Archetype-oriented CHI
A Formal Approach to User Friendliness

J.N.Oliveira
F.M.Martins

C.C.E.S.
Universidade do Minho
Rua D.Pedro V, 88-3°
4700 Braga
Portugal
Tlx : 32135 UMINHO P

ABSTRACT

Theoretical computer science has the aim of formalizing previous empirical, innovative creations in computing. The possibility of automatically constructing instances of such creations is dependant on the existence of formal models. This is particularly true in computing since computers do not manipulate informal models of problem-solutions.
This paper describes an exercise in using *constructive* and *algebraic* specifications in the formalization of some aspects of computer-human interaction (CHI). Formal specification can not only contribute to a better understanding of CHI, but also provide the almost non-existent link between CHI and Software Engineering.
In this paper, from a modest analysis of user behaviour, we build mathematical models which lead to a formalization of the "standard" **assisted-user-interface** (AUI). These formalisms are applied to the specification of **ASSIST**, a mechanical generator of *assisted-user-interfaces* which matches with command-line parser-generators technology. Finally, we show how the **assist-paradigm** can be implicitly combined with formally specified software modules, in the sense that each formal specification itself contains enough information for ASSIST to generate the relevant parts of the AUI.

1. Introduction.

In every scientific field, after the event of creation, the typical following steps are: empirical experimentation, development of theories and automation. A role of theoretical computer science is to provide formal models which can be used in the automation process. The usefulness of formalization is two-fold : it leads to a better understanding of particular bodies of knowledge and gives the possibility of automatic replication. Informal knowledge is not amenable to mechanization since machines do not 'understand' informal models of problem-solutions.

In computer science the number of existing and applicable theories is still insufficient, and whenever a new topic emerges, appropriate models must be built up through experience. Since programming languages are cumbersome for reasoning, one important problem is that of notation. Therefore, formal specification languages have been used as alternatives to support such reasoning.

This paper describes an exercise in using both *constructive* [Jo86] and *algebraic* [G*77] specifications in the formalization of some aspects of computer-human interaction, particularly those related with the **user interface**.

1.1. Background.

Recent proliferation of personal workstations in the computer market has entailed the widespread of the so-called *user-friendly* interactive devices. This has made room for yet more problems with compatibility and consistency in computer engineering[1]. Once again, software-engineers have failed to provide the promised easy-to-learn interface whereby no manuals would be necessary to read, as pointed out in [Th86].

Of course, consistent interaction facilities are nowadays vital to the widespread use of computer systems, and CHI-standards are urgently required. However, software-designers should properly 'understand' the interaction phenomena before defining implementation standards. Informal software-standards are unprecise and hard to formalize "a posteriori" (take the GKS example [Du87]).

This paper shows how formal specification can not only make some aspects of CHI more understandable, but also provide a way of relating these aspects to other phenomena in computing science.

1.2. Paper Structure.

Our starting-point for theoretical CHI is a simplistic analysis of user behaviour (section 2). However simplistic, it is sufficient for building mathematical models which lead to a formalization of the **standard assisted-interface** (section 3-5). In section 6 these formalisms are applied to the specification of ASSIST, an automatic generator of AUI's which may coexist with standard command-line parsing technology. In section 7 we show how abstract user-interfaces can be directly derived from the formal specification of applications, and then input to ASSIST for the generation of concrete ones.

[1] Suggestion: Have a look to your favourite PC-software.

2. On the Nature of Users.

In sharp contrast with computers, users are informal living beings who are temperamental, unpredictable, vague, unprecise and unreliable (at least!). CHI is bound to take these facts into account.

Unpredictability has to do with *non-deterministic behaviour*. Vagueness means *incomplete information*, so inference mechanisms must be provided to cope with hierarchical levels of incompleteness. Unreliability suggests that users should be *assisted by syntax-direction* where possible. A formal framework is described below where unpredictability, vagueness and unreliability are modeled in terms of simple formal notions such as *non-deterministic choice* and *archetype* [MO86].

