A Framework for Precise, Reusable Task Abstractions

H. Rex Hartson & Kevin A. Mayo
Department of Computer Science
Virginia Tech, Blacksburg, VA 24061-0106

hartson@cs.vt.edu mayo.chi@xerox.com

1. The Need for Structuring in Task Abstraction

1.1 Interface as Illusion

Direct manipulation interaction typical of today's graphical user interface (GUI) designs relies on an illusion that attempts to render application entities indistinguishable from the interface artifacts that represent them (Weller & Hartson, 1992). In this regard an interface design is successful if the user, facilitating this mimesis through suspended disbelief (Laurel, 1986), can manipulate application entities without being conscious of their interface representations. Users have the feeling of engaging real application entities. In the case of such directness, there is no difference to the user between, for instance, a file and a file icon—e.g., dropping a file icon on a folder icon is putting a file in a folder.

To a lesser extent, this kind of directness can also occur in a command-oriented interface, where the name for a file is its representative in the interface. For example, although a filename cannot be manipulated as an object by a user, users seldom think of the distinction between a file and its name.

From most perspectives of analysis and design, supporting this illusion is what interface design is about. We normally speak of interaction as though the illusion were assumed; e.g., no distinction is made between file and file icon. Interaction designers and software engineers work together to create this illusion. However, when interaction designers communicate with software designers and implementers, the reality behind the illusion comes to the surface at the articulatory boundary between interface and non-interface components. Here, designers must communicate precisely about what is behind the illusion in terms of user actions and systems actions. The precision required for this kind of communication requires a structured approach involving semantics in the task abstraction process.

1.2 Behavioral and Constructional Domains

User interface development can be differentiated into two types: *development of the interaction design* and *development of the user interface software* that implements the interaction. These development types occur in different domains, the behavioral and constructional domains, respectively (Hix & Hartson, 1993). The behavioral domain is the working world of people who design and develop interaction components of user interfaces; i.e., a user interface design is described in terms of tasks and artifacts that comprise the interaction in terms of user and interface behavior. The constructional domain involves development of GUI widgets, algorithms, programming, procedure libraries, control and data flow, state transition diagrams, event handlers, callbacks, and object-oriented representations; i.e., this domain contains the tools and techniques used to construct the interface software.

Current methods for abstraction within task descriptions are ad hoc, leading to possible imprecision, inconsistency, and ambiguity in task- and user-centered interface design representations. However, as interaction designers communicate with software designers and implementers, precise and consistent design representations become crucial.

In order to achieve precision and consistency in task abstraction we have drawn upon task structuring from the User Action Notation (Hartson, Siochi, & Hix, 1990; Hartson & Gray, 1992), the Task Mapping Model (Mayo, 1993; Mayo & Hartson, 1993) and the task vocabulary of Lenorovitz et al. (Lenorovitz, Phillips, Ardrey, & Kloster, 1984). This structure and vocabulary, we argue, can be infused with semantics associated with user actions. In turn, these semantics provide a richer description that supports precision and consistency across behavioral and constructional domains.

1.3 The User Action Notation, a Behavioral Representation Technique

The User Action Notation (UAN) was created by researchers within the Human–Computer Interaction Project at Virginia Tech as a behavioral interface design representation technique. UAN descriptions are intended to be written by interaction designers and read by all developers, particularly those designing and implementing user interface software. The UAN is a user- and task-centered notation that describes physical (and other) behavior during task performance. The primary abstraction is a *user task*; An interface is represented as a quasi-hierarchical structure of asynchronous tasks, the sequencing within each task being independent from that of others. User actions, corresponding interface feedback, and state change information are represented at the lowest level, i.e., the articulatory level, or level of physical user actions. Higher levels of abstraction hide these details and are used to build the task structure.

2. ABSTRACTION IN TASK DESCRIPTION

Describing details of user interaction can require much effort and patience. However, the effort a designer puts into detailed task descriptions is an investment that can be reused. In order to control complexity, most methods of task description use abstraction as a means to hide details of tasks at various levels (Kirwan & Ainsworth, 1992).. This is also true in the UAN where the lowest level of abstraction, the level at which details of keystrokes and mouse buttons show up, is called the articulation level. Most of a design, however, is represented at higher levels of abstraction, where this low level detail is neither visible nor necessary. Higher levels of a UAN task structure merely make reference to details via "invocations" of lower level tasks. In this way, detailed descriptions are reused in the same way a macro or procedure is reused in software.

More specifically, a task description written in the UAN is a set of user actions and tasks combined with temporal relations such as sequencing, interleaving, and concurrency, to describe time-related user behavior (Hartson, et al., 1992). Each task can then be named and the name later used as a reference to that task. A task name used as a reference is called a *macro* task because the name is used as a substitute for the whole task description. (Although this is an analogy to programming macros, one should remember that the steps in task macros are performed by *users* during task performance.) This name reference can be used as a higher level user action in another task description. This is very similar to the way the name of a software procedure is used to refer to that procedure within the code of a program; the task name is a surrogate for the task (representing a user invocation of the task), and is an abstraction, hiding task details.

In order to explore the concept of task abstraction, we first introduce a simple example in Section 2.1, the concept of task parameterization in Section 2.2, and a closer look at task semantics in Sections 3.

