A Data Modeling Approach to Simplify the Design of User Interfaces

Michel Pilote

Department of Computer Science *
University of Toronto
Toronto, CANADA

ABSTRACT

What is most crucially lacking in the area of user interface design is a set of tools to integrate various mechanisms that are well understood and useful, but each addressing only limited aspects of the problem.

This paper demonstrates how Data Modeling techniques can greatly simplify the design of user interfaces. The main contribution of this work is a single, highly coherent and very simple framework that can uniformly represent any aspects of a user interface.

The most significant departure of our approach from other programming languages is the complete and explicit separation that we make between semantic and syntactic aspects of an application. We also introduce internal objects to model various states of an application and the various state transformations that are allowed between these states.

I. Motivation

More than 50% of the code in most commercial applications involves the definition of interfaces with the users of these applications; furthermore, this part of the code is the one that typically requires the most customization and maintenance, to reflect changes in users' requests and methods of operation. Any simplification and reduction of this portion of software development is therefore critically needed [Lientz and Swanson 81].

A *user interface* can be defined as any mechanism used to mediate between users and applications. These include in the simplest case all Input/Output formatting, up to the most complex cases of so-called "natural language" front-ends to data management systems. Although our approach could also be applied to other (graphical or hardware oriented) types of user interfaces, we concentrate our research on word-oriented user-interfaces in this work.

* Current address: Netron Inc., 99 St. Regis Cres. N., Downsview (Toronto), CANADA, M3J 1Y9

In this range of interfaces, going from rigid input formats and command languages to systems trying to make sense of arbitrary English expressions, the latter obviously presents a more difficult problem. But any solution to this larger problem can also be applied to represent simpler mechanisms. The following discussion concentrates on the problems and issues involved in building the more sophisticated types of user interfaces, with the understanding that these solutions also apply to simpler cases. We will, however, restrict our attention to natural language systems used in practical situations today, as opposed to the more advanced ones still under research.

II. Overview

This paper demonstrates how Data Modeling techniques can greatly simplify the design of user interfaces.

The first simplification is to break down the problem into three components: semantics, syntax and pragmatics, according to a distinction familiar to linguists. We approach the design problem for interactive information systems from a knowledge representation point of view. To us, such a system is above all a knowledge base of "facts". Some involve the outside world, its entities, their interrelations, the events they participate in and their histories ("semantic" knowledge). Others are about the grammar and the lexicon used for a particular user interface ("syntactic" knowledge). Yet others are about the dialogue structure the system is expected to support ("pragmatic" knowledge).

A second simplification is to re-use as much as possible of the syntactic means used to specify the programs and data definitions typically specified by a programmer, to automatically provide a working prototype of a user interface. This approach is made possible by a Data Modeling framework which *uniformly* represents all three aspects of a user interface.

This opens the way to a third simplification, which is to defer to eventual users of the interface the definition of any syntactic variation above the generated prototype. This might include defining more English-like front ends with English words and idioms, or conversely, defining shorthands and codes for often-used input messages.

III. Related work

Among systems for natural language access to data bases that perform well enough to deserve consideration as practical systems, we find LIFER [Hendrix *et al.* 78], LUNAR [Woods 77], PLANES [Waltz 78], SOPHIE [Burton and Brown 76], REL [Thompson and Thompson 78], RENDEZ-VOUS [Codd *et al.* 78] and TQA [Plath 76, Petrick 81]. These research projects have already lead to commercial products, starting with Harris' INTELLECT [Harris 78] at Artificial Intelligence Corporation, and now followed by Gary Hendrix' STRAIGHT TALK at Symantec and Roger Schank's work at Cognitive Systems. The state-of-the-art in non-experimental natural language systems, as recognized by [Barr and Feignenbaum 79], is still very much typified by INTELLECT and LIFER.

III.1. Problems and issues in user interfaces

The experience with users of LIFER applications, as reported in [Hendrix *et al.* 79], has demonstrated that a number of facilities are required to obtain more "natural" user-interfaces. The most important of these desired facilities are:

1. a syntactically motivated grammar;
2. more complete and flexible paraphrase mechanism;
3. meta-knowledge access;
4. uniform representation for all components of a user-interface.

An important goal of current research in user interfaces is *transportability*: to enable non specialists to adapt a natural language processing system for access to an existing conventional data base. Because of the near-impossibility of transporting LIFER grammars from one domain to another, the LIFER development team has moved, in their subsequent research [Hendrix *et al.* 79], toward developing linguistically motivated grammars, which would facilitate the transfer from one domain to another. [Robinson 82] is the most recent account of these efforts. However, the use of linguistically motivated grammars introduces new problems. "The root of these problems lies in the very uniformity of syntactic coverage that makes linguistically motivated grammars transportable and resistant to gaps in coverage. In particular, a uniform treatment of syntax demands a uniform semantic system" [Hendrix *et al.* 79], i.e. a uniform treatment of syntax and semantics, not only for data, but for all aspects of the system.

Also, "users want more than just access to the data actually recorded in their data bases; this has been shown in [Tennant 79], where a distinction is made between linguistic completeness and conceptual completeness" [Hendrix *et al.* 79]. According to Hendrix, the most promising approach to this problem involves the creation of an intermediate representation level that mediates between the language processor and the various resources available.

A significant reduction in development and maintenance efforts could be achieved if the users themselves could specify the particular ways in which they would prefer to use their applications. "System users and members of the academic community are in general agreement that one of the most interesting and useful feature of (LIFER-based systems) is their ability to be taught new syntactic constructs by ordinary users at run time" [Hendrix *et al.* 79].

A related approach is to minimize the basic core of rules that is essential to process the user's initial lexicon entries and rules, as illustrated in [Shapiro and Neal 82], so that "a user could then input rules and assertions to enhance the system's capabilities to acquire both linguistic and non-linguistic knowledge. In other words, the user will define his own input language for entering knowledge into the system and conversing with the system."