3. Interaction versus Non-determinism.

A simple (although primitive) model for user behaviour wrt. machines, can be found in the mathematical notion of a **non-deterministic choice** [MO86].

Suppose that an alphabet **A** of acceptable symbols is meaningful in a given linguistic context, eg. a command in **A** = { H,J,K,L } should be provided as an argument for a function **f** : **A** → **B**. For example **A** may be exactly the set of acceptable symbols of a non-terminal symbol at the top of a parsing-stack context.

The typical approach in compiling is to define an error-recovery routine which is invoked if a symbol not in **A** is given. An interpreter for the same language may save error-recovery by enforcing the user to input a symbol in **A**[2]. Abstractly, this can be specified by [3]:

```
let $x \in$ A in f(x)
```

a standard implementation of which is :

```
let $x =$ any(A) in f(x)

where

any(S) $\hat{=}$ let $x \in$ Keyboard in
                  if $x \in S$ then x else any(S)
pre-any(S) $\hat{=}$ $S \neq \varnothing$
```

The mathematical fact that **any**(S) may not terminate, models the operational behaviour of the interpreter which will loop forever if no value in S is ever provided. If S is a finitely enumerated set, **any**(S) can be further refined into a **menu-choice** :

[2] This possibility arises from the fact that an interpreter consumes its *input stream* in a "lazy evaluation" mode [He84] in contrast with the "eager evaluation" mode of compilers. This is a subtle formalization of the relationship between the *interactive* and the corresponding *batch* processing of the same job.

[3] For mathematical formulae in this paper we adopt the VDM [Jo86] notation.

```
any(S) ≙ any_loop( buildmenu(S), default )

any_loop( menu, defch ) ≙
        let k ∈ {go, down} in
            if k = go then menu(defch) else any_loop( menu, 1+ (defch mod (# dom menu)))

buildmenu(S) ≙
            if S = ∅ then []
                        else let s ∈ S in  [ #S → s ] ⊗ buildmenu( S - {s} )
```

Note that this refinement step adds an extra design decision: the selection of a *default* menu-choice, **menu(defch)** in S, corresponding to the initial option **go**.

For infinite sets S, this *menu-refinement* is not feasible since **buildmenu** would not terminate. However, if S is ZF-abstractable,

$$S = \{ y \in \text{Keyboard} \mid p(y) \}$$

then the evaluation of **p(y)** can formally substitute the test $x \in S$, in the definition of **any**. The corresponding action, at implementation level, is to invoke a input-validation routine.

4. Interaction versus Unreliability.

As discussed above, man-machine communication is bound to be established by *formal command languages*. Since users think informally and speak *natural languages*, there is a mismatch between the *fixed syntax* of the machine command-language and the user's *linguistic freedom*. Therefore, users tend to forget the details of such a fixed-syntax and erroneous interactive sessions appear.

A standard approach to similar problems in editing software, is the use of *syntax-directed* editors.

This strategy is clearly applicable to interaction, which is formally equivalent to editing a command-language ("*lazy*") stream. In fact, it is interesting to note that this leads to what is informally known as the **assisted-mode** for user interfaces. For the sake of generality, [MO86] uses algebraic language description [G*77], which is an algebraic approach to abstract-syntax [BJ82].

For example, the following fragment of a concrete BNF for a simplified command-language syntax,

```
<CLANG> ::= <FILE-COMMAND> | <DIRECTORY-COMMAND> | ...
<FILE-COMMAND> ::= COPY <FILESPEC> <PATH> | DEL <FILESPEC>
<DIRECTORY-COMMAND> ::= DIR <PATH> | ...
<PATH> ::= ...
```

is abstractable in terms of the *heterogeneous* algebraic signature $\Sigma = \langle S, F \rangle$, whose *sorts* are exactly the non-terminal symbols of the BNF above, and F involves abstract operation-symbols such as :

```
op1 : <FILE-COMMAND>       → <CLANG>
op2 : <DIRECTORY-COMMAND> → <CLANG>
op3 : <FILESPEC> <PATH>    → <FILE-COMMAND>
......
```