2.1 Example—The Task of Selecting a File

Consider the UAN description for the task of selecting a file, in Figure 1:

TASK: `select file`*	
USER ACTIONS	**INTERFACE FEEDBACK**
`~[file A] Mv`	`file A-!: file A!`
`M^`	

Figure 1. Simple UAN task description for selecting a file

* A mono-spaced font is used to distinguish UAN expressions

The ~ is used to denote cursor movement and `[` and `]` denote "context of" (`file A`)—that by which the file icon is manipulated. In this case the context is the whole icon itself. `Mv` means the mouse button is depressed and `M^` means it is released. The expression `file A-!:` is a condition, interpreted as "if file A is not already highlighted, then do the following." And that following feedback action, `file A!`, represents highlighting `file A`. This notation is defined and examples given in Hix and Hartson (Hix, et al., 1993).

Figure 1 is a good start on a task description for `select file`, but it is only a start. So far it describes only clicking on, and highlighting, a file icon. No real `file A` is yet connected to the icon. To associate this icon to a file within an application requires a connection between interface and non-interface components of the system, i.e., a connection to task semantics, a topic to which we will return in Section 3.

2.2 Parameterization for Reuse of Task Abstractions

Selection is a low-level task used frequently within interaction designs. Therefore, it is a good candidate for reuse to avoid repeating essentially the same articulatory details every time selection is needed in a design. Selection details can be encapsulated into one task description and abstracted into a name reference.

However, in the `select file` task, described in the Figure 1, the reference to `file A` is specific and "hard-wired." If task descriptions are to be reused, they must be written in a general way, using parameters to distinguish specific instances.

The `select file` task description can be generalized to describe user actions to select a single object from a set of objects having similar behavior. Hopefully, it can even be used to represent selection of several different types of entities. In order to allow this kind of object substitution in task descriptions, we must use a parameter; that is, `file A` can be made more general by making it a parameter of the task and calling it `object`. Figure 2 shows a UAN description for a more general selection task:

TASK: select(object)	
USER ACTIONS	**INTERFACE FEEDBACK**
~[object] Mv	object-!: object!
M^	

Figure 2. Parameterized task description for file selection

When another task description requires selection, for example, of a particular `File A`, the macro is invoked by the expression, `select(File A)`, binding `File A` as the `object` during instantiation of the task. This relationship is shown in Figure 3.

Generic Expression	**Invocation**
select (object)	select (file A)

Figure 3. Generic and invocation expressions for `select` task

3. SEMANTICS IN TASK ABSTRACTION—CONNECTING ILLUSION AND REALITY

A common scenario for the kind of object selection shown in Figure 2 involves these user and system actions: move cursor to object icon, click mouse button, and icon is highlighted to indicate selection. The concepts of clicking on a file icon, selecting a file, and the highlighting of an icon are usually taken to mean the same thing. But this association of meaning, in fact, must be built into the interface or it does not exist, and designers must include these semantic connections in their design representations.

To make the necessary distinction we adopt the term "entity" to refer to something within the system (e.g., file or system function) and "object" to refer to the interface representation (e.g., icon) of an entity. An entity, therefore, belongs to an application and is not visible to users; an object belongs to an interface and is visible, but illusory.

Expanding the simple UAN description of the `select` task of Figure2 to include this semantic connection between the actual file and its iconic representation yields the description in Figure 4:

TASK: select(object, entity)		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[object] Mv	object-!: object!	selected={entity}
M^		

Figure 4. `Select` task description with parameters for semantic connection to file entity

The difference in this description is the inclusion of the third column which indicates a state change, namely the `entity` becoming selected. (In the third column, `selected` is the name of a set that contains the names of the `entities` currently marked as selected.) The `object` is highlighted—a visible sign that the `entity` has undergone a state change. It is clear that `entity` and `object` referring to it are different but associated. This is an example of a situation where invisible semantics (`file` is selected) are made visible through the syntax of highlighting. Here, "syntax" is taken in the broadest sense, to include what is perceivable in the interface by the user.

This distinction between `object` and corresponding `entity` introduces additional complexity into the `select` task description. When we parameterize the `select` task for reuse, we must include parameters to indicate which interface object is pointed to and which entity is selected.

The reusable task description in Figure 4 contains two items that must be bound through parameters to items in the invoking world—`entity` and `object.` The former can be a file, a system function, etc. The latter can be an icon, button, scroll bar, etc. Reference to the task becomes: `select(object, entity` and an invocation to select a specific file, say `file A,` would become: `select(file A icon, file A)`. This relationship is shown in Figure 5.

Generic Expression	**Invocation**
select (object, entity)	select (file A icon, file A)

Figure 5. Generic and invocation expressions for `select` task

This interpretation of `select` crosses the interface/non-interface boundary of a software system because it connects the behavioral act of selecting an object in the interface to the constructional state of an entity being selected in the non-interface system component. As we said at the end of this Section 1.1, if user interface designers are to communicate their designs to software designers and implementers with precision, addressing details is obligatory, especially at this cross-boundary connection where reality of the illusion comes to the surface as implementation.

Usability issues also arise at this interface/non-interface boundary. Designs that do not give good selection feedback can lead to a task situation where the system "thinks" the user has caused a state change but the user doesn't, or vice versa; state knowledge is not communicated. As examples, consider the lack of feedback in command-line interfaces such as DOS or UNIX.

