
human working memory to carry certain information from one task context
to another. From the user view, there is a task interruption, but the design
does not support it well. Proper evaluation and design iteration will lead to a
better design, but temporal relations in the behavioral representation tech-
niques can help in two ways. First, temporal considerations might be in the
design but cannot be explicit in its representation without temporal relations.
The UAN temporal relations allow the designer to declare explicitly the
temporal relationships among the tasks. Second, treatment of temporal
aspects in this context is ad hoc, whereas temporal relations in the UAN help
the designer to think a priori about temporally related design issues.

In retrospect, many UAN temporal relationships may appear deceptively
obvious, but without them it is very difficult to discuss important asynchro-
nous aspects of interface designs with precision and to distinguish among
temporal alternatives within a design. In the next section, we begin to develop
the formalization of temporal relations in behavioral interface representa-
tions. As mentioned at the end of Section 1, those interested in just an
intuitive understanding of the temporal relations can skim or skip over
Sections 6, 7, and 8.

6. TIME

In what follows, we take as given that our universe of discourse contains
time, which is a one-dimensional quantity, made up of points, where each
point is associated with a value. The points are ordered along the dimension
by their values. The common concepts of later and earlier correspond to
larger and smaller values of time, respectively. The view of time taken here
is compatible with the traditional psychological, thermodynamic, and cosmic
views of time (Hawking, 1988).

Nothing we say in this article depends on whether this quantity is discrete
or continuous. User behavior certainly occurs in continuous time. At the
lowest level, most corresponding computer events occur in discrete time—
Continuous user inputs are sampled in the hardware, and outputs are subject
to timing constraints (e.g., a system clock). Resolution of time, however, is
usually sufficiently fine so that the difference in views from user to computer
is insignificant.

An example of a case in which sampling resolution does make a difference
is seen in a Macintosh interface when using multiple display monitors, for
example, with one on top of the other. Depressing the mouse button within
the menu bar causes the corresponding menu display. With a single monitor, a user cannot move the cursor above the menu bar. However, the second monitor can provide screen space above the normal application display. In this configuration, moving up to the bar, and beyond into the second screen above, causes the menu to disappear. It is possible, though, to move up through the bar fast enough so that the cursor position is not sampled within the bar. In this case, the menu remains displayed, even though the cursor could not have gotten above the bar without passing through it. Fortunately, examples such as this are more oddities of timing than real interface problems.

The rest of Section 6 is devoted to the fundamental notions that events happen in time and that intervals of time occur and can be compared to determine if one precedes another or if two or more intervals overlap in time.

6.1. Events in Time

Things that happen in the world (i.e., events) can be thought of as happening in time; that is, each event can be associated with a set of points in time so that it is possible to answer questions such as: "Given a point in time, $t$, and an event, $e$, is $e$ Happening at $t$?" Formally, where $t$ is a point in time and $e$ is an event, consider a binary relation $H$ such that:

$$e \ H \ t \Leftrightarrow \text{event } e \text{ is Happening at time } t$$  \hspace{1cm} (1)

Note that the right-hand side of this definition involves an appeal to the physical world, and thus it is a postulate of the model that the right-hand side can be evaluated. (The reader is referred to the Appendix for an explanation of mathematical notation used in the equations and elsewhere in this article.)

6.2. Time Intervals

An interval of time is an ordered set defined by an ordered pair of two points in time. Thus, an interval of time, $T$, denoted by $[t_1, t_2]$ is defined:

$$T = \{ t \mid (t \geq t_1) \land (t \leq t_2) \}$$  \hspace{1cm} (2)

We define two projection functions on intervals, B and E, to extract their Beginning and Ending points. Where $T$ is the interval $[t_1, t_2]:$

$$B(T) = t_1$$  \hspace{1cm} (3)

$$E(T) = t_2$$  \hspace{1cm} (4)
Our adoption of time points as primitives, and the definition of intervals in terms of them, is in contrast to Allen's theory (see Section 3.3). Allen argues that the use of points of time as a primitive leads to certain semantic difficulties, particularly in handling change over time. Thus, if an action, say selecting a menu item, is defined in terms of its temporal endpoints, and time is continuous, then there must exist a time at which the user is neither selecting nor not selecting the menu item. As we argued in the previous section, however, we are not committed to treating time as continuous for the purposes of modeling user actions. Furthermore, as a consequence of taking intervals as primitives, Allen is forced to introduce separate concepts of event (indivisible through its defining interval) and process (interruptible during its defining interval). Our approach avoids this problem, resulting in an ontology containing only one type of entity for actions and an account of interruptibility with greater explanatory power (see Section 9.5).

6.3. Preceding and Overlapping

Two important relations between intervals are Precedes and Overlaps, denoted respectively by P and O. Where \( T_1 \) and \( T_2 \) are intervals:

\[
T_1 \mathbin{P} T_2 \iff \forall i, j((t_i \in T_1) \land (t_j \in T_2)) \Rightarrow (t_i < t_j)
\]  \hspace{1cm} (5)

\[
T_1 \mathbin{O} T_2 \iff \exists t((t \in T_1) \land (t \in T_2))
\]  \hspace{1cm} (6)

The following section formalizes the concepts of task and user action, as used in the UAN. Then Section 8 relates user actions to time, setting the stage for the development of UAN temporal relations in Section 9.

7. TASKS AND ACTIONS

The primary abstraction of the UAN is the task. A human-computer interface is represented as a quasi-hierarchical structure of asynchronous tasks, the sequencing within each task being independent of that in the others. Each task is, in turn, represented in a notation describing user actions and interface feedback, offering a structured way to describe the cooperative performance of a task between a human user and a computer system.