Wording the system's questions in a comprehensible form involves a number of human engineering difficulties, as reported in [Grosz 82]. An important issue is therefore, as expressed in [Haas and Hendrix 80], that "a set of readily understandable questions is needed for eliciting information from tutors. The length and number of questions should be minimized to impose as small a burden on tutors as possible."

A common issue faced by all of the above mentioned existing or proposed natural language interfaces is the problem of "complexity": the barrier imposed by the sheer number of details to handle. Various proposals to control this complexity revolve around the notion of abstractions, and organization of abstractions into hierarchies and network structures. These tools apply as much to the concepts that we want to represent with natural language as to the programming constructs needed to implement the support for handling natural language interfaces.

[Winograd 79] provides a good summary of what needs to be done to achieve a higher level programming system better suited to develop and maintain complex applications like natural language interfaces:

> A higher level programming system must emphasize the use of descriptive languages for communication, with the ability to create and manipulate descriptions in an effective, understandable way. Existing formalisms for description (e.g. predicate calculus) are clear and well understood, but lack the richness typical in descriptions which people find useful. They can serve as a universal basis for description but only in the same sense that a Turing machine can express any computation. They lack the higher level structuring which makes it possible to manipulate descriptions at an appropriate level of detail.
>
> - T. Winograd, *Beyond Programming Languages*, 1979

Only a few of the above issues are addressed in this paper, but a more complete treatment can be found in [Pilote 83a].

IV. Our Approach

IV.1. Overview

As argued in [Pilote 83b], most programming languages offer very few constructs to explicitly deal with the design of user interfaces. We believe that what is most crucially lacking in this area of user interface design is a set of tools to integrate various mechanisms that are well understood and useful, but each addressing only limited aspects of the problem.

The essence of our approach is a ***uniform*** representation framework, able to describe and provide access to its own definition. We call this representation framework INTERPRET, to express the essence of our approach, which consists of ***interpreting*** (or ***translating)*** a user-oriented notation into a machine understandable one, and also to acknowledge a strong influence from the so-called "interpreter-oriented" methods of describing the formal semantics of programming languages, such as Denotational Semantics [Gordon 79], on the design of the INTERPRET language.

We limit ourselves to mechanisms allowing primarily *access* to data and programs ***already stored*** on a computer. By this we mean that we view the role of a user interface as providing access to data and programs *previously* defined by a professional programmer using a programming language, as opposed to allowing users to define directly new data types and programs through an interface. This restriction is conjectured as very significant in reducing the number of possible intents in user utterances.

As for the Data Modeling foundation of our work, we used TAXIS [Mylopoulos *at al.* 80] as starting point since, among a "bewildering variety of knowledge representations ... one of the most complete is the TAXIS system which has aspects of all basic kinds of knowledge" that can be distinguished in the current projects and approaches to knowledge representation [Sowa 80]. From its strong influence from Artificial Intelligence, TAXIS is acknowledged as "rich enough to support a natural language interface to knowledge-based systems" [Sowa 80]. INTERPRET, described in detail in [Pilote 83a], simplifies and extends TAXIS [Mylopoulos *et al.* 80] to allow for the description of all aspects of user-interfaces.

IV.2. Basic representation framework

As in TAXIS, the INTERPRET framework considers three basic types of **objects**: tokens, classes and metaclasses. *Tokens* are undecomposable units of information, usually modeling actual entities in an application domain. *Classes* correspond to collections of tokens sharing some common "properties" (to be defined below), which tokens are said to be ***instances*** of the class; this INSTANCE-OF relationship relates an object, e.g. **John**, to a class of which it is an instance, e.g. PERSON. Similarly, collections of classes can be themselves grouped into higher level classes, called *metaclasses*.

All (meta)classes constituting a TAXIS or INTERPRET program are organized into an IS-A hierarchy in terms of the binary relation IS-A which is a partial order. This IS-A relationship will sometimes be referred to as "specialization" when going from more to less general, or conversely, "generalization". The IS-A relationship relates a class, e.g. STUDENT, to another more general one, e.g. PERSON. In particular, relations, transactions and exceptions are all treated as classes defined through the properties that relate them to other classes, and organized in terms of the IS-A relation into a hierarchy.

The main difference between TAXIS and INTERPRET's IS-A and the traditional ***subset*** relationship is that the IS-A relation holds even between classes with no instances. The subset relation between the sets of instances of IS-A-related classes is thus simply a side-effect of the ***definition*** of a particular IS-A relation.

Classes and metaclasses model conceptual objects which are "defined" by their relations to other concepts, and the operations that are allowed on their instances, much in the spirit of "Abstract Data Types" in Programming Languages. Both these relations and operations are viewed as ***definitional properties*** attached to (meta)classes. These definitional properties restrict the ***factual properties*** that can be defined on instances of these classes. For example,

property age **on** PERSON **is** {0..200}

specifies that the *age* of a particular person, say **John**, must be in the range {0..200}. The following expression is then acceptable:

John.age ← 22

meaning that the value of the property *age*, when applied to the object **John**, becomes the number 22.

A new feature of INTERPRET over TAXIS is to consider properties as objects. This means that a property category is itself a class of objects in INTERPRET, whose instances are properties. The most general class of properties is called "**property**" (or alternatively "**properties**"), of which all other property categories are specializations.

IV.3. Three steps design methodology

The design of a user interface can first be simplified by decomposing the problem into separate subcomponents. The *Example* section below illustrates the design methodology that is made possible by the uniform representation of all aspects of a user interface:

1. Define the semantic data objects and programs;
2. Define the valid dialogue paths, and specify responses to exceptional situations;
3. Add a syntactic covering on top of the above facilities, aiming at making them more "natural"; following the approach advocated in this research, this step could even be handled by a "casual" user.

The decomposition of a user interface into three components is particularly significant to reduce the effort involved in setting up new user-interfaces for casual users. The methodology proposed, which is as far as we know original and unique, is to use as much information as possible from an initial specification by a programmer, to the extent of being able already at this point to provide quite "flexible" facilities to a "non-technical" user. This working basis can then be successively extended in accord with the syntactic preferences of the user, which may be more English-like, or may even be more formal and abbreviated if so wished.

