"Generating" Design Spaces: An NLP Approach to HCI Design

Margherita Antonia
Demosthenes Akoumianakis
Constantine Stephanidis

This chapter proposes a grammar-based approach to the population of design spaces in the context of computer-based interactive systems. This new approach exhibits properties of multiple-metaphor environments, and allows the generation of alternative design specifications, combining elements from different toolkits, on the basis of task models. It builds on the one hand, on the frequently occurring parallelism between user interfaces and sign systems, and on the other hand, on concepts and techniques recently elaborated in the field of Natural Language Processing, and in particular in the Head-Driven Phrase Structure Grammar theory. The grammar constitutes a simple and effective instrument for classifying and structuring the complex knowledge underlying the design of multiple-metaphor environments, as well as of capturing a level of abstraction suitable for a principled ‘generation’ of design spaces, and can be integrated in design tools for specification-based development environments.

The notion of metaphor in interface design is increasingly becoming a critical aspect in the attempt to provide more effective and higher quality interaction between humans and artifacts. In the past, the predominant metaphor for computer-based applications has been that of the office environment, which was visually encapsulated as the desktop in various graphical user interface (GUI) toolkits (e.g., Windows95, OSF Motif, Athena Widget Set). The Rooms (Henderson & Card, 1986) and the Book (Moll-Carrillo, Salomon, March, Fulton Suri, & Spreebner, 1995) metaphors are examples of alternative embodiments of real-world metaphors into a user interface. With the advent of the World Wide Web, the conventional visual desktop has been enriched with concepts from hypermedia, such as links, browsing, and so on. In the future, it is expected that neither the conventional desktop, nor the currently prevalent Web metaphors will provide adequate solutions to designing for the broadest possible end-user population and the variety of contexts of use (Stephanidis et al., 1998). In fact, the number and diversity of application domains in which the use of metaphors is critical continuously increase (examples include education software, home-based interaction environments, virtual and augmented realities, and electronic health care records).
Whereas, in the past, the use of metaphors was at the discretion of the designer, or in the best case, bound to what the existing development toolkit offered (e.g., trashcans, form filling), today, embedding metaphors to interface design is compelling for the wide adoption and user acceptance of (the) applications, in particular for nontraditional/nonbusiness applications such as the ones mentioned previously.

Metaphors may be applied at various levels, ranging from the overall interactive environment offered by an application, to the task level (i.e., how users engage and perform specific goal-oriented activities), to the physical level of interaction (i.e., representations used to convey intended meaning). Each of those levels may not involve the articulation of the same real-world metaphor, but variants of different ones (Stephanidis & Akoumianakis, in press). Thus, at the level of the overall interactive environment, users may be exposed to a booklike (Moll-Carrillo et al., 1995) or roomslike (Henderson & Card, 1986) metaphor, whereas in order to accomplish specific tasks (such as, e.g., the deletion of a file) alternative metaphors (e.g., deleting a file from a folder) may be used. Progressively, interactive computer-based applications move toward a state that can be characterized as multiple-metaphor environments (Akoumianakis, Savidis, & Stephanidis, 2000; Stephanidis & Akoumianakis, 1999; see also chap. 13, this volume).

In this perspective, human–computer interaction (HCI) design is anticipated to become a much more complex activity than in the case of single-metaphor design, and to require appropriate supporting tools (Stephanidis & Akoumianakis, in press). Design tools will be required, among other things, to facilitate designers in the population and exploration of the design space of multiple-metaphor environments, that is, the (sets of) design alternatives, or possible combinations of interface artifacts, integrating elements from different toolkits, which in turn may implement potentially different interaction metaphors. Furthermore, design tools will be required to facilitate designers in shaping such design spaces in a principled and coherent way on the basis of the user’s characteristics, abilities, and preferences, the context of use, and the input/output modalities required or recommended as suitable to a given task. Population of design spaces and support for multiple metaphors are important aspects of designing user interfaces for all (see chap. 21, this volume).

Taking into account the preceding requirements, this chapter proposes a grammar-based approach to the population of design spaces in multiple-metaphor environments, which allows the generation of alternative design specifications on the basis of task models. Such an approach builds, on the one hand, on the frequently occurring parallelism between user interfaces and sign systems (Goguen, 1998; see also chap. 13, this volume), and on the other hand, on concepts and techniques recently elaborated in the field of natural-language processing (NLP), and in particular in unification-based approaches to natural language. The grammar is claimed to provide a simple and effective instrument for classifying and structuring the complex knowledge underlying the design of multiple-metaphor environments, as well as of capturing a level of abstraction suitable for a principled
“generation” of design spaces. The approach described in this chapter relates to
the notion of user interface in the context of user interfaces for all, because it pro-
vides support for informed choice and principled combination of metaphor(s) in
multiple-metaphor environments (see also chap. 21, this volume).