The UAN was originally created to provide a pragmatic and effective means for conveying interface design ideas from designers to implementers and evaluators. It is a goal of this article to be more precise about the concepts of task and user action, which were not formally defined in the original UAN. Additionally, we wish to make the connection between user actions and time.
7.1. Basic Definitions

The basic concepts of UAN are those of task, action set, and user action. With the inclusion of temporal relations, a UAN task is an ordered triple:

\[
\text{task} = \langle \text{action set}, \text{temporal relation set}, \text{application function} \rangle \quad (7)
\]

The elements of the temporal relation set, when applied by the application function, specify the temporal relationships among actions in the action set. A user action is either a primitive user action or a task:

\[
\text{user action} = \text{primitive} \mid \text{task} \quad (8)
\]

The action set of a task, \( \alpha \), is the union of all user actions mentioned in the description of \( \alpha \) and is obtained by applying the projection function, \( A(\alpha) \), to the triple of Equation 7.

The definition of task in Equation 7 is recursive in that the elements of the action set may themselves be tasks via Equation 8. The primitives of the UAN, into which all tasks may be decomposed, are simply those actions which, by definition, are not further decomposed; these include basic physical operations by the user on input devices (e.g., cursor movement, mouse button press and release, keypresses). Task descriptions can also include memory, cognitive, perceptual, and decision-making user actions (Sharratt, 1990), but they are not discussed here. The boolean function, \( \text{prim}(\alpha) \), is used to determine if an action, \( \alpha \), is primitive:

\[
\text{prim}(\alpha) = \begin{cases} 
\text{true, if } \alpha \text{ is a primitive user action;} \\
\text{false, otherwise}
\end{cases} \quad (9)
\]

In the following sections, the contents of the latter two elements of the task (viz., the temporal relation set and the application function) are discussed in relation to user actions. It should be noted that task descriptions using the UAN include annotations referring to feedback, display state, and communication with the application. Although these are essential in determining the adequacy of a design specified by means of the UAN, these annotations are not germane to the issues discussed in this article.

7.2. Instances of Tasks

The UAN uses the names of actions (primarily tasks in this context) as intensional design-time references to extensional run-time instances or invocations of those tasks. The intensional descriptions specify constraints on
temporal possibilities for extensional instantiations of the tasks within a specific performance of the containing task. We use the term instance as needed for clarity. However, the terms task and action and related terms are often sufficient to refer to their instances, unless it is important to make the distinction.

8. TIME AND ACTIONS

8.1. Actions as Happenings in Time

Instances of user actions are events in time, and thus we wish to apply the relation H to them, where α is an instance of a user action and t is a point in time:

\[ \alpha \text{ H } t \iff \text{action } \alpha \text{ is Happening at time } t \]  \hspace{1cm} (10)

Again, it is postulated that the right-hand side can be evaluated for any user action, α. Equation 10 is fundamental in that it relates user actions to time.

8.2. Lifetimes

An instance of a user action, α, has a Lifetime, denoted L(α), which is the interval spanning just those times that satisfy H:

\[ L(\alpha) = [B(L(\alpha)), E(L(\alpha))] \]  \hspace{1cm} (11)

where the Beginning of the Lifetime, B(L(α)), is the least point in time such that the action instance is happening at that time:

\[ B(L(\alpha)) = t \ \exists \ ((\alpha \text{ H } t) \land \neg \exists t'((t' < t) \land (\alpha \text{ H } t'))) \]  \hspace{1cm} (12)

and the End of the Lifetime, E(L(α)), is the greatest point in time such that the action instance is happening at that time:

\[ E(L(\alpha)) = t \ \exists \ ((\alpha \text{ H } t) \land \neg \exists t'((t' > t) \land (\alpha \text{ H } t'))) \]  \hspace{1cm} (13)

8.3. The Boustrophedon Argument

Given these definitions, a user action (primarily a task in this context) need not be happening at all times during its lifetime; there may be times of inactivity as well as times of activity. The graph of activity versus time, which we call an activity waveform diagram, is a boustrophedon (alternating rectangu-
Figure 15. The boustrophedon activity waveform and its envelope, the lifetime of the task instance.

The boustrophedon (alternating) waveform. By the previous definition, however, the lifetime of an instance of a task is the "envelope" of the corresponding boustrophedon waveform, as shown in Figure 15.

8.4. Interruption

One way that a task can become inactive is due to interruption by another task. An interruption occurs when the user and system activities of one task are suspended before the end of the task's lifetime and the activity of another task is begun in its place. Task interruption usually occurs due to actions initiated by the user, but they can also be the result of system-initiated actions (e.g., to update a clock or announce the arrival of electronic mail).

8.5. Idle Time

Another way that a task can be inactive is due to idle time, when neither user nor system is doing anything significant to this task. All tasks are decomposable into primitive physical user and system actions. There are natural lulls between physical actions—times between keystrokes and pauses to see, to think, or to get a cup of coffee. These lulls correspond to inactive periods in the activity waveform diagram but, by the boustrophedon argument, are part of the lifetime of the task.

8.6. Periods of Activity

Formally, a period of activity, \( \pi \), of a task, \( \alpha \), is an interval such that \( \alpha \) is happening at all times in the interval:
\[ \pi(\alpha) = [t_1, t_2] \ni \forall t_i ((t_1 \leq t_i \leq t_2) \supset (\alpha \bowtie t_i)) \] (14)

The lifetime of a task, then, contains one or more periods of activity. As just noted, a period of activity of an instance of a task continues until it is terminated by interruption from another task or by inactivity within itself.

9. TEMPORAL RELATIONS AMONG USER ACTIONS

In this section, several temporal relationships among user actions are identified and formally represented. In its simplest form, each relationship is represented as a binary relation between two user actions: \( \alpha_1 R \alpha_2 \). In this context, it is not very useful to regard these relations as mapping operators, to think of giving an \( \alpha_1 \) in the domain of \( R \) and yielding an \( \alpha_2 \) in the range. As mappings, the relations described here are usually not total and not functional and are often many-to-many. Rather, it is better to think of these relations as algebraic combining operators. If user actions \( \alpha_1 \) and \( \alpha_2 \) are related by \( R \), \( \alpha_1 R \alpha_2 \), it means that they bear a certain temporal relationship within a task, and one can perform a kind of abstraction by combining them; that is, apply \( R \) to \( \alpha_1 \) and \( \alpha_2 \) and, by closure (see Section 9.1), get a new user task, \( \alpha_3 = R(\alpha_1, \alpha_2) \).