Finally, the specification of an abstract **assisted**-interface is readily available from the *editor-scheme* of [MO86],

```
assist(Σ) ≙ let s ∈ S in buildterm(s,Σ)
        % user chooses the sort (non-terminal) of the input command %

buildterm(s,Σ) ≙
                let op ∈ ops?(s,Σ) in
        % choice of an operator of co-arity s, ie. BNF production for s %.
                        let a = arity(op,Σ) in
                                mk-Term( op, <buildterm(s1,Σ) | s1 ← a> )
```

in which the recursive calls to *buildterm* terminate wherever constant operators **o : → s** are selected.

The usual menu-oriented appearance of *assisted-interfaces*, derives from systematic refinement of non-deterministic choices such as let s ∈ S. Note that assist is a structural extension of **any**.

5. Interaction versus Incompleteness.

The user-interface standard formalized so far, is not yet adequately tuned to user-psychology. This is because *vagueness* has not been taken into account. This aspect of user-behaviour is, perhaps, the most relevant. In fact, users do not usually have a comprehensive knowledge of the jobs they want machines to do and even when syntax-assisted, they do not grasp all the *semantic* details of the command-language. Humans *vague idea* of a concrete command can be regarded as a *precise* idea of a generic **class** of commands. This calls for the notion of **archetype**, previously developed in [MO85, MO86] in a computer-graphics environment[4].

5.1. Command-Archetypes.

Command-archetypes are *generic-commands* whose formalization is immediate in an algebraic setting, via the notion of a *higher-order Σ-term* (cf. [MO86] for details). In the signature above, for example, the lambda-expression

λx . op3(x,b:)

equivalent to the BNF-expression

λx . COPY x b:

denotes the archetype (ie. *class*) of all "copy ... to drive b:"-commands. The informal notion of vagueness is captured by uninstantiated variable-names. Instantiation of higher-order terms is the operation provided for supplying further information. For example the instantiation,

['a:z.bat'/x]'COPY x b:' = 'COPY a:z.bat b:'

leads to a term denoting the class of all commands which "copy a batch file from drive a: to drive

[4] The term *archetype* was suggested by Plato's reminiscence theory, which is curiously related to the formalisms described in [MO85, MO86].

b:" (neatly a *subclass* of the original).

5.2. Archetype-oriented CHI.

Among the formal aspects of this notion of an archetype, one is particularly relevant to CHI: variables can be substituted in any order (ie. λ-expression "currying" is associative and commutative). This is the basis of flexible command-parameter setting such as, for example, in the Lotus 1-2-3[©] software-package user-interface.

How can command-archetypes be executed ? The answer amounts to defining automatic instantiation mechanisms for all archetype variables. Possible schemes are :

a) Non-deterministic choice among the set of all possible 'substitutions';

b) Interactive choice, which realizes non-deterministic choice ;

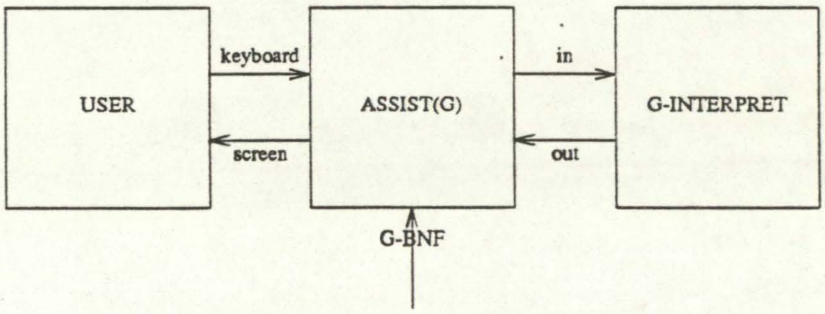
c) Supplying *default-data* (ie. a *default substitution* per each Σ-'context').

Again the Lotus 1-2-3[©] user-interface uses a combination of schemes (b) and (c), together with 'flexible currying'.