As a result of these simplifications, the task of designing a user interface can in the extreme case be reduced to the specification of dialogue paths.

Note that, in a sense, the mechanisms of the syntactic and pragmatic components, since they must be represented and stored as data and programs in a computer, are also part of what we called the semantic component. In fact, one of the prime goals of this work is to provide access to these program and data objects, using the same mechanism and procedures that are used for more traditional data and programs in data bases and program libraries. But for the sake of the discussion, it will be useful to distinguish the *accessing* mechanisms, classified into syntactic and pragmatic, from the *representation* of information stored in the computer.

IV.4. Maximize the use of predefined information

The second way in which we simplify the task of designing a user interface is by taking advantage of the information already provided by a programming language specification. This information is made of two parts: 1) identifiers; 2) a grammar for the programming language. By replacing the programming language grammar by a more English-like one, we already obtain a more flexible and user-friendly interface. Then a wide variety of user inputs in "natural" language can be translated into a formal equivalent that can be accepted by the system.

The approach, illustrated in the *Example* section below, of *explicitly* and *systematically* using syntactic information provided by programming language specifications to support a user interface appears to be new: this approach is made possible by our uniform framework which can combine the representation of "internal" information with "external", user-defined knowledge.

IV.5. Deferring syntactic customization to the user

Finally, a third way to reduce the job of the user interface designer is to defer part of this job to a user of the resulting interface. First, by representing much of the above mechanism in a sufficiently organized formalism (i.e. in terms of INTERPRET constructs), we increased the comprehensibility of the interface and reduced the effort required for a user to understand and introduce further modifications himself. This involves primarily modifying the syntactic component of a user interface, to fit particular preferences, since the semantic and pragmatic aspects of a user interface are supposed to be defined by professional programmers For example, one programmer may like a very concise and dense notation, while another will prefer full length words, with lots of prompting from the system. Understanding the underlying information is very important to make modifications possible, but is even more critical to "debug" and integrate modifications into the rest of the system.

The next step, illustrated in our example, is to increase the flexibility of the vocabulary by introducing synonyms for already defined identifiers and custom paraphrases beside standard English transformations. The motivation for a syntactic training mechanism is that users need to be able to adapt the syntax of an interface to the particular vocabulary of an application, and their own particular tastes. These are too diversified to rely on computer or linguistic experts to provide the required changes. Furthermore, customizing is believed, from the author's personal experience with "end-users" of computer systems, to be one of the most important aspects of a successful user interface.

Current solutions to this requirement for the acquisition of new syntactic knowledge typically involve long-winded dialogues to gather the syntactic classification and features of new words; alternatively, such information is often explicitly given via a programming language. The first method is likely to strain the patience of its users while the second requires a deep familiarity with programming and linguistics.

Our approach here clearly belongs to the "language engineering" stream: it is all based upon conventions between users and programs. Our main innovation in this respect is to allow the customization of an interface, as shown below; we also describe in [Pilote 83a] techniques that could be used to describe and access the interface itself, to allow a user to understand its features and limitations and, eventually, to modify the interface itself.

V. Example of the three steps design of a User Interface

The last section of this paper present a highly simplified example of the kinds of information that must be incorporated in a user interface to make it truly "flexible". Although restricted, this example also illustrates the complexity of the phenomena to handle, most of which are often only skimmed in many interfaces aiming at "user-friendliness".

Our sample application domain is the institutional world of a university, and our particular example is the identification of a particular student. The first step of an interaction between a user and the system consists of the user answering a request from the system to identify himself, from which the system will decide which data and programs can be made available to this user. We describe how, even for such a simple situation, the number of possible responses is unbounded. However, by taking advantage of the rules offered by a built-in English grammar, the number of "patterns" required to match most of these user inputs are very limited.

We will first aim at a minimal mechanism able to produce a particular desired result, without any concern for user oriented features. Our interface will include barely enough information to support the *semantics* of an application. The important point is that, already after the first step, the user is provided with a working system which can be evaluated against his requirements.

Then, in a second step, we will specify dialogue paths as an extension of the semantic component of our interface. These dialogue paths will specify at any time the range of possible actions that can be triggered in the system or the items of information that must be obtained by the system from the user.

Thirdly and finally, we will define a syntactic interface for the above facilities that will allow either access in a programming language-like format, or in an English-like more "natural" fashion. This last step illustrates that artificial and natural languages are not incompatible but can, in fact, be intermixed according to the user's needs and preferences.

Appendices 1-4 collect the detailed and complete INTERPRET declarations supporting the following example. In this example, user inputs are shown following the symbol ">".

V.1. The semantic component

The semantic component for our example includes a data class representing student information, as shown in *Appendix 1*. Graphically, our semantic component includes the data classes and tokens shown in Figure 1.

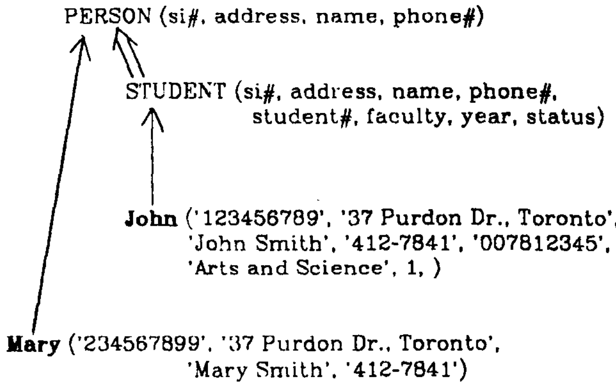

PERSON (si#, address, name, phone#)
STUDENT (si#, address, name, phone#, student#, faculty, year, status)
John ('123456789', '37 Purdon Dr., Toronto', 'John Smith', '412-7841', '007812345', 'Arts and Science', 1,)
Mary ('234567899', '37 Purdon Dr., Toronto', 'Mary Smith', '412-7841')

Fig.1- Semantic object defined in Appendix 1; double arrows represent the IS-A relationship, and single arrows represent INSTANCE-OF.