The most basic temporal relationships we have identified are:

- sequence
- waiting
- repeated disjunction
- order independence
- interruptibility
- one-way interleavability
- mutual interleavability
- concurrency

The set of temporal relations in a task definition defines a set of constraints (or, perhaps, the relief of constraints) among the elements of the action set. For example, if two tasks are related by a sequence, the temporal possibilities for their performance are completely constrained. On the other hand, if the same tasks are related by mutual interleavability, they are less constrained temporally, allowing the user more freedom with respect to their relative timing. Of course, that freedom is not necessarily exercised by the user at run-time; given that a set of actions can be interleaved by the user, it does not
follow that they are interleaved. From the point of view of describing human–computer interaction for design purposes, the interesting relationships are those that express the possibilities for actions.

As mentioned at the end of Section 1, readers not wishing to get into the details of symbols and equations can skip the numbered equations in Section 9 and still understand most of the concepts. Each equation is preceded by a prose description.

9.1. Sequence

Perhaps the simplest temporal relationship between two tasks is that which is expressed by the binary relation sequence; one task is performed immediately and entirely after the other. More formally, two user actions, $\alpha_1$ and $\alpha_2$, are in sequence (related by the sequence relation, $S$) if and only if the entire lifetime of $\alpha_1$ immediately precedes the lifetime of $\alpha_2$:

$$\alpha_1 S \alpha_2 \leftrightarrow ((L(\alpha_1) \ P \ L(\alpha_2)) \land \neg \exists \pi_i(\alpha_j)((L(\alpha_1) \ P \ \pi_i(\alpha_j)) \land (\pi_i(\alpha_j) \ P \ L(\alpha_2)))) \text{, for } j \neq 1, j \neq 2$$ (15)

Note that this sense of sequence, which does not allow an intervening action between two actions in sequence, can be thought of as a strong precedence relation. This observation will be of importance when examining the concepts of interleaving and interruption. Figure 16 is an activity waveform diagram illustrating a sequence. The actions shown here could all be different or could involve different instances of the same task. Notice that each action instance is performed to completion before another is begun; no interruption is occurring.

In the UAN, a sequence is represented in the following way. The $S$ is dropped, and the temporal sequence of actions, $\alpha_1$ and $\alpha_2$, is represented iconographically by writing the actions as a spatial sequence horizontally:

$$\alpha_1 \ \alpha_2$$
or vertically:

\[ \alpha_1 \]
\[ \alpha_2 \]

As an example from the CMS, a high-level task may be defined as the sequence of two other tasks (as in Figure 9):

**Task: add_appointment**

access_appointment

edit_appointment

The task of adding an appointment is defined to be the sequence of accessing the appropriate appointment followed by the editing (including typing, corrections, etc. in predefined fields in the time slot) of the appointment.

So far, a sequence is a binary temporal relation; that is, it applies to exactly two tasks as operands. There are two ways that sequences, and the other temporal relations, can be applied on a larger scope. One way is to build up levels of abstraction; the second way is by grouping with parentheses. In the next two subsections, it is convenient to define these two methods of expansion in terms of sequences, but the concepts apply to each of the temporal relations equally well.

**Task Names and Levels of Abstraction**

A task description written in the UAN is a set of actions interspersed with temporal operators according to the rules for their application, following Definitions 7 and 8. This task can then be named, and the name is used as a reference to the task. This name reference is used as an action in another (higher level or containing) task. As an example, consider a simple task that has only a sequence, \( \alpha_1 \alpha_2 \). This task can be named "\( \beta \)," and then \( \beta \) can be used in task \( \gamma \) in sequence with some other task \( \epsilon \). The use of a task name as a user action corresponds at run-time to the invocation of a user-performed procedure. The use of a task name as a reference to the task is an *invocation* and serves two purposes (just as invocations do in programming systems): abstraction (hiding the details of the procedure) and instantiation (creating a task instance—see Section 7.2—and giving it a lifetime).

The recursive nature of this abstraction operation makes it possible to build layers of abstraction, allowing the entire interface design to be organized into a quasi-hierarchical user task structure. Just as in the case of program code, the levels of abstraction are necessary for controlling complexity to promote understanding by readers and writers of the UAN. Also, just as in the case of software procedures and their calling structure, there will be a path of active
tasks down to the level of primitives. To illustrate with the example just given, consider the performance of task \( \gamma \). At some time \( \beta \) will be invoked from within \( \gamma \). During the performance of \( \beta \), task \( \alpha_1 \) will also be performed. At that moment, all of the tasks \( \alpha_1, \beta, \) and \( \gamma \) will be active. This kind of simultaneity is only an artifact of the hierarchical decomposition structure of tasks; a "calling" task and a "called" task will always have overlapping lifetimes. This is not the same, however, as two independent tasks being interleaved or concurrent. All the temporal relations described in this article are applied to independent tasks at the same level of abstraction—not between calling and called tasks in the task hierarchy.