This chapter is organized as follows. The next section introduces issues related
to user interfaces as sign systems, whereas the third section briefly outlines the
main characteristics of one of the more recent and widely applied language theo-
ries of the unification-based family, namely head-driven phrase structure gram-
mar—HP SG (Pollard & Sag, 1987, 1994) and of its underlying formalism, namely
typed feature structures—TFS (e.g., Carpenter, 1992; Carpenter & Penn, 1998).
The same section also briefly describes how unification-based grammars can be
used for the generation of text from abstract representations of meaning. Subse-
sequently, the fourth section discusses the population of multiple-metaphor design
spaces in HCI as a semantic-driven generation process relying on principles and
mechanisms of the previously mentioned NLP approaches. The final section pres-
ents a discussion on the proposed approach.

THE USER INTERFACE AS A SIGN

User interfaces are often characterized in HCI literature as signs in a structuralist
sense, that is, as associations of perceivable forms with some type of meaning
(Goguen, 1998; see also chap. 13, this volume). Forms (or presentations) of user
interface signs correspond to elements in a presentation domain, such as the
(graphical representation of) interaction objects provided by conventional toolkits
for the design and implementation of GUIs. Meaning, on the other hand, corre-
sponds to elements in a function domain, that is, the machine-oriented version of
the functional core of a system (functions of a computer environment, or user
tasks). The form–meaning relationship in interface signs is typically metaphoric;
that is, it relies on some nonarbitrary mapping between the elements in the presen-
tation domain and the elements in the semantic domain, based on some character-
istics of the presentation that correspond to some characteristics of the meaning.
Commonly adopted presentation elements (e.g., buttons, text fields, icons) are
usually representations of objects in everyday life domains (such as the office, the
library, etc), and represent computer functions in terms of common activities al-
ready known to the user. For example, the menu interaction object class, as com-
monly encountered in popular user interface development toolkits, follows the
“restaurant” metaphor, whereas push buttons, potentiometers, and gauges follow
the “electric device” metaphor. Toolkits, which provide reusable collections of
presentation elements and facilities for associating the desired functions to presen-
tation elements, are often characterized as computational embodiments to design
languages (Winograd, 1995).

Another important characteristic of interface signs, which can be characterized
as the syntactic dimension, is the possibility to combine them to form complex
signs whose meaning is determined by the meaning of the components. It can be
plausibly argued that such a syntactic combination is determined by the underlying structure of the complex tasks to be represented in an interface, and that it obeys to (often implicit) general rules and principles that are, to a large extent, independent from specific metaphors or toolkits.

Furthermore, interface signs can be investigated, in a *pragmatic* dimension, with respect to their capability of conveying an intended meaning through some form in a given context. The notion of context is intended here in a very broad sense, including user abilities, requirements, and preferences, type of use, type of available modalities, and so forth (Stephanidis & Akoumianakis, 1999).

In the context just presented, the population of design spaces can be conceived as a process of mapping concepts in a function domain to symbols in a presentation domain, and vice versa (Akoumianakis et al., 2000; Stephanidis & Akoumianakis, 1999). Such a process should also be intended as compositionally combining single-interface artifacts into more complex ones on the basis of general principles also including context-dependent factors.

Research in the recent past has considered some of these issues. Some work has investigated the form–meaning relationship in user interfaces, and has proposed techniques for matching computer functions with representations on the basis of shared characteristics between the source and the target domain (Goguen, 1998; see also chap. 13, this volume). Other approaches, mainly in the field of model-based HCI, have proposed techniques and tools for task-driven interface design, for example, MASTERMIND (Browne, Davilla, Rugaber, & Stirewalt, 1998) and TLIM (Paternò & Meniconi, 1998). These efforts, however, though strong at capturing task semantics and interrelationships, do not account for alternative mappings between abstract task structures and concrete interaction elements. For example, let us assume that a menu is to be accessed by a sighted and blind user in order to delete a file from a list. Adopting a task-based approach, alternative options such as those depicted in Fig. 12.1 would typically be obtained.

All options in Fig. 12.1 share a common task organization with differentiation at the level of leaf nodes. Existing task-based techniques rarely provide the constructs to explicitly model the differentiated elements, thus resulting in a single design, and consequently a single implementation of the interaction. Therefore, the issue is to provide representations that do not limit the design space, but instead facilitate the selection of the maximally preferred alternative given the constraints of a particular context of use, thus facilitating the design of adaptable interfaces (see also chap. 21, this volume).