Instances of the classes PERSON and STUDENT are data objects with a number of attributes. Using only semantic facilities (with a standard programming language syntax -- INTERPRET in this case) we can explicitly refer to instances of these classes with expressions like:

a1> **John** ← **the** STUDENT
with name = 'John Smith';

where **John** is an identifier assigned values by the INTERPRET expression following the arrow. An expression like '*variable-name* ← **the** *CLASS* **with** *property* = *value*' can be viewed (in terms more familiar to many readers) as a database "query".

If this request is not sufficient to identify uniquely an instance of STUDENT, the INTERPRET construct **the** will raise the exception MORE-THAN-ONE. The user then has to examine the environment, maybe query the class STUDENT to examine the multiple instances named 'John Smith', to be able finally to pinpoint a combination of properties able to identify the desired individual. To get a unique instance of STUDENT, the user may end up having to re-enter a more complete expression like:

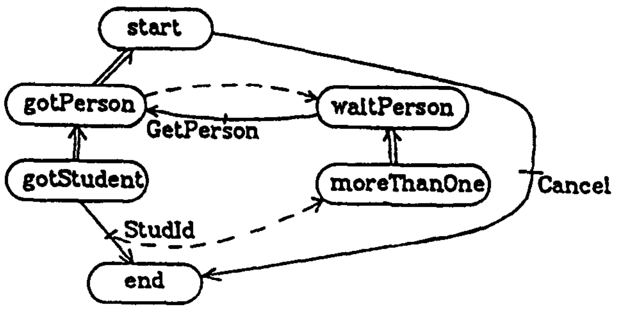
John ← **the** STUDENT **with** name='John Smith',
address='37 Purdon Dr., Toronto';

V.2. The pragmatic component

V.2.1. Definitions

A user could obtain essentially the same result as in the above semantic component more simply by allowing the program to guide the interaction and explicitly ask for needed information. This effect is obtained via "scripts", which basically specify the allowable successions of input/output interactions between the user and the program. INTERPRET scripts simplify and extend TAXIS scripts [Barron 80], inspired from Zisman's Augmented Petri Nets [Zisman 77]. A script to support and complement the above semantic component is shown in *Appendix 2*.

Fig.2 - Graphical representation of MASTER-SCRIPT.



This "master" script can help identify unique instances of the class STUDENT. It directs a complete session with a user, and is activated by some unspecified means. The first action of this script is to execute the INTERPRET expression

take(user, person);

The variable *user*, defined for this master script, contains a reference to the user terminal and must be included in any script expression used to communicate from the program to the user. Once a request has been made for some value, these can be furnished to the system in any order, after an arbitrary period of time. Scripts are designed to stay active for as long as they have not completed their purpose, i.e. until their state *end* is "reached". Conversely, this user sends messages to the script by mentioning a reference to this script in an expression such as:

give(system, person ← **John**);

where 'system' is the internal lexical token denoting the instance of MASTER-SCRIPT used in our example, and where '**John**' is the name of an instance of the data class PERSON. This instance could have been obtained as in the "semantic component" section above. So far, the only gain is to allow to specify argument when desired, and, to possibly perform some action automatically, in case a particular exception is raised.

We could also further simplify the user responsibility by automating the process of selecting a unique instance of a variable class, in this case STUDENT. This simplification would involve defining an additional script-class to gather enough property values on STUDENT to identify uniquely an instance of this class.

V.3. The syntactic component

V.3.1. Internal syntax

Nothing has been said in the above "script" example about how the user is informed of a request by a script for a particular item of information. Unless explicit messages are provided, the above **take** commands will generate default expressions in terms of the requested object. The default format on a **take** command is:

"<script-id>: Please enter a <object.name>"

where the identifier of the script requesting an input from a user is shown before the command. This set-up results in the following dialogue from the user point of view:

```
System: Please enter a PERSON.
 c1> John.
```

The purpose of the syntactic component of a user interface is to support the syntactic aspects of printed man-machine communications, such as extracting semantic information from the above user input, or from any equivalent form, as we will see below.

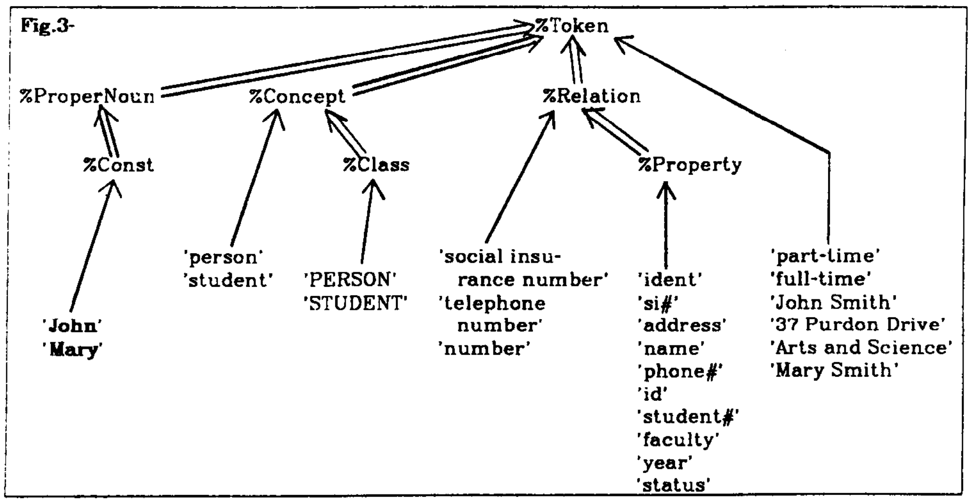
We have seen above the representation of the semantic component for our example of a user interface for an Interactive Information System. The semantic declarations also define entries in an "internal lexicon". These lexical objects can be used by two built-in grammars to provide, on one side, a programming language format (the INTERPRET language) for programs listings, maintenance and further developments by programmers and, on the other side, an English-like format to describe and recognize semantic objects while interacting with "casual" users unfamiliar with INTERPRET syntax. We want to stress that already at this stage, an INTERPRET program can be used and manipulated by both kinds of users, using this internal lexicon and built-in grammar rules.