**Grouping, Closures, and Composition of Relations**

An instance of a temporal relation between two tasks can be enclosed within parentheses. The effect is similar to the grouping into a named task as described in the previous section, except the resulting task is not named. For example, the sequence of actions:

\[
\alpha_1 \alpha_2
\]

can be grouped with parentheses into the following task:

\[
(\alpha_1 \alpha_2).
\]

Each of the temporal relations, \( R \), maps a pair of actions into a task:

\[
R: \{\text{actions}\} \times \{\text{actions}\} \rightarrow \{\text{tasks}\}
\]

and a task is also an action; thus the temporal relations are closed over the set of all actions. Therefore, by composition, one can apply another relation between a group in parentheses and some third action, \( \alpha_3 \), yielding a new task, as in the case of this sequence:

\[
(\alpha_1 \alpha_2) \alpha_3.
\]

Composition of relations allows large task description to be built up of user actions (especially tasks) and temporal relations.

Applying the concept of grouping to the sequence relation, one can derive the property of associativity directly from the definition of the sequence relation in Equation 15:

\[
(\alpha_1 \alpha_2) \alpha_3 = \alpha_1 (\alpha_2 \alpha_3)
\]
The binary sequence relation can be generalized to the ternary case by extending Equation 17 in this way:

\[(\alpha_1 \alpha_2) \alpha_3 = \alpha_1 (\alpha_2 \alpha_3) = (\alpha_1 \alpha_2) \alpha_3 = \alpha_1 \alpha_2 \alpha_3\]  

\[\text{(18)}\]

Similarly, the sequence relation can be extended to the n-ary case:

\[\alpha_1 \alpha_2 \alpha_3 \ldots \alpha_n\]  

\[\text{(19)}\]

9.2. Waiting

Sometimes an interface designer wishes to constrain the time interval between tasks in a sequence. For example, to define a close relationship that combines two tasks into one, the interval could be required to be less than some time value. To illustrate, two mouse-button clicks, when performed within a short interval, are to be recognized as a distinct user action called a double click. In such cases where waiting is significant in a task description, the waiting interval acts as a temporal relation between the actions, constraining the temporal distance between actions in a sequence. Within a UAN task description, a waiting relation between tasks \(\alpha_1\) and \(\alpha_2\) is written as:

\[\alpha_1 (t \text{ comparison-operator } n) \alpha_2\]  

\[\text{(20)}\]

where \(t\) is the time to wait, comparison-operator makes an arithmetic comparison (such as less than or greater than), and \(n\) is a numeric value in units of time.

The example of the double click of a mouse button is represented in this specific UAN expression:

\[\text{MVA} (t < n) \text{ MVA}\]

where \(\text{MVA}\) denotes the mouse button being depressed and released (clicked) and \((t < n)\) declares that the wait between mouse clicks must be less than \(n\) units of time. If the user waits longer than \(n\) time units, this action will be seen as two single mouse clicks. The value of \(n\) can be controlled by the user via an interface setting.

Another way waiting can be used in a UAN description as a temporal relation between two tasks is to indicate a minimum wait to cause some kind of time-out by the system:
\( \alpha_1 \ (t > n) \ \alpha_2 \)

### 9.3. Repeating Disjunction

The vertical bar (|) is used to indicate a disjunction of choices among user tasks. For example, \( \alpha_1 \ | \ \alpha_2 \ | \ \alpha_3 \) denotes a three-way choice among \( \alpha_1, \alpha_2, \) and \( \alpha_3 \). A common high-level construct in the UAN is seen in this example of a repeating disjunction:

\[
(\alpha_1 \ | \ \alpha_2 \ | \ \alpha_3)^*
\]

This notation means that tasks \( \alpha_1, \alpha_2, \) and \( \alpha_3 \) are initially equally available. The * means that the disjunction (the whole task within parentheses) is repeated any number of times. Once a task from the disjunction is begun, it is performed to completion, at which time the three tasks are equally available again. The cycle continues arbitrarily each time any one of the three tasks is selected by the user and performed to completion.

As an example from the CMS, the highest level task is defined in Figure 4 as the repeating disjunction of the five main user operations:

---

**Task: manage_calendar**

- (access_appointment
  - | add_appointment
  - | update_appointment
  - | delete_appointment
  - | establish_alarm)^*

---

Repeating disjunction is also used in the access_appointment task definition of Figure 5:

---

**Task: access_appointment**

- (search
  - | access_month
  - | access_week
  - | access_day)^*
  - access_time_slot

---

Observable user behavior in performing the access_appointment task is a series of instances from among the five tasks of search, access_month, access_week, and so on. Ways in which the user might decide which choices to make can be described within the access_appointment task using cognitive, perceptual, and decision-making activities, but these are not in the scope of the present article.
9.4. Order Independence

In the use of interactive computer systems, as in the world outside interfaces, it is not uncommon to find situations in which several tasks are to be performed but the order of their performance is immaterial. In the UAN, two user actions, $\alpha_1$ and $\alpha_2$, are order independent if and only if both actions are required, but the lifetime of either may precede that of the other:

$$\alpha_1 \& \alpha_2 \leftrightarrow ((\alpha_1 \& \alpha_2) \mid (\alpha_2 \& \alpha_1))$$ (21)

The order independence relation is not associative, but it nonetheless can be extended to the n-ary case:

$$\alpha_1 \& \alpha_2 \& \ldots \& \alpha_n$$ (22)

where this expression denotes a disjunction of all the possible sequential orderings of the actions. In practical terms, this means that all of the tasks—$\alpha_1$, $\alpha_2$, $\ldots$, $\alpha_n$—must be performed but that any order among them is acceptable.

An example of order independence at a very low user action level is seen in the task of entering a "command-X" on a Macintosh keyboard—a combination of the "⌘" and "X" keys. The UAN uses "\(^v\)" to denote the depressing of an input device such as a key or mouse button. The symbol \(^v\) is used to indicate the release of such a device. The symbol on the key is the name of the device that is the key. Because the "⌘" key must be depressed before the X key, but the order of their release does not matter, the task is defined in the UAN as:

```
Task: command_X
   \(^v\) XV (\(^v\) & XV)
```

The `edit_appointment` task provides an example of order independence from the CMS. Suppose an appointment object has text fields for name of person, description of appointment, and location. The task of editing an appointment breaks down into the set of tasks for editing these smaller objects, and the order in which they are edited does not matter:

```
Task: edit_appointment
view_level = time_slot:
   (edit_person
   & edit_description
   & edit_location)
```
Figure 17. The simplest case of interruption.