The approach presented in this chapter is inspired from recent computation-oriented approaches to language theory that establish a framework and provide an expressively adequate formalism for capturing the syntax, semantics, and pragmatics of natural languages as sign systems in a declarative and mutually constraining way. The aim is to provide a mechanism that allows the principled combination of elements from different toolkits in a task-driven fashion while imposing context-dependent constraints.
FIG. 12.1. Alternative options for the task "delete a file."

LINGUISTIC THEORY AND NATURAL-LANGUAGE GENERATION

The inherent complexity and ambiguity of natural language, as well as the emergence of new applications such as natural-language generation (NLG), has driven research in both linguistic theory and NLP toward the development of approaches that allow capturing linguistic phenomena in concise, easily maintainable, computationally effective, and processing algorithm-independent ways. This trend emerged with the appearance of unification-based representation formalisms for the representation of linguistic knowledge (Shieber, 1986), and developed into a family of language theories (and underlying formalisms) based on such a notion of unification. Unification-based grammars are characterized by the declarative representation of linguistic objects, such as words, phrases, and sentences, by means of (recursive) feature structures (i.e., attribute-value pairs), which may be only partially instantiated (i.e., some values may be variables). A basic operation over feature structures, called unification, is used for checking their consistency and merging them. In essence, unification combines (partial) feature structure descriptions of a linguistic object into a unique (more informative) description that includes all the attributes and values of the unified descriptions, provided that they are consistent, that is, no attribute is explicitly assigned different values in those descriptions. Such an approach is also called con-
straint-based, because it emphasizes the notion that a grammar constitutes a set of constraints that the admissible signs of a language should satisfy.

One of the more recent and widely adopted unification-based knowledge representation languages is the TFS formalism (e.g., Carpenter, 1992; Carpenter & Penn, 1998). In TFS, elements in the modeled domain are declared as types organized in an inheritance hierarchy. This means that types may have subtypes (more specific instances of types), which inherit and further specify their properties. A unique most general type is assumed as the root of such a hierarchy. Thus, for example, the type hierarchy shown in Fig. 12.2 could be defined for modeling sentence structure in English.

Each typed feature structure is thus composed of a type and a collection of (possibly empty) attribute-value pairs for that type. Appropriateness of attributes for a type must be explicitly declared, and is inherited by its subtypes. Values of attributes are also typed (i.e., for each feature, the type of its possible values must be declared), and they may be either atomic or feature structures. Feature structures may also share values; that is, the values of two attributes in a feature structure may be declared as identical. For example, TFS I as follows is a TFS representing the sentence “John runs”:

\[
\begin{align*}
\text{SYN} & \quad \text{sentence} \\
\text{SEM} & \quad \text{run} \\
\text{SYN} & \quad \text{RUNNER} [1] \\
\text{SEM} & \quad 1] \\
\text{SYN} & \quad \text{noun-phrase} \\
\text{SEM} & \quad 1] \text{john} \\
\end{align*}
\]

FIG. 12.2. A sample type hierarchy for modeling sentence structure in English.
The structure in TFS 1 represents the fact that a linguistic entity of type phrase has a feature named SYNSEM that has a feature structure value composed of the features PREDICATE and ARGUMENT. The ARGUMENT feature represents the syntactic and semantic characteristics of the sentence subject, and namely that it is of syntactic category noun-phrase and denotes the individual john. The feature PREDICATE refers to the sentence as a whole and states that it is of syntactic category sentence and denotes a proposition whose predicate is run. Finally, such a proposition has a semantic argument, represented by the feature RUNNER, which shares the value of the sentence argument, "john." The sharing is represented graphically through the notation [1].

TFSs are inherently partial with respect to the information they provide, and may be ordered according to how specific is the information that they provide in the inheritance hierarchy. Such an ordering is called subsumption. Accordingly, a feature structure subsumes another one if: the type of the first is more general than the type of the second; both are assigned the same attributes and the values of these attributes in the first feature structure are more general than those in the second; and finally, if two attributes share their value in the first feature structure, they also share it in the second. Unification of typed feature structures is thus defined in terms of subsumption: The result of unification amounts to the most general feature structure subsumed by input feature structures. Unification is a very important characteristic of typed feature structure formalisms, which makes them suitable for representing not only linguistic knowledge, but also other types of knowledge for which the merging of partial representations is required, as, for example, in the case of the approach presented in the next section of this chapter, the knowledge associated to the syntax and semantics of interface signs.