For example, the syntactic component of a user interface could extract from the above user input a reference to an instance of the data class PERSON.

After the data definitions shown in *Appendix 1*, and before any additional syntactic information, the user interface knows about the lexical tokens shown in Figure 3 as direct instances of the classes &Const, %Class, %Property and %Token. As shown in this figure, beside the lexical token '**John**', which is recognized as an instance of the class PERSON, the lexical information known about the class PERSON includes the lexical tokens 'PERSON', 'ident', 'si#', 'address', 'name' and 'phone#', for the definition of the class itself, plus the lexical tokens 'STUDENT', 'id', 'student#', 'faculty', and 'year', for its specialization classes. Also known in the internal lexicon are the values for enumerations of lexical tokens, such as 'part-time' and 'full-time'. Finally, the lexicon includes the printable values for the factual properties of the semantic tokens defined in the data base. These include the lexical tokens 'John Smith', '37 Purdon Dr., Toronto', 'Arts and Science' and 'Mary Smith'. We can see from the above list that many of these lexical tokens can be used directly in natural English expressions.

Note how "internal" lexical classes like *%Const* and *%Class* are specializations of "English" lexical classes like *%ProperNoun* and *%Concept* respectively,

Fig.3-



which are used by a built-in English grammar such as shown in *Appendix 4* to recognize English-like expressions. In our example, the lexical token **'John'** is first recognized as an instance of *%Const*, and therefore also as an instance of *%ProperNoun*. Any rule of the Built-in English Grammar using a *%ProperNoun* will therefore accept **'John'** as a proper noun.

V.3.2. Syntactic customization

A distinctive characteristic of INTERPRET over TAXIS and over most other programming languages is that we explicitly model all syntactic objects, i.e. there is an internal object for each printable object in the interface.

Syntactic (lexical) objects are connected to the semantic object which they *denote* by the relation named *den*. Many syntactic objects can be defined as denoting the same semantic object by using the relation *trans* between them, specifying that one is a *translation* of the other. For example, given the lexical token 'John', the identifier **'John'**, the token **John**, and the following factual properties:

```
'John'.trans = 'John-Smith'
'John-Smith'.den = John-Smith
```

we can derive the following equalities:

```
'John'.den
    = 'John'.trans.den
    = 'John-Smith'.den
    = John-Smith
```

although we gave no direct representation of the factual property

```
'John'.den = John-Smith
```

In summary, the essence of our approach to syntactic customization is to introduce intermediate *translations* leading to some *denotation* predefined by a programmer.

The most interesting aspect of the syntactic component is the extensions that a user interface designer may define for a particular application, and for particular users. These extensions include both additions of lexical and grammatical information. These additions can again be done in two modes, corresponding to the two main types of users. Programmers can use INTERPRET syntax to define new lexical and grammatical classes, assign their denotations and insert new lexical tokens in the lexicon. Casual users can obtain the same effect, indirectly, under the control of another built-in grammar for "syntactic training", whose patterns trigger the same semantic actions as those specified by programmers. Of course, this "syntactic training" grammar, as the "ordinary English" one, can not claim to capture *all* semantic actions that a programmer may define. It is sufficient that they capture most of the common uses and definitions, with facilities to describe the limits of their capabilities and guide one into extending these limits when desired.

The syntactic component can thus be extended to handle more natural interactions like the following sample dialogue:

```
System: Please identify yourself.
 c1> I am John Smith.
System: OK, you are the student John Smith.
```

The main additions required to produce the behavior shown above are for data class names and for some abbreviated property names. These will be defined as:

```
%CONCEPT 'person' with trans ← 'PERSON';
%CONCEPT 'student' with trans ← 'STUDENT';

%RELATION 'social insurance number'
                  with trans ← 'si#';
%RELATION 'telephone number'
                  with trans ← 'phone#';
%RELATION 'number' with trans ← 'student#';
```

Our lexicon now contains the l-classes and lexical tokens shown in Figure 3.

Some of these lexical tokens need to be further classified to be correctly used by the built-in English grammar:

```
%PROPER-NOUN 'Mary Smith' with den ← Mary;
%PROPER-NOUN 'John Smith' with den ← John;
%PROPER-NOUN '123456789' with den ← John;
%PROPER-NOUN '234567899' with den ← Mary;
```

These expressions define the corresponding lexical tokens as able to play a subject or object role in an English sentence, according to the Built-in English Grammar shown in Appendix 4.

We also have to extend the built-in English grammar with patterns that relate specifically to the data classes defined in the semantic component of an INTERPRET program (unless they are also built-in for such common data classes as PERSON, but we assume it is not the case here). Some of the most common expressions that could be used to answer the request 'Please identify yourself' are:

'John Smith'
'I am John Smith'
'My name is John Smith'
'123456789'
'My social insurance number is 123456789'
'I live at 37 Purdon Dr., Toronto'
'My phone# is 412-7841'
'I am 009812345'
'I am the student 007812345'
'I am the first year student
from Arts and Science'
'I am the person named John Smith'
etc...

and the list could go on. But much of this apparent diversity can be captured with a few basic patterns:

G-CLASS &I-AM
:= (["I am"] &NOUN-PHRASE);
G-CLASS &MY-IS
:= "My <%PROPERTY> is <%TOKEN>"

A more detailed specification of these grammatical classes is given in *Appendix 3*. The definition of grammatical classes for a given application can be much simplified by taking advantage of built-in grammatical classes. For example, the definition of the specialization of the grammatical class &GET-PERSON illustrates the gains obtained by borrowing grammatical classes from the built-in English grammar: any expression involving lexical tokens satisfying the rules of the built-in grammatical class &NOUN-PHRASE is recognized and decomposed into standard components, and receives an also standard interpretation from the interpretation of the lexical tokens involved.