The edit_person, edit_description, and edit_location tasks will feature repeating disjunctions of editing subtasks such as type_string, select_string, cut_string, copy_string, paste_string, and the like.

9.5. Interruptibility

We begin by refining the concept of interruption, introduced earlier in Section 8.4. An instance of an action, \( \alpha_2 \), is interrupted by another action, \( \alpha_1 \), if and only if a period of activity of \( \alpha_1 \) overlaps the lifetime of \( \alpha_2 \) but does not overlap a period of activity of \( \alpha_2 \):

\[
\alpha_2 \text{ is interrupted by } \alpha_1 \iff \\
\exists \pi_i(\alpha_1)((\pi_i(\alpha_1) \circ L(\alpha_2)) \land \neg \exists \pi_j(\alpha_2)((\pi_j(\alpha_1) \circ \pi_j(\alpha_2)))
\]

(23)

The simplest case of interruption is shown in Figure 17. Task \( \alpha_1 \) is begun and task \( \alpha_2 \) interrupts, dividing \( \alpha_1 \) into two periods of activity, \( \pi_i(\alpha_1) \) and \( \pi_i(\alpha_2) \). The lifetime of \( \alpha_1, L(\alpha_1) \), spans the two periods of activity.

Because a design representation is intensional, there is no symbol in the UAN for “is interrupted by.” Rather, there is a temporal operator to denote cases of interruptibility, cases where interruption can occur. Thus, the definition of interruptibility requires the use of alethic (truth-related) modalities in our expressions. That is, the defining proposition must assert the possibility of a certain state of affairs. For this purpose, we add to the first-order predicate calculus used so far the primitive monadic modal operator, \( M \) (Hughes & Cresswell, 1968), with the following definition:
 TEMPORAL ASPECTS OF TASKS IN THE UAN  

\[ Mp = \text{it is possible that } p \text{ (i.e., it is not a tautology that } \neg p) \]  

(24)

Note that M expresses an alethic rather than a temporal (time-related) modality; although we are speaking of temporal relations, we do not use temporal modes.

An instance of an action, \( \alpha_2 \), is defined to be interruptible by another action, \( \alpha_1 \) (\( \alpha_1 \) can interrupt \( \alpha_2 \)), if and only if a period of activity of \( \alpha_1 \) can overlap the lifetime of \( \alpha_2 \) but cannot overlap a period of activity of \( \alpha_2 \):

\[
\alpha_1 \rightarrow \alpha_2 \leftrightarrow M(\exists \pi_i(\alpha_1)((\pi_i(\alpha_1) \circ L(\alpha_2)) \land \neg \exists \pi_j(\alpha_2)(\pi_i(\alpha_j) \circ \\
\pi_j(\alpha_2))))
\]

(25)

The interruptibility relation is not symmetric; \( \alpha_1 \rightarrow \alpha_2 \) implies neither \( \alpha_2 \rightarrow \alpha_1 \) nor \( \neg (\alpha_2 \rightarrow \alpha_1) \).

Uninterruptible Tasks and Preemptive States

Consider a task \( \alpha \) for which the action set is \( A(\alpha) \). If task \( \alpha' \) can interrupt task \( \alpha \), \( \alpha' \rightarrow \alpha \), there are two ways that the definition in Equation 25 can be satisfied: \( \alpha' \) can interrupt between the lifetimes of instances of the \( \alpha_1 \in A(\alpha) \), or interruption can occur during the lifetime of an \( \alpha_i \); that is, \( \alpha \rightarrow \alpha_i \). The general interpretation of the interruptibility relation includes both these cases.

It is also necessary to be able to define exceptions to this second case, namely, to be able to specify those \( \alpha_i \) for which \( \neg (\alpha' \rightarrow \alpha_i) \). One kind of exception occurs when \( \alpha_i \) is primitive, denoted by the unary relation \( \text{prim}(\alpha) \). Primitive user actions are not interruptible.

A second situation in which a task instance must be specified as uninterruptible occurs in preemptive interface features (Thimbleby, 1990). A dialogue box is a good example. While using a dialogue box in task \( \alpha_1 \), a user generally cannot click in the window of task \( \alpha_2 \) to change tasks until the dialogue box is exited. Preemptive states correspond to sets of user actions, the boundaries of which cannot be crossed by the interleaving relation. In other words, while in the dialogue box, the user can still interleave tasks but only among tasks within the dialogue box. In the UAN, pointed brackets, \( <,> \), represent the unary relation "is uninterruptible," enclosing those parts of a task description that are uninterruptible by other user actions at any level. For example, \( < \alpha_1 \alpha_2 \alpha_3 > \) denotes that the sequence of these user action instances cannot be interrupted.

Preemptive states in this view are a means of partitioning the user's task domain. A preemptive state is a task subdomain with circumscribed asynchronism. A preemptive state limits the user to a set of tasks usually disjoint from those available outside that state. Consider the graph or set of graphs that is the nondeterministic state transition diagram of the dialogue control for an interface. The part of the dialogue without preemptive states
can be considered the main dialogue. The main dialogue and the set of preemptive states would each be simply-connected components of the graph, the preemptive states being isolated from the rest of the interface except for the single transitions entering and leaving the preemptive state set. Modes are usually preemptive; consider the input mode in the Unix "vi" editor. There are many commands that lead to the input mode (open line, append, input, etc.) at which point almost all keystrokes are considered as input text. The Escape key allows the user to leave the input mode, and many vi keyboard commands once again become active. Inputs that apply in the input mode are more or less disjoint from the commands that apply to vi outside the input mode.