Our discussion of selection has mostly been oriented toward files and file icons. We want to be sure we can generalize to include selection with other interface objects, e.g., buttons, scroll bars, etc. For example, one would expect our `select` task macro be reusable for selection of a system function (e.g., with a command push button labeled `Open`). The concept of illusion, as discussed in Section 1.1, should apply to buttons as well as file icons, and it does. However, the semantics are a little different. Clicking on a file icon causes a file to be selected, but clicking on the `Open button` activates a system function to open files that are already selected.

Thus, command button selection doesn't appear to need the second parameter and would have a different semantics in the `Interface State` column of the task description.

In Sections 2 and 3 we encountered the possibility of ambiguity, imprecision, and inconsistency in current *ad hoc* task abstraction processes. Even cases which were believed to be simple and well understood, such as the `select` task, were shown to be problematic. We have argued for inclusion of semantics in task abstraction, but we also need a framework and vocabulary for abstractions.

4. FRAMEWORK FOR ABSTRACTION

In order to provide a framework within which to structure abstraction, we adopted the framework of the Task Mapping Model (TMM). Much like other modeling techniques such as CLG (Moran, 1981), ETAG (Tauber, 1990), etc., TMM depicts user task performance in terms of a series of mappings from the user's problem domain (the world in which the user wishes to use a computer to solve a problem) to the computer semantic domain (the world of files and application functions) to the computer syntactic (the world of commands and grammatical elements) to the articulatory domain (the world of keystrokes, cursor movement, and button presses). In order to use a computer to perform a problem domain task, the user must make a series of mappings to translate the task through this series of domains. We believe that knowledge necessary to make the mappings, but lacking in users and not supported by interface designs, accounts for a large proportion of usability problems.

As an example, consider the problem domain of document writing, and consider the task, in this domain, of discarding an existing document. Vocabulary used to describe tasks varies from domain to domain. In this problem domain, the task might be called `discard manuscript`.

In the computer semantic domain, this same task is expressed using computer semantic entities and application functions. The document `manuscript` from the problem domain is represented as a `file` entity in the computer semantic domain, because documents are stored in files. For the moment, almost any general term, such as `eliminate`, can be used to name the function for discarding files. Thus, `discard manuscript` in the problem domain maps into the unordered pair {`eliminate`, `file`} in the computer semantic domain. The use of set braces ({,}) denotes the fact that the pair is unordered.

The `function-entity` pairs of the computer semantic domain take on a command-like appearance in the computer syntactic domain, referring to specific interface objects, possibly with grammatical ordering.

Going to higher levels of abstraction from physical actions in the UAN corresponds to moving from articulatory to computer syntactic to computer semantic to problem domain in the TMM. Thus, the TMM domains give us guidance for establishing structured levels of abstraction for the UAN, which are summarized in Figure 6. In this figure, "grammar" refers to general syntactic aspects of expressions within task descriptions, including ordering of terms and design-, interaction style-, or implementation-dependent terminology. "Style" refers to specific interaction style or technique, such as form-filling, direct manipulation, or command line interaction.

We will refer to the highest UAN levels as *semantic levels* of abstraction. A typical interface design expressed in the UAN will have many levels of semantic abstraction, each higher level abstracting more and more detail into its task macros. At every semantic level the general form for a task macro is an unordered pair, {`function`, `entity`}, where `function` denotes an internal application system function (e.g., a procedure for deleting) and `entity` denotes an internal application system entity,(e.g., file). The semantic levels are independent of how the functions and entities are represented within an interface design. Grammar, user actions, and interaction style are all abstracted out of task descriptions at semantic levels. UAN expressions at any level have always corresponded to user actions. At semantic levels UAN expressions are task names (invoking the user actions of the tasks) connected by temporal relations. In the context of this paper, UAN semantic level user actions are task names expressed as {`function`, `entity`} pairs (e.g., {`edit form field`}, {`open file`}).

Below the semantic levels are the syntactic levels of abstraction. We shall distinguish three kinds of syntactic levels—grammar, grammar-action, and grammar-action-style levels. At the grammar level, the unordered {`function`, `entity`} pairs of the semantic levels appear as ordered `function-entity` (or `entity-function`) pairs. Grammatical ordering has been added, so that some constraints and implications about specific interface designs are already evident, but action and style are still abstract.

The border between the grammar and grammar-action syntactic levels is significant because above this line are only entities and functions and below the line are only user actions used to communicate requests for the entities and functions.

The grammar-action level includes abstract user actions used to request the `function`, `entity` pairs of the grammar syntactic level, but the expressions are still independent of specific interaction style (e.g., are not specific to menus, forms, buttons, command lines).

Levels	Sub-Level	Attributes	Expressions
Semantic Levels		Unordered pairs of functions and entities— *Grammar Abstracted* *Action Abstracted* *Style Abstracted*	{function, entity}
Syntactic Levels	Grammar (G) Syntactic Level	Ordered pairs of functions and entities— *Grammar Specific* *Action Abstracted* *Style Abstracted*	(function, entity) *—or—* (entity, function)
	Grammar-Action (G-A) Syntactic Level	Ordered pairs of functions and entities with actions defined to communicate the pairs— *Grammar Specific* *Action Specific* *Style Abstracted*	Actions to communicate function, entity *—or—* Actions to communicate entity, function
	Grammar-Action-Style (G-A-S) Syntactic Level	Ordered pairs of commands and objects with style specific actions defined to communicate the pairs— *Grammar Specific* *Action Specific* *Style Specific*	Style specific Actions to communicate function, entity *—or—* Style specific Actions to communicate entity, function
Articulatory Level		Physical user actions necessary to communicate the G-A-S level expressions to the computer	*Written in terms of keystrokes and cursor movement*

Figure 6. Structured levels of task abstraction

The grammar-action-style level is the lowest syntactic abstraction level, the level just above the articulatory level. Abstractions at this level are of particular interest. This is because macros at this level are reused the most throughout designs, much of the interface design activity occurs here, and because issues found here (such as those concerning parameterization) have high impact throughout the rest of the design. The grammar-action-style level will be discussed more in Section 6.5.