6. The Specification of ASSIST.

Given the above presented formalisms, it was easy to write a concise VDM specification of the kernel of ASSIST [P*87]. A *metoo* [He85] prototype of this specification is currently being exercised.

ASSIST is parameterized wrt. the BNF-description of a command-language G. When invoked, ASSIST(G) becomes a node "grafted" between the user and the G-interpreter, which can cancelled via the "!" ("bang") meta-command and re-invoked via the "A" ("assist) meta-command.



After a G-command being synthesized by ASSIST(G), it is forward along channel **in**, which is thus a *lazy G-stream*. This is triggered by the "G"(o) meta-command, which replaces all (if any) uninstantiated variables in the command-archetype by default substitutions[5], and sends the fully-instantiated command to the G-interpreter.

In summary, the *keyboard-channel* is of type lazy-stream of 'meta-commands' (basically a simple language for menu navigation) which (through ASSIST) becomes a lazy-stream of

© Copyright 1985 by Lotus Development Corporation.

[5] These are specified in the first production of each G-nonterminal N. However, the user-choice in the last unfolding of N, in a given context, is kept as a local default N-substitution. So, user most-recently-created archetypes are "stronger" than default values. This *locality of context* is a relevant aspect of the archetype paradigm [MO86].

syntactically correct G-commands, sent forward to the G-interpreter.

The usefulness of ASSIST should be equated in terms of not only already existing software, but also software to be designed. In the former case ASSIST requires only a formal syntax description of the application's command-language. It is more interesting to consider the later case, in which no formal command-language previously exists.

7. Automatic Generation from Formal Specifications.

Suppose that a formal specification methodology is being used in the design of a package. It is shown next how interactive (or batch) communication with the package can be automatically inferred from the specification itself. Consider the following excerpt of a VDM specification of a bank-account-management system.

INIT
State : Bank_Account_DataBase
post-**Init**(db,db') = ...

BALANCE
State : Bank_Account_DataBase
Type : Account_Nr → Amount
pre-**Balance** (db,a) = ...
post-**Balance** (db,a,db',c') = ...

OPEN
State : Bank_Account_DataBase
Type : Account_Nr Holder_Id →
pre-**Open** (db,a,h) = ...
post-**Open** (db,a,h,db') = ...

DEPOSIT
State : Bank_Account_DataBase
Type : Account_Nr Amount →
pre-**Deposit** (db,a,c) = ...
post-**Deposit** (db,a,c,db') = ...

From the operation names and operation type-arities the following abstract syntax is defined,

```
<COMMAND> ::= INIT |
        BALANCE <Account_Nr> |
        OPEN <Account_Nr> <Holder_Id> |
        DEPOSIT <Account_Nr> <Amount> |
        ...
```

which should be completed with the abstract-syntax provided for the involved objects, eg. Account_Nr. For parsing reasons, this syntax might be "sugared" into the form,

```
<COMMAND> ::= INITIALIZE |
        BALANCE FOR <Account_Nr> |
        OPEN <Account_Nr> FOR <Holder_Id> |
        DEPOSIT <Amount> INTO <Account_Nr> |
        ...
```

Such syntax-details are no longer required for the user-interface, since ASSIST will automatically provide a <COMMAND>-directed interface. Generally speaking, behind any VDM model-specification, of the form <Abstract Syntax, Operations>, there is, implicitly associated, a family of command-languages such as, for example,

```
<COMMAND> ::= <OP1CMD> | <OP2CMD> | ...
<OP1CMD> ::= OP1 <A1> <A2> ... <An>
......
<A1> ::= ...
<A2> ::= ...
 ......
```

At the refinement phase, the first level of semantic-checking can also be inferred from the specification, by running data-type invariants and pre-conditions where appropriate (this may be called *design-time* type checking). Running data-type invariants is required only for keyboard-input operations, since the VDM "proof obligations" discipline forces the software designer to supply formal arguments that all operations preserve the invariants. Post-conditions can be regarded as the specification of the semantic-actions whose code will be called by a <COMMAND>-parser.