Modes and preemptive states in interface designs are the result of decisions (conscious or not) about the task domain. It is not our intention to argue for or against such decisions here, only to be able to represent the designs.

**Scope of Interruptibility**

To understand the effect of interruptibility on a task or action, \( \alpha \), it is useful to determine which subtasks (tasks or actions invoked by \( \alpha \)) themselves are interruptible. We must begin by formalizing the concept of invocation, introduced in Sections 7.2 and 9.1. User action \( \alpha' \) can directly invoke user action \( \alpha \) (\( \alpha \) is directly invocable by \( \alpha' \)) if and only if \( \alpha \) is a member of the action set of \( \alpha' \):

\[
\alpha' \triangleright \triangleright \alpha \iff \alpha \in A(\alpha') \tag{26}
\]

Action \( \alpha' \) can invoke action \( \alpha \) (\( \alpha \) is invocable by \( \alpha' \)) if and only if there is a progression of possible direct invocations, \( \alpha_1, \alpha_2, \ldots, \alpha_k \), connecting \( \alpha' \) and \( \alpha \):

\[
\alpha' \triangleright \triangleright \alpha \iff \exists (\alpha_1, \alpha_2, \ldots, \alpha_k) \ni
\begin{align*}
\text{i. } & \alpha' \triangleright \triangleright \alpha_1 \\
\text{ii. } & \alpha_i \triangleright \triangleright \alpha_{i+1}, \text{ for } i = 1, 2, \ldots, k - 1 \\
\text{iii. } & \alpha_k \triangleright \triangleright \alpha 
\end{align*} \tag{27}
\]

It follows that \( \alpha' \triangleright \triangleright \alpha \supseteq \alpha' \triangleright \triangleright \alpha \), where \( \alpha_1, \alpha_2, \ldots, \alpha_k \) is a null progression.

In like manner, we define the "can interruptibly invoke" relation \( (\ast) \). Action \( \alpha' \) can interruptibly invoke action \( \alpha \) (\( \alpha \) is interruptibly invocable by \( \alpha' \)) if and only if \( \alpha' \) can invoke \( \alpha \) and \( \alpha \) is neither uninterruptible nor a primitive.

\[
\alpha' \cdot \alpha \iff ((\alpha' \triangleright \triangleright \alpha) \land \neg (\triangleleft \alpha \lor \text{prim}(\alpha))) \tag{28}
\]

If \( \alpha' \to \alpha \), the full set of user actions collectively known as the *scope of interruptibility* is \( \alpha \) plus all the user actions invocable by \( \alpha \), except primitives
and uninterruptible actions; that is, the scope of interruptibility is $\alpha$ and all user actions interruptibly invocable by $\alpha$:

$$\text{I}(\alpha', \alpha) = \{\alpha\} \cup \{\alpha^* | \alpha \cdot \alpha^*\}$$ (29)

### 9.6. One-Way Interleavability

There may be times when the interface designer wishes to specify $\alpha_1 \rightarrow \alpha_2 \land \neg(\alpha_2 \rightarrow \alpha_1)$. For example, consider the case of help as a facility available during some other complex task such as the editing of a document. If the high-level task is described as follows:

$$\text{help} \rightarrow \text{edit document} \land \neg(\text{edit document} \rightarrow \text{help})$$

the user can invoke the help task at any time during the editing, but closure of the help task is required before editing can continue. In other words, help tasks can interrupt the editing, but editing cannot interrupt the help. We call this one-way interleavability.

An instance of an action, $\alpha_1$, is defined to be one-way interleavable with action $\alpha_2$ if and only if $\alpha_1$ can interrupt $\alpha_2$ but $\alpha_2$ cannot interrupt $\alpha_1$:

$$\alpha_1 \Rightarrow \alpha_2 \iff ((\alpha_1 \rightarrow \alpha_2) \land \neg(\alpha_2 \rightarrow \alpha_1))$$ (30)

### 9.7. Mutual Interleavability

Two user actions, $\alpha_1$ and $\alpha_2$, are mutually interleavable if and only if they can interrupt each other; that is, it is possible that a period of activity of either action can interrupt a period of activity of the other:

$$\alpha_1 \Leftrightarrow \alpha_2 \iff ((\alpha_1 \rightarrow \alpha_2) \land (\alpha_2 \rightarrow \alpha_1))$$ (31)

Unqualified use of the terms interleaving and interleavability is reserved for the more general two-way (mutual) case. As previously discussed, $\alpha_1 \Leftrightarrow \alpha_2$ means that $\alpha_1$ and $\alpha_2$ are mutually interleavable throughout the scopes of interruptibility of $\alpha_1$ and $\alpha_2$.

One can derive the following property of associativity directly from the definition of the interleavability relation in Equation 31:

$$(\alpha_1 \Leftrightarrow \alpha_2) \Leftrightarrow \alpha_3 = \alpha_1 \Leftrightarrow (\alpha_2 \Leftrightarrow \alpha_3)$$ (32)

The binary interleavability relation, again, can be generalized to the ternary case by extending Equation 32 in this way:
Figure 18. Activity waveform diagram of interleaved tasks.

\[
\begin{align*}
(\alpha_1 \Leftrightarrow \alpha_2) \Leftrightarrow \alpha_3 &= \alpha_1 \Leftrightarrow (\alpha_2 \Leftrightarrow \alpha_3) = (\alpha_1 \Leftrightarrow \alpha_2 \Leftrightarrow \alpha_3) = \\
\alpha_1 \Leftrightarrow \alpha_2 \Leftrightarrow &\alpha_3
\end{align*}
\]

(33)

and to the n-ary case:

\[
\alpha_1 \Leftrightarrow \alpha_2 \Leftrightarrow \ldots \Leftrightarrow \alpha_n
\]

(34)

Interleavability is one of the cases in which it is necessary to distinguish between tasks and instances of tasks (Section 7.2). At run-time it is the instances, of course, that are interleaved. Consider the case of help as a facility available during some other complex task (e.g., editing a document). Suppose that the help information, when invoked, appears in a separate window from the document being edited. The editing and help tasks are interleavable in that the user may alternate attention, and actions, from one window to the other. There is only one instance of each user task: one editing task and one help task. Additionally, it is possible that the user might terminate one help task during the editing task and subsequently start another interleaved help task while still within the initial editing session. This is a case of two instances of the same task type (e.g., help) being interleaved with a single instance of a distinct task (e.g., editing). Thus, \( \alpha \Leftrightarrow \alpha \) does not mean that interleavability is reflexive; rather, this expression refers to the interleavability among different instances of \( \alpha \).