The lowest level of abstraction in the UAN is the articulatory level. The articulatory level is where tasks are described in terms of specific physical user actions required to carry out these functions on a particular implementation platform—e.g., keystrokes of a command line interaction (CLI) or entity selection and menu choices or button presses of a direct manipulation GUI style interaction. The UAN task descriptions in Figures 1, 2, and 4 occur in the articulatory level.

5. VOCABULARY FOR ABSTRACTION

By and large, creating task abstractions for reuse is a matter of factoring out similar sequences that recur over task descriptions. The question is: How does one determine what is similar? We have been looking mostly at articulatory details, which are very much design dependent. By giving two tasks a similar articulatory design, one can make them look similar, but they may in fact be different semantically. To overcome this difficulty, we sought a precise and consistent vocabulary, so that the appearance at syntactic levels (terms used to describe tasks and their abstractions) will reflect distinctions at semantic levels.

5.1 Taxonomy of User Actions by Lenorovitz et al.

Our need for consistent naming implies the need for a vocabulary, defined in terms of semantics, for identifying elements of abstractions, especially those occurring just above the articulatory level. The user-system interface action taxonomy by Lenorovitz et al. (Lenorovitz, et al., 1984), which is built upon the work of Berliner, et al. (Berliner, Angell, & Shearer, 1964), provides a vocabulary that can be used for precise and consistent naming of certain common user tasks, based on semantic distinctions. We will to use the term “function” for operations that are performed by the computer (usually at the request of the user—e.g., deleting a file) and the term “action” for operations that are performed by the user (e.g., selecting a file).

Figure 7 shows the user-input function/action taxonomy from Lenorovitz et al., adapted here to include their definitions (which contain much of the semantics).

CREATE	ASSOCIATE	NAME	Give title to or attach label to for purposes of identification/reference.
		GROUP	Link together or associate for purposes of identification.
	INTRODUCE	INSERT	Make space for and place an entity at a selected location within the bounds of another such that the latter wholly encompasses the former, and the former becomes an integral component of the latter.
	ASSEMBLE	AGGREGATE	Combine two or more components so as to form a new composite entity.
		OVERLAY	Superimpose one entity on top of another so as to affect a composite appearance while still retaining the separability of each component layer.
	REPLICATE	COPY	Reproduce one or more duplicated of an entity (no links to *master*).
		INSTANCE	Reproduce an original (*master*) entity in such a way as to retain a definitional link to the master—i.e., such that any subsequent changes or modifications made to the master will automatically be reflected in each and every *instance* created therefrom.
INDICATE	SELECT (*POS/OBJ*)		Opt for or choose an entity (e.g., a position or an object) by *pointing* to it.
	REFERENCE		Opt for or choose an entity by invoking its name.
ELIMINATE	REMOVE	CUT	Remove a designated portion of an entity and place it in a special purpose buffer (residual components of the original entity usually close in around *hole* left by *cut-out* portion).
		DELETE	Remove and (irrevocably) destroy a designated portion of an entity.
	STOP	SUSPEND	Stop a process and temporarily hold in abeyance for future restoration.
		TERMINATE	Conclude a process such that it cannot be restarted from the point of interruption, only by complete re-initiation.
	DISASSOCIATE	RENAME	Change an entity’s title or label, without changing the entity itself.
		UN-GROUP	Eliminate the common bond or reference linkage of a group of entities.
	DISASSEMBLE	SEGREGATE	Partition and separate an entity into one or more component parts such that the structure and identity of the original is lost.
		FILTER	Selectively eliminate one or more layers of an overlayed composite.
		SUPPRESS	Conceal or keep back certain aspects or products of a process without affecting the process itself (i.e., affects appearance only).
		SET-ASIDE	Remove entire contents of current (active) work area and store in a readily accessible buffer (for future recall).
MANIPULATE	TRANSFORM		Manipulate or change one or more of an entity’s attributes (e.g., color, line type, character font, size, orientation) without changing the essential content of the entity itself.
ACTIVATE	EXECUTE __ *FN*.		Initiate or activate any of a set of predefined utility or special purpose functions (e.g., sort, merge, calculate, update, extract, search, replace).

Figure 7. The Lenorovitz et al. user-input taxonomy with definitions

For our example task, `select(object, entity)`, we need look at only the parts of this table that include the `indicate` and `activate` action types ; see Figure 8.

INDICATE	SELECT *(POS/OBJ)*	Opt for or choose an entity by *pointing* to its interface representation.
	REFERENCE	Opt for or choose an entity by invoking its name.
ACTIVATE	EXECUTE __ *FN*.	Initiate or activate any of a set of predefined utility or special purpose functions (e.g., sort, merge, calculate, update, extract, search, replace).

Figure 8. Extract from taxonomy for `indicate` and `activate`

In Figure 8 both `select` and `reference` are ways to `indicate`—`select` by pointing and `reference` by referring to a name.