TFSs, through their ability to represent, combine, and further specify partial information, can be used for modeling linguistic phenomena according to a variety of language theories.1 HPSG (Pollard & Sag, 1987, 1994), a recent theory that has emerged as one of the best suited for NLP purposes, directly builds on the notion of TFS. Linguistic information in HPSG is modeled through types, type hierarchies, and inheritance. Such a theory views linguistic expressions (such as words, phrases, and sentences) as signs in the structuralist sense, relating a (phonological) form to a meaning.2 Linguistic signs in HPSG also specify information related to the utterance context.3 The notion of context includes the speaker of an utterance, the hearer, the utterance time, and so on.

HPSG is a lexicalist theory, as it takes the view that the largest part of linguistic information is encoded in the lexicon. In particular, the lexicon also contains the major part of the syntactic information of a language. Some words, the so-called

1For a brief review and further references, see Uszkoreit and Zaenen (1996).
2In HCI design, this can be assumed to be equivalent to mapping a task (e.g., database search) to an appropriate perceivable representation (e.g., icon, picture, figure, text, music, voice) or combination of representations.
3This chapter does not discuss the HPSG view of natural-language semantics and pragmatics. Pollard and Sag (1994) presented a detailed discussion of such a topic.
heads, are considered as determining the syntactic structures of the phrases in which they occur, because they impose constraints on other phrase components. These constraints are defined as the valence of a sign, that is, a list of specifications of possible complements the head can take. Thus, for example, the word *runs* is assigned the (partial) representation\(^4\) depicted in TFS 2 as follows:

\[
\begin{align*}
&\text{word} \quad \text{PHONOLOGY} \quad \text{<runs>} \\
&\text{SYNSEM} \\
&\text{CATEGORY} \\
&\text{CONTENT} \quad \text{HEAD} \\
&\qquad \text{SUBCAT} \\
&\qquad \left( \text{HEAD noun, CONTENT} \ [1] \right) \\
&\qquad \text{verb} \\
&\quad \text{run} \\
&\text{RUNNER} \ [1]
\end{align*}
\]

(2)

In TFS 2, the feature PHONOLOGY corresponds to the form of the lexical sign, whereas the feature SYNSEM contains the syntactic and semantic information of the sign, respectively specified in the CATEGORY and CONTENT features. In particular, the feature HEAD refers to the syntactic category of the sign, in this case verb. The feature SUBCAT represents the sign valence, that is, the fact that the word *runs* combines with a sign of category noun, whose content is identical with the semantic argument of "run," represented in the feature RUNNER.

The encoding of syntactic information in the lexicon does not introduce redundancy or unnecessary complexity, because properties of lexical entries can be described in a concise and principled fashion through inheritance hierarchies that allow the cross-classification of words into word classes according to their properties. Thus, each word inherits the characteristics of all classes to which it belongs. For example, the word *runs* inherits features of finite verb forms, of all verbs, of words whose meaning is a unary relation, and so on.

Phrases (i.e., phrasal signs) are seen in HPSG as projections of their heads, constructed on the basis of the valence information encoded in head lexical entries. Phrasal signs are represented as mother feature structures that include the specification of their daughters, that is, of the (lexical or phrasal) signs that compose the phrase. For example, the sentence "John runs" would be (partially) represented as TFS 3:

\[
\begin{align*}
&\text{phrasal} \\
&\text{PHONOLOGY} \quad \text{<john runs>} \\
&\text{DAUGHTERS} \\
&\quad \text{HEAD-DTR} \\
&\quad \text{SYNSEM} \\
&\quad \text{CATEGORY} \\
&\quad \text{SUBCAT} \left( \text{HEAD noun, CONTENT} \ [1] \right) \\
&\quad \text{PHONOLOGY} \quad \text{<runs>} \\
&\quad \text{COMPL-DTR} \\
&\quad \text{SYNSEM} \\
&\quad \text{CATEGORY} \\
&\quad \text{HEAD noun} \\
&\quad \text{CONTENT} \ [1] \text{john}
\end{align*}
\]

(3)

\(^4\)Examples of HPSG representations in this chapter omit information not related to the present discussion.
The feature structure in TFS 3 shows how the valence information encoded in lexical entries is used in the representation of phrases: A phrase is composed of a head sign (HEAD-DTR) and a (list of) complement sign(s) (COMPL-DTR), which share their feature values with the specifications in the head SUBCAT list. This mechanism allows for a very concise and elegant formulation of phrase structure rules, that is, of rules that recognize, or generate, phrasal signs. Rules in HPSG are called schemata, because they do not depend on the specification of syntactic categories. A single schema can combine a noun with its article, or a verb with its complements, and so on, by simply unifying with: (a) the head of a phrase, and (b) the elements required by the head’s valence specification. A phrasal sign is complete (saturated) when all complements required by its head are included in its DAUGHTERS feature.