There are many possible configurations of interleaving. Figure 18 shows several interleaved tasks. Task \( \alpha_1 \) is interleaved with \( \alpha_2 \) and with \( \alpha_3 \), but \( \alpha_2 \) is not interleaved with \( \alpha_3 \). Task \( \alpha_2 \) is not interrupted by \( \alpha_1 \) after the period of activity \( \pi_{\text{th}} \), because this period is the termination of \( \alpha_2 \). In general \( \alpha_1, \alpha_2, \) and \( \alpha_3 \) are different tasks, but again it is possible for interleaving to involve
different instances of the same task. For example, two help windows could be
open simultaneously, with the user shifting attention from editing to each of
the help windows alternately.

An example from the CMS can be used to illustrate interleaving. The five
main user operations shown in Figure 4 are subtasks of the main task,
manage_calendar. In Sections 4 and 9.3, these were represented as a
repeating choice. A more asynchronous design would allow an instance of
each subtask to be created in its own window. The user could go back and
forth, interleaving activity among the subtasks by activating one window after
another (e.g., by clicking in each window). The task description for this
interleaved design is:

Task: manage_calendar
(access_appointment
⇒ add_appointment
⇒ update_appointment
⇒ delete_appointment
⇒ establish_alarm) *

9.8. Concurrency

Two user actions, $\alpha_1$ and $\alpha_2$, can be concurrent if and only if it is possible
that a period of activity of one can overlap a period of activity of the other:

$$\alpha_1 \parallel \alpha_2 \iff M(\exists \pi_i(\alpha_1) \exists \pi_j(\alpha_2) (\pi_i(\alpha_1) \circ \pi_j(\alpha_2))) \tag{35}$$

where $M$ is the modal operator defined in Equation 24. In Figure 19, tasks $\alpha_1$ and
$\alpha_2$ are concurrent. A period of activity in $\alpha_1$ is overlapped by periods of
activity of $\alpha_2$.

One can derive the following property of associativity directly from the
definition of the concurrency relation in Equation 35:

$$(\alpha_1 \parallel \alpha_2) \parallel \alpha_3 = \alpha_1 \parallel (\alpha_2 \parallel \alpha_3) \tag{36}$$

This can be generalized to the ternary case:

$$(\alpha_1 \parallel \alpha_2) \parallel \alpha_3 = \alpha_1 \parallel (\alpha_2 \parallel \alpha_3) = (\alpha_1 \parallel \alpha_2 \parallel \alpha_3) = \alpha_1 \parallel \alpha_2 \parallel \alpha_3 \tag{37}$$

and to the $n$-ary case:

$$\alpha_1 \parallel \alpha_2 \parallel \ldots \parallel \alpha_n \tag{38}$$
Concurrency is a temporal relation that has not been greatly exploited in user interfaces. This may be because users are not skilled enough to carry out tasks concurrently. Norman (1988) noted that much conscious activity is both relatively slow and sequential in nature. We are able to switch attention from one task to another, or even transfer information to and from tasks. But this is interleaving, not concurrency, of action. Nevertheless, there are cases in which it is possible and, indeed, preferable, to carry out more than one task at the same time. A user can be perceptually responsive to information on the display while typing or manipulating the mouse. Buxton (1983) described input techniques that rely on the use of both hands concurrently. Such situations require the full power of the concurrency relation as described before.

Another kind of concurrency is seen in the actions of two or more users doing computer-supported cooperative work. These users, using different workstations, may be able to perform actions simultaneously on shared instances of application objects, possibly operating through different views. For a representation of the CMS in the case where periods of activity among the tasks can overlap, the task description becomes:

```
Task: manage_calendar
  (access_appointment
    add_appointment
    update_appointment
    delete_appointment
    establish_alarm)*
```
10. DISCUSSION

10.1. How the UAN Helps With Interface Development

User interface design has two separate parts: design of the user interaction and design of the corresponding user interface software. The interaction designer receives, as inputs, requirements for the design from the systems analysis process, which in turn includes inputs from marketing, task analysis, user analysis, needs analysis, and so forth. The interaction designer—who works in the behavioral domain of tasks, user actions, and perceived feedback—produces as output a behavioral design of the user interaction part of the interface. This interaction design now becomes the requirements for the user interface software designers and implementers. A precise and formal technique is needed to convey these requirements independently of the software by which the interaction is implemented. In conjunction with screen pictures showing interface objects and state diagrams showing user modes and interface states, the UAN is such a technique. Because no behavioral representation technique appropriate for documenting interaction design previously existed, current practice has been to use software objects (e.g., widgets from software toolkits) directly for interaction design representation.