8. Conclusions and Further Work.

This paper has shown how mathematical specification techniques can be used to formalize some basic aspects of CHI, starting from informal analysis of user behaviour. A more elaborate user-model than the one used here would naturally lead to a richer formalization. Our point was, however, to show how *mathematical* models can effectively be built from such *informal* models.

This work also shows that formal specification makes for better understanding of computers and systems, facilitating the establishment of relationships among apparently disjoint areas of computing. For example, we have shown that the standard *assisted*-interface is a mere instantiation of syntax-directed editing. Furthermore, it has been very rewarding to realize that our formal model of user-interface embodied the concept of an **archetype**, previously developed in another context. Note that our notion of 'generic command' is different from the one of [RM85], where generic commands are "recognized in all contexts of a computer system" and are "extremely general actions which make minimal assumptions about their objects". In algebraic terms, they correspond to *overloaded* signature operators rather than *derived* ones [G* 76].

We have also shown how these formalisms are applied to the specification of ASSIST, a mechanical generator of *assisted*-user-interfaces (AUI), how *model-oriented* and *property-oriented* specification styles may coexist and, the last but not the least, how an interaction-language can be formally derived from a formal specification.

In the future, we will need to formalize the "output-language" as well, ie. the one implicit in each operation type-definition co-arity. This has to do with ASSIST's behaviour towards the **out**-stream (cf. picture above) which, at the moment, is copied onto the screen.

References

[BJ82]
Bjorner D. and Jones C.B., *Formal Specification and Software Development,* Prentice-Hall International, Series in Comp. Science, Hoare C.A.R. (Ed.), 1982.

[Du87]
Duce D.A., *Formal Specification of Graphics Software,* Int. Report Rutherford Appleton Laboratory, Chilton, Didcot, England, 1987.

[G*76]
Goguen J., Thatcher J. and Wagner E., *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types,* Tech. Rep. RC 6487, IBM T.J.Watson Research Center, Oct. 1976.

[G*77]
Goguen J.A., Thatcher J.W., Wagner E.W. & Wright J.B., *Initial Algebra Semantics and Continuous Algebras,* JACM 24(1), 68-95, Jan. 1977.

[He84]
Henderson P., *Communicating Functional Programs,* Internal Report FPN-8, Dep. Comp. Science, University of Stirling (Scotland), Dec. 1984.

[He85]
Henderson P., *"me too" - A Language for Software Specification and Model Building - Preliminary Report,* Internal Report FPN-9, Dep. Comp. Science, University of Stirling (Scotland), 1985.

[Jo86]
Jones C.B., *Systematic Software Development Using VDM,* Prentice-Hall International, Series in Computer Science, Hoare C.A.R. (Ed.), 1986.

[MO85]
Martins F.M. and Oliveira J.N., *Graphics Programming with Archetypes - A Preliminary Study,* Proceedings of the EUROGRAPHICS'85 Conference, 401-412, Sept. 1985, Nice, France, Nort-Holland Ed., 1985.

[MO86]
Martins F.M. and Oliveira J.N., *On the Specification of Archetype-Oriented Graphics Editors,* EUROGRAPHICS'86 Special Session, Aug. 1986, Lisbon , Portugal, Int.Rep. CCES:FMM-JNO/R2-86, Univ. of Minho, Braga, Portugal, August 1986.

[P*87]
Pinto A., Pacheco O. and Magalhaes P., *The Formal Specification of ASSIST,* Int. Rep. CCES:OP1/R1-87, Univ. of Minho, Braga, Portugal, May 1987.

[RM85]
Rosenberg J.K. and Moran T.P., *Generic Commands,* Human Computer Interaction-INTERACT'84, B. Shakel (Ed.), Elsevier Science Publishers B.V. (North Holland), 245-249, IFIP 1985.

[Th86]
Thimbleby H., *Ease of Use - The Ultimate Deception,* People and Computers: Designing for Usability, Proc. of the Second Conf. of the BCS-HCISG, Cambridge University Press, 1986.