We have not yet defined the lexical tokens 'named' and 'live at'. This is done as follows:

%ONE-PLACE-MOD 'named' **with** trans← 'name'

%TRANSITIVE-VERB 'live' **with** trans ← 'address'

These definitions then allow for input sentences such as 'I am named ...', 'I live at ...', using the specializations %I-AM-MOD and &I-VERB of &I-PRED, also defined in *Appendix 3*.

Fig.4 -

$$\begin{tikzcd}
 & & \text{\&SENTENCE} & & \\
 & & \text{\&GET-PERSON} \arrow[u, Rightarrow] & & \\
\text{\&I-AM} \arrow[urr, Rightarrow] & & & & \text{\&MY-IS} \arrow[ull, Rightarrow] \\
 & & \text{\&I-PRED} \arrow[uu, Rightarrow] & & \\
 & \text{\&I-AM-MOD} \arrow[ur, Rightarrow] & & \text{\&I-VERB} \arrow[ul, Rightarrow] &
\end{tikzcd}$$

As an example of the analysis of an input sentence matched by the grammatical class &GET-PERSON and its specializations, the input sentence 'I am 123456789' is first matched by the grammatical class &I-AM (see *Appendix 3*), because the input fragment '123456789' matches the specialization &PROPER-NOUN of &NOUN-PHRASE (see *Appendix 4*). This fragment becomes the value of the local variable *person* in &I-AM, with a *den* property specified in &PROPER-NOUN as the result of '123456789'.den.

We then have:

'I am 123456789'.den
= '123456789'.den
= **John**

Depending on the degree of flexibility desired at this level we may want to define other lexical or grammatical classes relating to the recognition of instances of PERSON. For example, 'first year' as a %MOD denoting the property-value pair 'year=1', 'from' as a %ONE-PLACE-MOD standing for 'faculty', etc. Or we may define &GET-PERSON as a specialization of the built-in grammatical class &SENTENCE, to take advantage of built-in transformations that will transform input sentences of the form "I think <&SENTENCE>", "I tell you that <&SENTENCE>", etc. into a recognizable "<&SENTENCE>".

Taking stock of the grammatical classes defined in *Appendix 3*, we have a set of specializations of the grammatical class &GET-PERSON, as depicted in Figure 4. The final mechanism needed is the attachement of this mini-grammar to the class PERSON, such that a request for an instance of this class may use &GET-PERSON to recognize and obtain its answer from an input sentence.

Finally, we need to extend in the following way the definitions of MASTER-SCRIPT given in *Appendix 2* to account for our dialogue example:

property format **on** MASTER-SCRIPT .. person **is**
"Please identify yourself";

property grammar **on** MASTER-SCRIPT .. person **is**
&GET-PERSON;

This means that any expression like 'I am John Smith' has to match the grammatical class &GET-PERSON to be recognized as a valid reference to an instance of the data class PERSON. A request for the value of a property uses the "pattern" attached to the *format* property of the first property to generate an expression to be sent to *user1*. The reply has to match the pattern attached to the *grammar* property to be recognized and accepted. A *grammar* pattern will decompose a successfully matched input expression, and produce a "denotation", i.e. a reference to some "internal" object(s). The denotation of an expression matched by a pattern is given by the value of the property *den* applied on this expression.

VI. Conclusion

This paper described three main ways in which the design of user interfaces can be simplified:

1. we reduced the complexity of the problem by breaking it down in three components; this decomposition results in a unique and novel approach to user interface design which allows the specification of user-oriented syntactic aspects to be postponed to the last step of the design;

2. we showed how to maximize the use of syntactic information already specified in the declaration of data types and programs;

3. part of the design job could then be deferred to a user of the interface.

As mentioned at the beginning, the main goal of this work is to *integrate* various mechanisms found useful in many different approaches to represent and design user interfaces, as opposed to trying to break new grounds along any of these directions. Once we reached a common basis to this effect, we found that we could expand it arbitrarily to follow any direction of current research, yet remained able to compare these different solutions between each other.

In particular, any of the current works on representing and analyzing complex queries can be expressed and integrated in our framework. An extreme example of this capability is the representation in [Pilote 83a] of the syntax and model semantics of of subset of English according to the work of Gazdar [Gazdar 82], which is storming the linguistic world.

The decomposition of a user interface into three components is particularly significant to reduce the effort involved in setting up new user-interfaces for casual users. These interfaces can be designed in three separate steps, taking successively care of the *semantic*, *pragmatic* and *syntactic* aspects of the interface.

Once a "natural" interface has been defined for data and programs represented using the *semantic* features of INTERPRET, an interesting side-effect of our approach is that this interface can then be used to examine and even modify its own structures, since all information underlying this interface is represented in exactly the same way as any other purely semantic information. The impact of such a mechanism remains to be explored but it promises far-reaching results. The feasibility of this approach has already been demonstrated by the users of LIFER [Hendrix *et al.* 79], relying exclusively on synonyms and paraphrases to customize a particular user interface. Again, our goal here is first of all to duplicate such results in a more organized framework. This allowed us in particular in [Pilote 83a] to integrate LIFER-like mechanisms with more linguistically oriented ones such as the work of Gazdar.

Acknowledgements

The author wishes to thank Dr. John Mylopoulos for providing the framework for this research and numerous discussions that led to the results presented in this paper.

Bibliography

[Barr, A. and Feignenbaum, E.A., 79]
Natural Language Understanding, a section of the **Handbook of Artificial Intelligence**, Stanford Computer Science Dept., Rep. No.STAN-CS-79-754 (microfiche, July 1979). Also, printed as a book, Barr & Feignenbaum (eds.), Los Altos, Cal., William Kaufman. Inc. (1981).