Schemata are constrained through general principles that apply to the information contained in the mother and in the head of a phrase and complement daughters. Two particularly important principles are the head feature principle and the subcategorization principle. The first states that a mother and its head daughter always share the same syntactic category (i.e., the value of the SYNSEM/CATEGORY/HEAD feature in the previous examples). This amounts to say that the phrasal projection of a verb is of category verb, and so forth. The second principle states that the valence of the head daughter of a phrase is equivalent to the valence of the mother with the addition of the identified complements: complement daughters come to (partially) fill the slots foreseen in the valence of the head. The slots that remain unfilled when a schema is applied constitute the valence of the mother. Other, less general, principles may concern: (a) construct specific phenomena, for example, phenomena concerning specific types of phrases such as coordinate phrases, and so on; (b) language-specific phenomena, that is, phenomena typical of specific natural languages, for example, the case of sentence complements.

Summing up, HPSG is characterized by: (a) a sign-based architecture, (b) the organization of linguistic information via types, type hierarchies, and constraint inheritance, (c) the projection of phrases via general principles from rich lexical information, and (d) the organization of such lexical information via a system of lexical types (Sag & Wasow, 1999). The HPSG approach to natural language has been applied in a variety of NLP applications. Its strength is due to the simplicity of the rules, to the easiness of specifying lexical information in multiple inheritance hierarchies and, in particular, to the possibility of working with partial, underspecified descriptions of linguistic entities. HPSG grammars are completely declarative, and reversible; that is, they can be used for both parsing and generation purposes (Neumann & van Noord, 1993). The semantic head-driven generation algorithm (Shieber, Pereira, van Noord, & Moore, 1990), developed

---

Reversible grammars can be compiled in different ways according to whether they are to be used by a parsing or a generation algorithm. Generation takes as input a (semantic) representation of the text content and produces as an output the actual natural-language text corresponding to such a content.
for generating text with unification-based grammars, is the most commonly used algorithm for NLG with HPSG grammars, because it directly relies on the notions of syntactic head and valence, and exploits them in order to perform the generation process.

A GRAMMAR FOR GENERATING DESIGN SPACES

In this section, concepts and mechanisms from HPSG are applied to facilitate the population of the design space in multiple-metaphor environments. Relying on the assumption that user interfaces can be studied as sign systems, the approach outlined here aims at investigating how the HPSG framework could be adopted for defining a design grammar that describes interface signs as the combination of a form (i.e., an interface object) with a content (i.e., a specification of an application function, or user tasks), as well as the principles that constrain the combination of such signs into phrasal signs (i.e., composite interface elements such as dialogues, or complete interfaces). A generation algorithm applied to such a grammar would produce specifications of design alternatives in multiple-metaphor environments, in the form of TFS. It is assumed that such specifications can be interpreted by the run-time libraries of a user interface development toolkit that undertakes the task of realizing the specifications into a user interface implementation. For this to be attainable, however, the toolkit should exhibit API functionality that allows interoperation. Example toolkits that serve this purpose have been developed, and are described in Savidis, Stergiou, and Stephanidis (1997) and Savidis, Vernardos, and Stephanidis (1997).

The grammar-based approach relies on the representation of interface objects, their syntactic properties, and their semantics into a TFS inheritance hierarchy. Figure 12.3 depicts a simplified example of such a hierarchy. Lexical interface signs, that is, representation of toolkit objects, as the lexicon of natural languages in HPSG, can be cross-classified on the basis of their ability to represent different tasks, and to combine with each other, independently from the specific metaphor in which they are embedded. Such an assumption is similar to the concept of abstraction over interaction objects in (Savidis, Stephanidis, & Emiliani, 1997). In a TFS hierarchy, abstraction can be elegantly captured through the extensive use of underspecification: Features are assigned as values that are not the most specific task types in the hierarchy, in such a way that a unique description subsumes a set of (more specific) descriptions.