5.2 Clarifying Selection by Pointing

The Lenorovitz et al. definition of `select` is, as shown in Figure 8 (with some terminology changes to be consistent with ours): "Opt for or choose an entity by *pointing'* to its interface representation." For example, moving the cursor to a `file icon` with a mouse, and clicking the mouse button is an instance of `indicate` by `select`. This definition of `select` needs augmentation for several reasons.

First, as discussed in Section 1.1, an entity being selected is only represented in the interface by the object. Pointing to the object causes an internal state change in an attribute of the corresponding entity representing selection.

Second, the basic underlying design of an interface/system will define the semantics for "pointing to." For example, in the case of Macintosh OS, the point of the arrow cursor (sometimes known as the hotpoint of the cursor icon) must enter the object. In other possible platforms, any part of the cursor may enter the object to constitute "pointing to."

Third, although cursor movement in GUI interfaces is usually accomplished via mouse, trackball, touchscreen, or similar input device, our interpretation is broad enough to include discrete cursor movement from object to object by such means as arrow keys or tab/return keys.

Finally, since specific designs for `select` by pointing often require an accompanying action (e.g., depressing or releasing or both of a mouse or trackball button), we extend the definition of `select` accordingly in Figure 9, including the addition of possible feedback.

INDICATE	SELECT *(POS/OBJ)*	Opt for or choose an entity by pointing to the object that represents the entity in the interface, with the possible addition of a confirming action. Selection causes an internal and invisible state change to an attribute of that entity, along with possible feedback denoting that internal state change.

Figure 9. Augmented taxonomy entry for `select` action

The definition now accounts for a set of possible user action sequences for selection. Figure 10 illustrates other actions that meet the definition of Figure 9 and that could be used to accomplish selection, depending on the paradigm of the interface style.

In a rich environment, say OSF Motif™, there are several more ways to select an entity or group of entities (OSF, 1991). Additionally, all of these possibilities could be in combination with shift, control, and/or command keys. Although definitions in Figures 8 and 9 refer to "an entity", multiple selection expands the possibilities even more. Further, it is clear that deselection is also a similar user input action, but not mentioned in the taxonomy. Thus, it becomes easier to see how complex even a selection action can be, in spite of its being a common user action that is often regarded as being simple.

Select Method (UAN)	Notes	Example
`~[X]`	Move Cursor to the context of object X	Select a Menu Name using TAB
`~[X]Mv`	Move cursor to X and depress mouse button	Select a icon representing a file
`~[X]M^`	Move cursor to X and release mouse button	Select "Delete" operation by "dropping" file icon into trashcan icon
`~[X]Mv^`	Move cursor to X and click mouse button	Select a button
`~[X]Mv^Mv^`	Move cursor to X and double click mouse button	Selecting launch operation for an application
`~[X]Mv` `~[Y]M^`	Move cursor to X and depress mouse button, then move cursor to Y and release mouse button	Select a group of file icons using "rubber-band" selection

Figure 10. Some possible user action sequences for selection

Regardless of these variations, the definition of `indicate` by `select` is distinct from `indicate` by `reference`. To `reference` is to "Opt for or choose an entity by invoking its name," as in Figure 8.

6. APPLYING THE FRAMEWORK AND TAXONOMY

We have argued for a more structured approach to task abstraction. To that end, we have introduced a framework and a vocabulary to support task abstraction. We have also argued for incorporating semantics into our task abstractions for precision and consistency. We now turn our attention putting the framework and taxonomy to work in the process of abstraction, in the context of UAN task descriptions.

6.1 Semantic Levels of Abstraction

Large parts of most non-trivial interface designs are represented at semantic levels, as {`function`, `entity`} pairs, as mentioned in Section 4. For example, at the highest level of abstraction for a word processing system, one would find the single most abstract task description: {`manage`, `documents`} and so on, down to much more specific tasks, such as {`open`, `file`}. Tasks that require files to be opened will invoke the {`open`, `file`} task name rather than spell out its details. In this way, abstraction is used to build a quasi-hierarchical structure of higher and higher level tasks.

In Figure 11 we have illustrated the various levels of abstraction. We refer to this figure throughout the rest of the paper. Box A in Figure 11 represents the generic form of a task description in the semantic level. Box B contains an instance of a task at this level.

6.2 Grammar Syntactic Levels of Abstraction

The main difference in going from semantic levels to grammar syntactic levels is that the `function-entity` (or `entity-function`) pairs become ordered; see box C in Figure 11.

Using a linguistic analogy, the semantic {`function, entity`} pairs are canonical, or deep structure, representations of intended operations with no grammatical ordering or other information about how the requests to the system are expressed. In contrast, expressions found in syntactic levels are surface structure or observable representations embodying grammatical order and possibly other interaction style and design dependent information about how user requests are expressed. It is often the case that several different surface structures correspond to the same underlying semantic deep structure.

In a syntactic view of either CLI or GUI designs, it is usual to think of "commands" in the simplest grammatical terms. For example, we often classify the basic interaction paradigm as having either Verb-Noun (**V-N**) or Noun-Verb (**N-V**) form. In a GUI selecting a file followed by executing an `open` function is considered to be in the **N-V** form. However, in a command line interface the command vi <filename> is considered to be in the **V-N** form.