[Barron, J.L., 80]
Dialogue Organization and Structure for Interactive Information Systems, Tech. Rep. CSRG-108, Univ. of Toronto, (M.Sc. Thesis, Dept. of Comp.Science, 1980).

[Burton, R.R. and Brown, J.S., 76]
Semantic Grammar: A Technique for Constructing Natural Language Interfaces to Instructional Systems, BBN Report No.3587 (ICAI Rep. No.5), Cambridge, Mass. (May 1977).

[Codd, E.F., Arnold, R.S., Cadiou, J.-M., Chang, C.L. and Roussopoulos, N., 78]
RENDEZ-VOUS Version 1: An Experimental English-Language Query Formulation System for Casual Users of Relational Data Bases, Res. Rep.# RJ2144, IBM Research Lab., San Jose, Calif, (Jan. 1978).

[Gazdar, G., 82]
Phrase Structure Grammar, in **The Nature of Syntactic Representation**, Jacobson, P., Pullum, G.K. (Eds.), Dordrecht, D. Reidel.

[Gordon, M.J.C., 79]
The Denotational Description of Programming Languages - An Introductick n, Springer Verlag.

[Grosz, B., 82]
Transportable Natural-Language Interfaces: problems and techniques, **Proc. of the 20th An.Mtg. of the Assoc. for Computational Linguistics**, University of Toronto, June 82. 46-50.

[Haas, N. and Hendrix, G.G., 80]
An Approach to Acquiring and Applying Knowledge . Tech. Note 227, SRI Intern. (Nov. 80).

[Harris, L.R., 78]
The ROBOT System: Natural Language Processing Applied to Data Base Query, **Proc. ACM Conf. 1978**, 165-172.

[Hendrix, G.G., Sagalowicz, and Sacerdoti, E.D., 79]
Research on Transportable English-Access Media to Distributed and Local Data Bases, Proposal for Research to DARPA No.ECU 79-103, (Nov. 1979), SRI International.

[Mylopoulos, J., Bernstein, P. and Wong, H.K.T., 80]
A Language Facility for the Design of Interactive Database-Intensive Applications, **Trans. on Database Systems**, Vol.5, No.2, June 1980, pp.185-207; also Tech.Rep. CSRG-105, Univ. of Toronto, July 1979.

[Petrick, S.R., 81]
Field Testing the Transformational Question Answering (TQA) System, **Proc. 19th Ann. Mtg. of the ACL**, June 1981, pp. 35-38.

[Pilote, M. 83a]
A Framework for the Design of Linguitic User Interfaces, Ph.D. Thesis, Dept. Computer Science, Univ. of Toronto, CSRG Technical Note #32, Jan. 1983.

[Pilote, M. 83b]
A Programming Language Framework for Designing User Interfaces, **SIGPLAN Notices**, Vol. 18, No. 6, Proc. of the SIGPLAN'83 Symp. on Programming Language Issues in Software Systems, San Francisco, California, June 1983, pp.118-136.

[Plath, W.J., 76]
REQUEST: A Natural Language Question-Answering System, **IBM Journal of Research and Development**, **20**, 4, July 1976, pp. 326-335.

[Robinson, J.J., 82]
DIAGRAM: A Grammar for Dialogues, **Comm. ACM**, Vol.25, No.1, Jan 1982, pp.27-47.

[Shapiro, S.C. & Neal, J.G., 82]
A knowledge engineering approach to natural language understanding, **Proc. of the 20th An.Mtg. of the Assoc. for Computational Linguistics**, University of Toronto, June 82. 136-144.

[Sowa, J.F., 80]
A Conceptual Schema for Knowledge-Based Systems, in **Proc. Workshop on Data Abstraction, DataBases and Conceptual Modeling**, (June 80).

[Tennant, H., 79]
Experience with the Evaluation of Natural Language Question-Answerers, **Proc. 6th Intern. Joint Conf. on Artificial Intelligence**, Tokyo, 874-879.

[Thompson, F. and Thompson, B., 78]
Rapidly Extensible Natural Language, **Proc. of ACM National Conf.**, Washington, D.C., (Dec. 1978), 173-...

[Waltz, D.L., 78]
An English Language Question Answering System for a Large Relational Database, **Comm. ACM**, **21**, No. 7 (July 1978), 526-539.

[Winograd, T., 79]
Beyond Programming Languages, **Comm. ACM**, Vol.22, No.7, (July 79), pp.391-401.

[Wong, H.K.T., 81]
Design and Verification of Interactive Information Systems Using TAXIS, Univ. of Toronto, Tech. Rep. CSRG-129 (April 81). PhD thesis, Univ. of Toronto, Jan. 1983.

[Woods, W.A., 77]
Semantics and Quantification in Natural Language Question Answering, **Advances in Computers**, Vol.17, Yovits, M.C. (ed.), Academic Press (1978), 2-88. Also Report No.3687, Bolt Beranek and Newman (Nov. 77).

[Zisman, M.D., 77]
Representation, Specification, and Automation of Office Procedures, Ph.D. Thesis, Dept. of Decision Science, The Wharton School, Univ. of Penn., Sept. 77.

Appendix 1

Definitions of semantic data classes

```
DATA-CLASS PERSON with
    keys person: (si#);
    characteristics
        si#: %DIGIT*9;
    attributes
        address: %Token;
        name: %Token;
        phone#: &Phone#;
    end

DATA-CLASS STUDENT isa PERSON with
    keys student: (student#);
    characteristics
        student#: %DIGIT*9;
    attributes
        faculty: %Token;
        year: {1..7};
        status: {'part-time', 'full-time'};
    end
```

In addition to the above "type" declarations, we define the following "data" as part of the extension of these classes:

```
    STUDENT John with
        name ← 'John Smith',
        si# ← '123456789',
        address ← '37 Purdon Dr., Toronto',
        phone# ← '412-7846',
        student# ← '007812345',
        faculty ← 'Arts and Science',
        year ← 1;

    PERSON Mary with
        name ← 'Mary Smith',
        si# ← '234567899',
        address ← '37 Purdon Dr., Toronto',
        phone# ← '412-7841';
```