The semantics of interface signs can be straightforwardly modeled in the grammar as a hierarchy of user and system tasks, based on the notion of task context (Akoumianakis & Stephanidis, 1997). A task context is considered as a characterization of "dialogue states," that is, indicators of what the user interface is doing at

---

6 Semantic head-drive generation combines top-down and bottom-up techniques to connect phrases in sentences. The algorithm recursively looks for a semantic head on the basis of the semantic specification to be generated, generates the head daughters, and then connects the head to its mother.
any point in time. Task contexts encompass a degree of abstraction that entails that the same task context may be performed differently by different users, or the interface may exhibit differentiated behavior in the same task context depending on the current user. As a consequence, the same task context may be mapped onto several interface objects. In the grammar, tasks can have subtasks, which are either tasks (i.e., they can have their own subtasks), or lists of tasks of the same type. So, for example, a search task could be represented as TFS 4 or 5 as follows:

\[
\begin{align*}
\text{search} \\
\text{SUBTASK1} & \text{ search\_condition} \\
\text{SUBTASK2} & \text{ result\_presentation}
\end{align*}
\]
In TFS 4 the task `search` is “decomposed” into two subtasks, namely the formulation of a query and the presentation of the result. In TFS 5 the second subtask of `search` is a list of tasks (all of type `page presentation`) whose number is not specified.

Additionally, the proposed approach assumes that interface objects combine in head-driven fashion, that is, some of them behave like heads that specify a valence partially determining the category and semantics of the other interface objects they can combine with. Abstractions over properties of interface objects as for their combination with other objects are adopted for modeling interface sign syntax. A typical example of such an abstraction is the concept of container (Savidis, Stergiou et al., 1997), which captures the characteristics of objects like windows, books, html pages, and so on. Containers represent the main interaction areas for tasks, and may encompass other objects representing subtasks like menus, buttons, and so on. In the grammar, containers play the role of heads; that is, they are assumed to have a (partially specified) valence, which becomes further instantiated according to the task assigned to a container. Other examples of heads could be menus, which subcategorize for menu items according to the subtasks of the main task at hand, and toolbars, which subcategorize for buttons. The phrasal projection of a menu, that is, a menu containing a specified list of items corresponding to commands, can constitute one of the daughters of a container phrase. Categories such as container, menu, toolbar, and so forth, are therefore used in the grammar in a fashion similar to that of syntactic categories in natural-language grammars.

The inheritance hierarchy of the devised grammar also includes interface objects, simply classified according to their physical properties. Thus, for example, the classification includes window objects, book objects, button objects, and so on. Furthermore, the hierarchy subdivides interface signs into lexical and phrasal signs, and assigns feature description to signs. Lexical signs, which provide the structure of entries in the grammar lexicon, have features representing constraints on the interface object of a sign and on its syntactic and semantic properties. Phrasal signs, on the

---

5Containers differ from other composite interaction elements insofar as they exhibit additional interactive behaviors such as navigation, access policy, and topology of contained elements.

6This chapter does not deal with issues related to spatial and temporal relationships of interface objects in interfaces, because the goal of the grammar is not to produce full interface specifications, but recommendations that can be interpreted and applied by the run-time libraries of toolkits, through appropriate application programming interface (API) functions. However, it is assumed that the inheritance hierarchy and the grammar can be extended to represent these relationships.
other hand, constitute the structures produced by phrase structure (generation) rules, that is, complex interface signs composed of lexical signs.

In the lexicon, semantic, and, to a lesser degree, syntactic properties of signs are underspecified. Lexical signs are assigned a general task type, so that they are compatible with all the specific tasks subsumed by such a type. This approach, though ensuring that the right type of task is associated with the right type of interface object (e.g., buttons are not associated with overall tasks), avoids the necessity of specifying for each interface object all the tasks it can be assigned, because this is defined implicitly through the inheritance hierarchy. TFS 6 is an example of lexical entry for a book container object:

```
[lexical INTERFACEOBJECT book
  [CAT [HEAD top_level_container
    [(CAT HEAD simple_container, SEMTASK [1])]
    [(CAT HEAD simple_container, SEMTASK [2])]
    [top_level_container_task]
  ]
  [SUBCAT [SUBTASK 1 non_top_level_container_task, [1]]
    [SUBTASK 2 non_top_level_container_task, [2]]
  ]
]
```

The feature structure in TFS 6 represents a lexical sign, relating an interface object, namely a book, as specified by the value of the feature INTERFACE_OBJECT, with a specification of syntactic and semantic properties, that is, the complex value of the feature SYNSEM. In particular, the sign is defined, as being a top_level_container (SYNSEM/CAT/HEAD), that is, a container representing an overall task. Its valence, represented by SYNSEM/CAT/SUBCAT, subcategorizes for two non_top_level_containers (corresponding to the two visible pages in an open book). Finally, the sign receives an underspecified semantic value (SYNSEM/SEMTASK) corresponding to top_level_container_task, with two subtasks also underspecified. This means that the sign can represent any individual task of type top_level_container_task with two subtasks of type non_top_level_container_task (e.g., search). These two subtasks correspond to the semantics of the non-top-level container (e.g., search_condition) required by the sign valence. Their values are shared in the feature structure. Figure 12.4 provides a graphical representation of an abstract "book" object as captured in TFS 6.