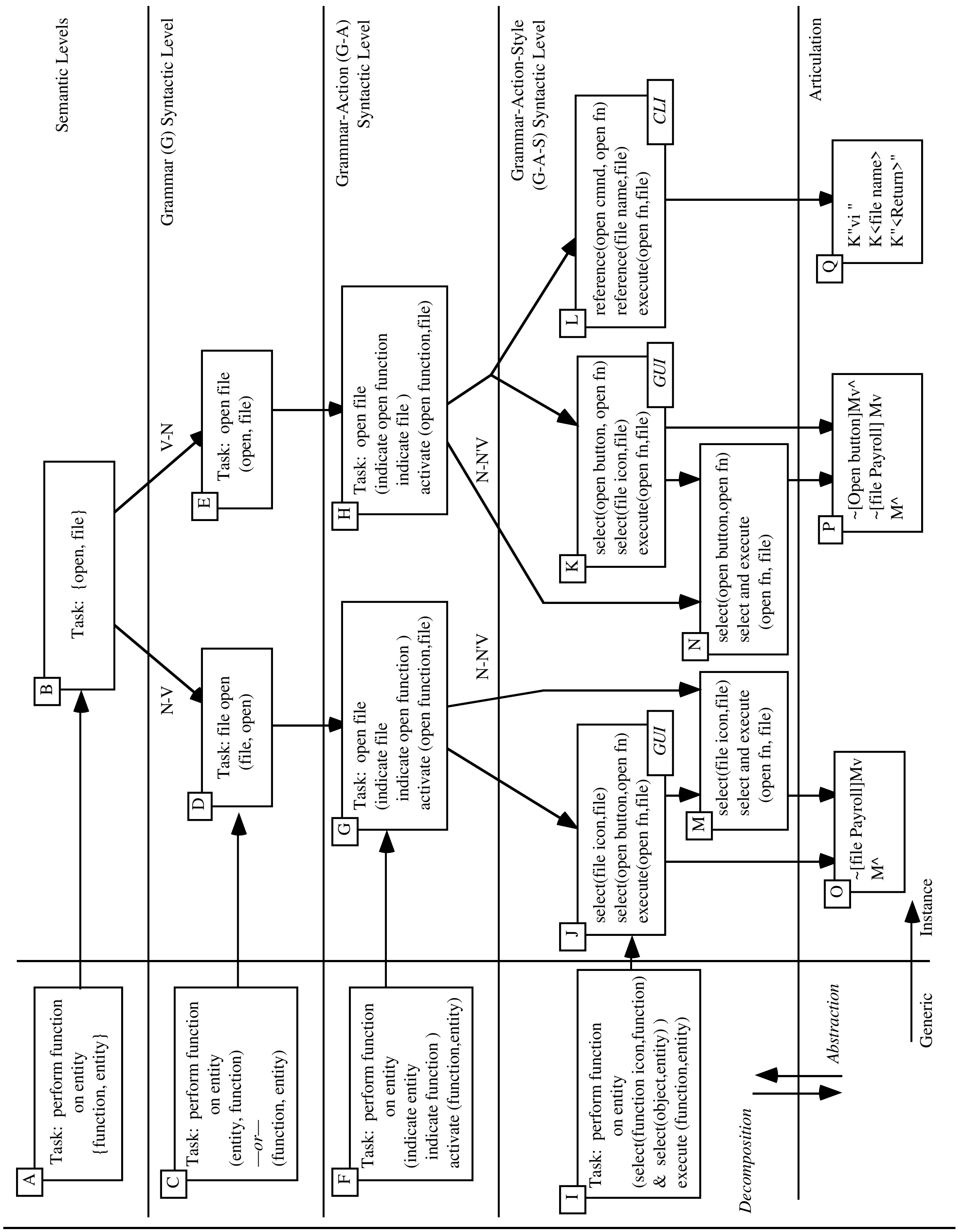

Figure 11. Overall view of abstraction structure

Thus, for the {open, file} function-entity pair in a semantic level, a task description in the grammar syntactic level with grammatical ordering that mentions the open function first would be of the form **V-N**, as in box E of Figure 11. A grammatical ordering that mentions the file entity first would be of the form **N-V**, as in box D.

6.4 Grammar-Action Syntactic Level of Abstraction

As pointed out in Section 4, above the grammar syntactic level, one sees functions and entities. In contrast, at the grammar-action level and below, one sees user actions requesting functions applied to entities. To represent the function, entity task of the grammar syntactic level (box C of Figure 11), we use the taxonomic terms indicate and activate at the grammar-action syntactic level (boxes G and H).

6.5 Grammar-Action-Style Syntactic Level of Abstraction

As stated earlier, in Section 4, the grammar-action-style syntactic level of abstraction is important. The articulatory level contains the details. Abstraction is a bottom-up process and the first level of abstraction occurs at the first level above the articulatory level, namely at the grammar-action-style syntactic level. This first level of abstraction is the level at which designers spend much of their time and effort. Second, this level of abstraction is where the illusion of the interface breaks down and reality behind the illusion must become explicit.

Designers or management often make early decisions on interaction style that determine look and feel of the product. This establishes many of the physical actions (e.g., keystrokes and mouse movements) for the lowest level tasks (e.g., button and menu selection). Once the details of these physical actions are known and agreed upon, designers tend to ignore them by abstracting them away, working only at higher levels. For example, a company designing a security system for banks and airports filled a large wall with a UAN description of their task hierarchy, but articulatory details were never explicitly expressed. The first level of abstraction above the articulatory level (the grammar-action-style syntactic level) was the lowest (most detailed) level they used in their thinking and design discussions. They had already decided on a specific interaction style and knew well the expressions and idioms at that level.