Notation

- the class PERSON is an *instance* of the ***metaclass* DATA-CLASS;**
- the property "name" is an instance of the ***property category properties*;**
- "name" takes an instance of PERSON as argument to produce a lexical token;
- the class STUDENT is a subclass of PERSON;
- the property *courses* is multi-valued;
- {1..7} is a *range* class of all integers between 100 and 2999;

Appendix 2

Script definition

```
SCRIPT-CLASS
      MASTER-SCRIPT: (start, user → end) with
  states
      start, end: STATE;
      gotPerson isa start: STATE with conditions
             person in STUDENT exc waitPerson;
      gotStudent isa gotPerson with conditions
             person in STUDENT;
      waitPerson: EXCEPTION-STATE;
      moreThanOne
         isa waitPerson: EXCEPTION-STATE;
 locals
      user: TERMINAL-CODE;
      person: PERSON;

 transitions

      GetPerson: (waitPerson → gotPerson) with
          actions
             a1: take(user, person);
         end
      StudId: (gotStudent → end) with
         actions
             a1: instantiate
                     STUDENT-SCRIPT(user);
             a2: give(user, "OK. You
                  are the student <user.name>");
         end

      GetCancel: (waitPerson → end) with
           actions
             a1: take(user, "Bye");
             a2: give(user, "OK. Bye.");
           end
end MASTER-SCRIPT
```

Notation

An instance of MASTER-SCRIPT represents a function from a *state* which is named 'start', to produce a new state named 'end'. This script also takes as another argument an instance of TERMINAL-CODE, which identifies an external entity (user or another script) which will send messages to the script and to which replies will be sent.

As in Petri nets, any successful transition deactivates its argument state(s) and activates its target state(s). As soon as one transition activates *end*, the script terminates.

Appendix 3

User-defined grammatical classes

```
GRAMMATICAL-CLASS &GET-PERSON isa &SENTENCE
    := &I-AM | &MY-IS | &I-PRED;

GRAMMATICAL-CLASS I-AM := "[ I am ] <person>" with
    locals
        person: &NOUN-PHRASE;
        den: PERSON default person.den;
    conditions
        c1: NP-ISA-RELATED(person.head, PERSON);
    end

GRAMMATICAL-CLASS &MY-IS :="My <prop> is <this>" with
    locals
        prop: %RELATION;
        this: &NOUN-PHRASE;
        trans: "the person whose <prop.trans> is <this.trans>";
    conditions
        c1: ISA-RELATED(prop.den.subject, PERSON)
            exc NOT-A-PROP-OF-A-PERSON;
    end

GRAMMATICAL-CLASS &I-PRED := &I-AM-MOD | &I-VERB with
    locals
        trans: "the person whose <pred.trans> is <obj.trans>";
        obj: &NOUN-PHRASE;
    end

GRAMMATICAL-CLASS &I-AM-MOD isa &I-PRED
            := "I am <pred> <obj>" with
    locals
        pred: &ONE-PLACE-MOD;
    end

GRAMMATICAL-CLASS &I-VERB isa &I-PRED
            := "I <pred> <obj>" with
    locals
        pred: %TRANSITIVE-VERB;
    end
```

Assuming that

```
TRANSACTION-CLASS NP-ISA-RELATED: (np, s-class → res) with
    locals
        s-class: ANY-CLASS;
        np: &NOUN-PHRASE;
        res: BOOLEAN default false.
    actions
        a1: if np.head ≠ nothing then
                if np.head in %PROPER-NOUN then
                    if np.head.den in s-class then
                        res ← true;
              else if np.head in %COMMON-NOUN then
                if ISA-RELATED(np.head.den, s-class) then
                        res ← true;
    end

TRANSACTION-CLASS ISA-RELATED: (s-class1, s-class2 → res) with
    locals
        s-class1, s-class2: ANY-CLASS;
        res: BOOLEAN default false;
    actions
        a1: if (s-class1 isa s-class2) or
                (s-class2 isa s-class1) then
                    res ← true;
    end
```

Appendix 4

Built in grammatical classes used in Appendix 3

```
GRAMMATICAL-CLASS &SENTENCE with
    locals
        subject: &NOUN-PHRASE;
        pred: &VERB-PHRASE;
    end

GRAMMATICAL-CLASS &NOUN-PHRASE

GRAMMATICAL-CLASS &VERB-PHRASE with
    locals
        verb: %VERB | &VERB-PHRASE;
        object: &NOUN-PHRASE | &PREP-PHRASE;
    end

GRAMMATICAL-CLASS &PREP-PHRASE with
    locals
        prep: %PREP;
        subject: &NOUN-PHRASE;
    end

GRAMMATICAL-CLASS &PROPER-NOUN isa &NOUN-PHRASE
    := head with
     locals
        head: %PROPER-NOUN;
        trans: "<head.trans>";
     end

GRAMMATICAL-CLASS &NOUN-PHRASE1 isa &NOUN-PHRASE
    := "the <head> whose <prop> is <np>" with
      locals
        head: %CLASS;
        prop: %PROPERTY;
        np: &NOUN-PHRASE;
        trans: "the <head.trans>
                with <prop.trans> = <np.trans>";
      end

GRAMMATICAL-CLASS &NOUN-PHRASE2 isa &NOUN-PHRASE
    := "the <head> <ap>" with
      locals
        head: %CLASS;
        ap: %ADJECTIVE;
        trans: "the <head.trans> with <ap.trans>";
      end

GRAMMATICAL-CLASS &NOUN-PHRASE3 isa &NOUN-PHRASE
    := "the <head> who <vp>" with
      locals
        head: %CLASS;
        vp: &VERB-PHRASE;
        trans: "the <head.trans> with <vp.trans>";
      end

GRAMMATICAL-CLASS &I-TRANSFORM isa &SENTENCE with
    locals
        subj: "I";
        trans: "<pred.verb> <user> <pred.object>";
    end
```