A very small number of general phrase structure rules is sufficient for generating design alternatives. For example, a set of three schemata is sufficient to cover headed constructs, including those allowing an (initially) unspecified number of subtasks for some components. Schemata work as follows:

1. The first schema combines a head with a specified set of daughters to generate a mother phrase by applying the head feature principle and the subcategorization principle.
2. The second schema combines a head with an unspecified set of daughters, all characterized by a subtask of the same type (e.g., page\_presentation) and an interface object of the same type (e.g., non\_top\_level\_container), by applying the head feature principle and the subcategorization principle. A dummy interface object can constitute the head of such a phrase (e.g., an “invisible” composite container made up of several pages).

3. The third schema simply generates phrases from heads with empty valences, and therefore no daughters.

The described grammar-based approach exploits the underspecification and multiple inheritance mechanism built in to the formalism and grammatical framework adopted. When a structured user task is given as input, the grammar produces all (multiple-metaphor) interface signs that can convey such a task. In the generation process, the input task and its subtasks are recursively unified with phrasal interface signs starting from the head of each construct in a head-driven fashion. The semantic underspecification of lexical entries allows the direct unification of the input with all interface signs whose semantics unify with the task at hand. So, for example, supposing that the grammar contains an entry for a book top-level container such as the one depicted earlier, as well as appropriate entries for two non-top-level containers, namely a window and an html\_container, when request-
ed to generate a phrasal sign for a search task, it will: (a) unify the semantics of book to search, with the two subtasks search_condition and result_presentation; (b) produce four alternatives, with all possible combinations of window and html_container as non-top-level containers instantiating the search subtasks. Figure 12.5 graphically represents the four generated alternatives.

A grammar of this type may also handle redundancy in user interface signs, that is, the degree to which alternative objects for the same subtask(s) are included in an interface sign. For example, a window may provide navigation commands in the form of menu items, grouped into a menu, or buttons grouped into a toolbar, or both. Various degrees of redundancy may be obtained, for example by encoding two (or more) lexical entries for the same object, which respectively enforce or disallow redundancy. In the window example, two entries for the object window would be encoded, both with an underspecified semantics constituted by a top_level_container_task, which has as one of its subtasks an underspecified command_list. The SUBCAT of the two entries would differ: One entry would subcategorize for one phrase repre-
senting the command list (i.e., sharing its semantics), whereas the second entry would subcategorize for two phrases representing the command list. Supposing that the grammar allows only buttons and menu items to represent commands, in generation the first entry would produce two output results, namely a window with a toolbar and a window with a menu. Both the toolbar and the menu would contain as many buttons or menu items as are the commands specified in the input semantics. The second entry would produce one output, namely a window with a toolbar and a menu. Globally, the generation of an input task involving navigation commands would produce the three alternatives mentioned previously. In summary, generated design alternatives may vary with respect to: (a) semantically equivalent objects from the same toolkit, (b) semantically equivalent objects from different toolkits, and (c) the degree of redundancy.

DISCUSSION

Reconsidering the key ideas underlying HPSG, as summarized at the end of the third section, from the viewpoint of HCI design, the outlined grammar-based approach appears to be characterized by: (a) a sign-based account of interface objects and their semantics, (b) the organization of knowledge concerning tasks, interface object properties, and combinatorial behavior via types, type hierarchies, and constraint inheritance, (c) the projection of complex interface constructs via general principles from rich lexical information, and (d) the organization of such lexical information via a system of lexical types.

The advantages of such an approach can be viewed at two different levels, namely a “representational” and an “architectural” level. At the representational level, the grammar constitutes a concise and elegant device for modeling properties of interface signs. In particular, the grammar appears well suited to establish an abstract representational level for modeling the syntax and semantics of interface objects in a metaphor/toolkit-independent fashion. In the adopted approach, abstraction is used to characterize not an object itself, but its combinatorial behavior in interface signs. The underspecification mechanism built into the TFS formalism provides a mean of associating interface objects with tasks at a very abstract level, relieved from the toolkit-specific intrinsics and details. At the same time, instantiated interface specifications are produced by unifying the underspecified lexical items with input semantic structures that correspond to specific instances of their semantics. Therefore, it can be claimed that the described approach is capable of generating interface specifications in multiple-metaphor environments without requiring an explicitly encoded pairing of interface objects with specific tasks, which would be time and resource consuming in an environment characterized by a wide variety of possible tasks and metaphors. This leads to the conclusion that design tools based on such an approach would minimize the effort required for encoding, maintaining, and updating the knowledge necessary for

These can be accounted during interface implementation, when mapping design specifications to concrete implementations.
automatically populating the design space of multiple-metaphor environments. The grammar-based approach would also maximize the tool extendibility to account for new applications and metaphors/toolkits. In fact, the modeling of new applications affects only the subhierarchy of tasks without requiring changes to the lexicon or the rules, whereas new interface objects can be added to the lexicon without concern about the low-level content of the task hierarchy.