Next, consider the question of the illusion (discussed in Section 1.1) breaking down. Although the direct manipulation interaction style makes it seem to the user as though he or she is performing functions directly on application entities, the reality is that the icons, buttons, and menu choices are really commands and parameters in requests to have the system perform the functions. These user requests connect the illusion to reality and the grammar-action-style syntactic level to levels below and above. This realization is crucial in understanding the grammar-action-style syntactic level of interaction. The reality behind the illusion is that mutual task performance is a case of the user communicating to the system what is to be done and the system doing it.

A possible first level (the grammar-action-style level) syntax for the open file task of box G in Figure 11 is shown in box K. An alternative grammar-action-style syntax for the open file task, this time derived from box H and one that implies a command line interaction style instead of point and click, is shown in box L.

These grammar-action-style syntactic level task descriptions in boxes J, K, and L all satisfy grammar-action syntactic descriptions given in boxes G and H. The version in box L however, leads to quite a different description at the articulatory level, as shown in box Q. Box Q contains command syntax one might type to use the Unix "vi" editor to open a file, where K refers to the keyboard string generating device in the UAN, "vi" is a literal string, file name is a variable string, and <Return> is the carriage return key, a single keystroke.

The grammar-action-style syntactic level task description in box J of Figure 11 represents a typical GUI design for the open file task, one in which the file is selected before the open function. Box K contains a symmetric description, selecting the open function first. In these symmetric designs, the function (open) and the entity (file) are both represented with icons (or other GUI interface artifacts). A third interface object (e.g., button) is required to represent a request for an action to execute the function on the entity (i.e., a Do It button). After selecting an entity and a function (the order depending on whether box J or box K is used), the user clicks on the Do It button to execute the function on the entity. These symmetric designs show how select and `execute` (instances of `indicate` and `activate` in the taxonomy) actions are used to request the {`Open, file`} `function-entity` pair.*

In terms of designing for usability, though, we realize that this sequence of three separate actions is extra work for the user, and a typical real GUI design combines the selection of the `open function`, in box J, with its application to the `file`. The result is a *task contraction* (Hartson & Mayo, 1994) used to combine the second `select` with the `execute(Do It)` action. A task contraction embeds the lexical form of one task within another, usually sharing a terminating action. The contraction of the task in box J is shown in box M of Figure 11.

Recalling the discussion of **N-V** vs. **V-N** forms of the grammar syntactic level in Section 6.2, it is interesting to explore this same concept at the grammar-action-style syntactic level. Boxes J and K have neither of these forms. The grammatical form of a `select` action is **N** (because the thing being selected—either entity or function is a noun) and the grammatical form of the `execute` is **V** (because it is the active application of the function). Thus, in either order (box J or K), the form of the task is: **N-N-V**. Also, at first glance, the task contraction in box M appears to be of the **N-V** form, but, as a contraction of **N-N-V**, this form is: **N-N'V**.

The discussion in this section has revealed the answer to a problem that has continued to escape solution through most of this paper until now. The intuition that a command button is different from a file icon because it led to an immediate execution of a function is not quite right. For the grammar syntactic level **N-V** case at (e.g., box D), the grammar-action-style syntactic level task contraction (e.g., box M) made it appear that selection of the function led to immediate execution. However, we now see that the description in box M is a contraction of three separate actions. What we learn from this is that, at the grammar-action-style syntactic level all buttons are similar and, even to `execute` a `function`, the user must first `indicate` which `function` to `execute`.

6.6 Articulatory Level of Abstraction

Let's continue with our GUI example for opening a file and move to the articulatory level of abstraction. Consider the task arena of the Open (file) dialogue box in, say, Microsoft Word. Within this arena consider the specific task of selecting a file by clicking on its name from the scrolled list. We assume the desired file name is visible, a condition of activation that might require other tasks to actualize) and activating the command (via the Open push button) to open the selected file. One way (task method) of doing this task, as expressed in the UAN at the articulatory level, is shown in Figure 12:

TASK: open file (file)		**Arena:** Open dbx
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
file name visible in scrolled list:		
~[file icon A] Mv	file icon A-!: file icon A!	selected={file A}
M^		
~[Open button] Mv	Open button!! (a second kind highlighting, i.e. blinks)	
M^		Open function is invoked on the selected file
		Selected file is open

Figure 12. A UAN description for the `open file` task

* Here only button, menus, and commands are considered. Other direct manipulation actions (such as resizing a box) are beyond the scope of this paper but are being considered elsewhere [ref the working paper extension of this].

7. THE RELATIONSHIP BETWEEN OBJECT-ORIENTED AND BEHAVIORAL VIEWS

In the context of interaction representation, the question arises of whether an object-oriented view, which represents interface object behavior, is a kind of behavioral representation. The answer is no, an object-oriented representation cannot by itself be considered a behavioral representation technique, because it lacks the essential task orientation. However, it does serve as a useful complement to the task view when it comes to representing system feedback.