At this point in the interface development process, the design is often set in a prototype, the beginning of a commitment to a software embodiment. Although the prototype is used for some kinds of formative evaluation (e.g., user testing), a behavioral representation of the design offers advantages for other kinds of evaluation. One is analytical evaluation, analysis (probably automated) of the design in search of inconsistency, ambiguity, and other undesirable characteristics. This kind of analysis is possible with the UAN, but so far it is still in the category of future work. Another important kind of evaluation at this point in the development process is a design walk-through, which typically involves designers, evaluators, and possibly implementers. Our experience has been that the UAN design representation is typically more accurate, more complete, and more precise than the prototype as a source for answers to the questions that arise in a design walk-through (e.g., about how a particular interface feature or task actually works). Also, because a prototype yields information about the design by showing examples of its operation, it is extensional or instance oriented. In contrast, a UAN representation is intensional and, therefore, explicitly states all possibilities. For example, consider a simple task that is the disjunction of tasks A and B (task: A | B). The UAN notation makes it immediately evident that there are precisely two alternatives from which to choose. Using the prototype, one might try task A and see that it works, but then possibly not realize task B had also been possible at that time (especially if the operation of task A in the
prototype makes task B subsequently unavailable). In addition, the prototype does not generally convey whether there is some other task C also available.

Although this intensional capability of the UAN is important for analytic evaluation and design walk-throughs, it is essential for conveying the behavioral design of the interaction to user interface software designers and implementers. Here, an extensional prototype simply does not suffice; solid and precise intensional specifications are a necessity.

10.2. Conclusions

As one moves from one temporal relation to another, from sequence to order independent, interleavable, and concurrent, it is in a direction of decreasing temporal constraints. The temporal nature of a sequence is quite constrained. The first action must be performed completely, then the next, and so on, until all the actions are completed. In many sequential interface designs, this constraint is arbitrary and even opposed to the cognitive and task needs of the user. For example, the initiation of a second task in the middle of a first task may be very useful in order to get information necessary for the completion of the first task. In this case, an interface design to support the user would allow the second task to be interleaved, so it does not destroy the context of the first task.

The repeating disjunction allows a very limited measure of asynchronism at a high level by allowing a choice of tasks. Once a sequence is initiated, however, no interruption, interleaving, or concurrency is allowed.

With order independence, all actions must be performed and each one completed before another is begun. But the constraint on the specific ordering among the actions is removed. Interleaving removes the constraint of being performed to completion before beginning another action, allowing an action to be interrupted.

Interleavability and concurrency are defined in different ways, but they share the fact that lifetimes of tasks can overlap. But interleavability means periods of activity cannot overlap, whereas they can in concurrency. The difference between interleaving and concurrency is something like the difference between real and apparent concurrency in an operating system (Lorin, 1972). Although multiprogramming is based on interleaving of processes, for many purposes the processes are regarded as being concurrent. Real concurrency, however, requires multiprocessing, such as is found between a central processor and an input/output processor (channel) in the hardware. Similarly, one can see the difference between interleaving and concurrency at the level of physical user actions, but perhaps not always at the level of the user's mental model of the tasks.

Analysis of the various cases using temporal relations gives the designer the
ability to distinguish task types that are significantly different but that, without these relations, would be difficult to identify. Furthermore, adding operators to the UAN to express these relations gives the designer a powerful means of representing such interfaces.

Acknowledgments. The authors acknowledge Dr. Antonio Siochi as the originator of the UAN. We also gratefully acknowledge helpful discussions with Dr. Deborah Hix, Dr. Kevin Waite, Cathy Wood, Dr. Stephen Draper, and Dr. John Urquhart. We thank the anonymous reviewers and the associate editor, Dr. Ruven Brooks, for careful thought and detailed suggestions that greatly improved the article. Dr. Marilyn Mantei originally created the CMS as an example exercise in interface design. Much mileage has been gotten from it since.

Support. The UAN was created in the Virginia Tech Dialogue Management Project during work sponsored by the Software Productivity Consortium and the Virginia Center for Innovative Technology. Partial support for our study was received by Grant Number IRI-9023333 from the National Science Foundation under the supervision of Dr. John Hestenes.

REFERENCES

Hartson, H. R., & Hix, D. (1989). Toward empirically derived methodologies and


---

**HCI Editorial Record.** First manuscript received March 1, 1990. Revisions received December 7, 1990, and May 31, 1991. Accepted by Ruven Brooks. Final manuscript received September 17, 1991. — Editor

---

**APPENDIX. MATHEMATICAL SYMBOLOGY**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>αRβ</td>
<td>α is related to β by relation R</td>
</tr>
<tr>
<td>↔</td>
<td>If and only if</td>
</tr>
<tr>
<td>∃</td>
<td>Such that</td>
</tr>
<tr>
<td>⊃</td>
<td>Such that (in set notation)</td>
</tr>
<tr>
<td>∨</td>
<td>Disjunction or logical OR</td>
</tr>
<tr>
<td>∧</td>
<td>Logical AND</td>
</tr>
<tr>
<td>¬</td>
<td>Logical NOT</td>
</tr>
<tr>
<td>⊃</td>
<td>Implies</td>
</tr>
<tr>
<td>∀</td>
<td>For all</td>
</tr>
<tr>
<td>∃</td>
<td>There exists a</td>
</tr>
<tr>
<td>∈</td>
<td>Is a member of (set)</td>
</tr>
<tr>
<td>⊂</td>
<td>Is a subset of</td>
</tr>
<tr>
<td>×</td>
<td>Cartesian product (of sets)</td>
</tr>
<tr>
<td>®</td>
<td>Maps to</td>
</tr>
<tr>
<td>⊥</td>
<td>Is a tautology (theorem) that</td>
</tr>
<tr>
<td>U</td>
<td>Set union</td>
</tr>
<tr>
<td>prim(α)</td>
<td>Boolean, true if α is primitive</td>
</tr>
<tr>
<td>A(α)</td>
<td>The action set of task α</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Can invoke</td>
</tr>
<tr>
<td>Mp</td>
<td>It is possible that p (is true)</td>
</tr>
<tr>
<td>&lt;α&gt;</td>
<td>Task α is not interruptible</td>
</tr>
<tr>
<td>•</td>
<td>Can interruptibly invoke</td>
</tr>
</tbody>
</table>

*Note.* The symbols are listed here approximately in the order of their appearance.