From an architectural point of view, the outlined grammar-based approach offers the possibility of investigating the interrelationships between syntactic, semantic, and also pragmatic characteristics of interface signs within a single framework. This brings into HCI design in general, and in user interfaces for all in particular, the insight, typical of contemporary computation-oriented language theories, that levels of analysis such as lexicon, syntax, semantics, and pragmatics, are not to be viewed as a sequence of separate models, but instead as a set of mutually constraining properties of signs. Such a perspective is particularly attractive for the purposes of investigating how different metaphors can be combined and integrated into user interfaces on the basis of user tasks in a principled and contextually grounded fashion. In fact, an interesting aspect of such an approach is the possibility of enforcing various types of constraints on the generation of design alternatives. Intrinsic grammar constraints are the already mentioned head feature principle and the subcategorization principle, which govern the combination of interface signs at an abstract level. Other principles may apply to specific types of constructs. For example, some interface signs can enforce metaphor coherence or object identity in some of their component parts: A toolbar can contain only buttons, a menu only menu items, and so forth. Another important category of constraints could be of a completely different nature, that is, extragrammatical constraints related to the context of use of an interface, such as user abilities, requirements, and preferences, and factors related to the type of usage and access device. For example, constraints could ensure that consistency is maintained in the specification of a design alternative with respect to the preferred user dialogue style, language, and so on. Following the distinction between "universal" and "language-specific" principles in HPSG, it could be argued that principles in an interface grammar can be classified into general principles whose validity holds across different toolkits or metaphors, for different users, and so on, and principles related to more specific aspects of user interfaces. Context-related constraints would be introduced in the grammar in the form of additional principles, which would then be applied in generation along with grammar internal principles. In such a way, context-related principles would "filter out" incoherent or contextually inappropriate design alternatives.

The design grammar proposed in this chapter is not meant to produce interface implementations, but, rather, interface specifications. It is intended to be applied in design tools integrated into specification-based development environments (Stephanidis & Akoumianakis, in press). In such environments, a design tool based on the proposed grammar would act as a source of design recommendations, in the form of specifications, which, once compiled, can be interpreted and applied by the
run-time libraries of the user interface development system, thus allowing the instantiation of alternative interactive behaviors. In particular, in the framework of unified user interface development, described in Part VI of this volume, such a design tool could contribute orthogonally by providing semantic-based design specifications to be integrated with physical-level specifications produced by other tools (see chap. 23, this volume). In order to capture and apply context-related principles, the grammar could cooperate with other tools, such as user-modeling (Akoumianakis et al., 1997; see also chap. 23, this volume) or decision-making (Karagianni, Koumpis, & Stephanidis, 1997) tools, which would provide information concerning user preferences, dialogue style to be used, modality, device, and so forth. The produced specifications would then be implemented according to unified user interface implementation (see chap. 22, this volume), which allows the instantiation of different design alternatives in a single interface.

SUMMARY AND CONCLUSIONS

This chapter has presented a sign-based approach to the population of design space in multiple-metaphor environments. Borrowing from recent developments in language theory and in NLG, the chapter has proposed a grammar for the generation of design alternative specifications that is capable of mapping abstract user tasks to interface specifications combining elements from different toolkits. The main features of the design grammar, which is inspired from the HPSG theory, are: (a) hierarchical structuring of knowledge through typed feature structures, (b) highly underspecified (abstract) representation of lexical knowledge, (c) head-driven approach to the generation of specifications from tasks, and (d) introduction of principles constraining the combination of interface objects in complex constructs.

The advantages of the proposed approach have been identified in its representational conciseness and ease of modification and extension, as well as in its capability of imposing a variety of constraints on the specification of design alternatives, including user- and context-related constraints. The proposed design grammar is claimed to introduce a new approach to the population of design spaces in multiple-metaphor environments, capable of supporting the designer in a task up to now largely carried out without system support, or at best, with artifact-oriented rather than user- and context-oriented support. Furthermore, the proposed approach has been claimed to be both compatible and useful to the unified user interface development framework, elaborated in Part VI of this volume.

REFERENCES