As we have seen in Section 6, most user actions in a UAN task description at the syntactic level of abstraction can be interpreted as an {$\mathtt{action}_i$, $\mathtt{object}_j$} pair, where $\mathtt{object}_j$ is an interface artifact. In the feedback column, the corresponding feedback, $\mathtt{fb}_{ij}$, is defined. If the interface artifacts exist in an object-oriented environment, when the designer specifies this feedback, $\mathtt{fb}_{ij}$, it becomes part of the definition of $\mathtt{object}_j$. Namely, it establishes the response of $\mathtt{object}_j$ to $\mathtt{action}_i$ (which invokes $\mathtt{method}_i$ of $\mathtt{object}_j$). In this sense the UAN crosses over the behavioral boundary into the constructional world at the articulatory level.

If {$\mathtt{action}_i$, $\mathtt{object}_j$} is encountered again in a task anywhere in the interface design, $\mathtt{fb}_{ij}$ is already defined and cannot be changed for this new instance (unless a decision is made to change it for all instances, or unless the $\mathtt{object}_j$ feedback for this $\mathtt{action}_i$ is also a function of some other state variable).

A task description, a set of these {$\mathtt{action}_i$, $\mathtt{object}_j$} pairs, connected with temporal relations, shows user behavior for an entire task. However, it shows only part of an object's behavior. A "trace of execution" within a task description might "pass through" an $\mathtt{object}_j$ just once, say, when $\mathtt{action}_i$ invokes $\mathtt{method}_i$. Similarly, in an object-oriented view, we see the entire behavior of an object, and only glimpses of user tasks as they "pass through," occasionally applying actions to the object.

A complete definition of an object is accumulated as more and more tasks are described, revealing the feedback specification for more and more actions upon the object. The sum of all the task descriptions in the interface design yields all the object designs, determining how many and which actions (or methods) are required of each object. In this sense, the feedback column of the UAN directly feeds object-oriented design of interface artifacts. In turn, an object-oriented view of interface artifacts provides a constructional representation, complementary to the UAN.

8. CONCLUSIONS

This paper is about modeling human-computer interaction for better understanding of interaction, leading to better designs and design representations. We argue for a structured approach to task abstractions that includes semantics, as well as a framework and vocabulary for abstraction. Our approach imposes a notational consistency that aids designer-implementer communication. Including semantic considerations allows designers to specify interaction so that implementers can properly support the connection between illusion and reality.

ACKNOWLEDGMENTS

We wish to thank Jeff Brandenburg, Joe Chase, Deborah Hix, and William Wake for many discussions on this and related topics down in the HCI brainstem-storming room in Femoyer Hall.

REFERENCES

Berliner, D. C., Angell, D., & Shearer, J. W. (1964). Behaviors, Measures, and Instruments for Performance Evaluation in Simulated Environments. *Proceedings of Symposium and Workshop on the Quantification of Human Performance*.

Hartson, H. R. & Gray, P. D. (1992). Temporal Aspects of Tasks in the User Action Notation. *Human-Computer Interaction*, *7*, 1-45.

Hartson, H. R. & Mayo, K. A. (1994). Languages for Task Abstraction. *Work in Progress.*

Hartson, H. R., Siochi, A. C., & Hix, D. (1990). The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs. *ACM Transactions on Information Systems*, *8*(3), 181-203.

Hix, D. & Hartson, H. R. (1993). *Developing User Interfaces: Ensuring Usability Through Product and Process*. New York: John Wiley & Sons, Inc.

Kirwan, B. & Ainsworth, L. K. (Ed.). (1992). *A Guide to Task Analysis. Volume* London, England: Taylor & Francis.

Laurel, B. K. (1986). Interface as mimesis. Chapter 4 In D. A. Norman & S. W. Draper (Ed.), *User Centered System Design* (pp. 67-85). Hillsdale, NJ: Erlbaum.

Lenorovitz, D. R., Phillips, M. D., Ardrey, R. S., & Kloster, G. V. (1984). A Taxonomic Approach to Characterize Human-Computer Interfaces. In G. Salvendy (Ed.), *Human-Computer Interaction* (pp. 111-116). Amsterdam: Elsevier Science Publishers B. V.

Mayo, K. A. (1993). *Task Mapping Model (TMM) Analysis Manual: A Reference* (TR-93-07). Department of Computer Science, Virginia Tech (VPI&SU), Blacksburg, Virginia.

Mayo, K. A. & Hartson, H. R. (1993). Synthesis-Oriented Situational Analysis in User Interface Design. *Proceedings of The 1993 East-West International Conference on Human-Computer Interaction*, Moscow, Russia, August 3-6: ACM, 134-150, V. II. Selected for reprint in *Lecture Notes in Computer Science.* V. 753. Bass, L. J., Gornostaev, J., and Unger, C. (Editors). Berlin: Springer-Verlag, pp. 53-67.

Moran, T. P. (1981). The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems. *International Journal of Man-Machine Studies*, *15*, 3-51.

OSF. (1991). *Open Software Foundation (OSF)/Motif™ Style Guide: Revision 1.1 (For OSF/Motif Release 1.1)*. Englewood Cliffs, New Jersey: PTR Prentice Hall.

Tauber, M. J. (1990). ETAG: Extended Task Action Grammar—A Language for the Description of the User's Task Language. *Proceedings of IFIP Conference on Human-Computer Interaction—INTERACT'90*, Cambridge, U.K., August 27-31: Elsevier Science Publishers B.V. (North-Holland), 163-168.

Weller, H. G. & Hartson, H. R. (1992). Metaphors for the Nature of Human-Computer Interaction in an Empowering Environment: Interaction Style Influences the Manner of Human Accomplishment. *Computers in Human Behavior*, *8*, 313-333.