

# Semi-Automatic Design and Prototyping of Adaptive User Interfaces

**F. Mario Martins**

Universidade do Minho  
Dept. de Informatica/INESC  
Campus de Gualtar  
4710 Braga - Portugal  
+351 253 604470  
fmm@di.uminho.pt

## ABSTRACT

This paper presents a software environment, GAIA, for the semi-automatic generation of self-adaptive user interfaces for API-based applications. Firstly, a characterization of the design space for the adaptive user interfaces the system is able to generate is discussed. System/s characteristics like the adapted constituents of the UI, the system/s degree of intelligence, the timing and information for adaptation, the adaptation method and how it models the user are defined.

The architecture of the system is then presented, and the two main modules, Adaptive Control Module (ACM) and Interface Control Module (ICM) discussed in detail. The module ACM contains all the relevant facts and knowledge about the user and its interaction history that are needed to infer adaptations. This is the intelligent, adaptive module. The module ICM receives control information from the module ACM and is responsible for updating the "look" and/or the "feel" of the UI accordingly .

Finally, we stress the fact that, being based on prototyping and iterative design techniques, the presented system may also be used as a valuable analytical tool,

allowing for the iterative design of adaptive user interfaces which may, eventually, be converted to well adapted (or customized) ones by step-by-step tailoring to the end-users.

## Keywords

Adaptive User Interfaces, Automatic Generation, Model of the User, Knowledge, AUI prototyping.

## INTRODUCTION

Adaptive User Interfaces (AUIs) differ from *adapted* (or customized) and *adaptable* (or customizable) UIs because they may change automatically (without user intervention) at real interaction time. Therefore, they are sometimes called *dynamic*.

The aiming of the possible changes of these UIs is to try to improve the usability of the interactive systems, by taking into consideration, at any time, an inferred view of the user/s knowledge of the interaction process, of the user/s individual requirements and of the user/s goals.

Research on AUIs has been increasing and despite the well known complexity of their design and implementation, AUIs are still

seen as a promising investment in order to deal with the always increasing complexity of UIs. Being very complex, the design of AUIs must be supported by a set of tools tailored to their iterative design and evaluation.

In this paper a software environment based on a set of tools that support the iterative design of AUIs is presented. The different dimensions for adaptation are firstly discussed. Then we point out the dimensions for adaptation that were taken into account in the system we developed. The internal architecture of the GAIA system is then presented. A particular attention is devoted to the module ACM which not only contains all the relevant adaptation knowledge but also implements the adopted adaptation algorithm.

Finally, we stress the fact that despite aiming at the design of AUIs, but aiming the design of adapted UIs is shown.

## **GAIA: AUTOMATIC GENERATOR OF AUIS**

### **GAIA: Adaptation Dimensions**

First of all, regarding the definition of tasks and agents in the adaptation process, it was assumed in GAIA that the initiative, proposal and execution of the adaptation should be a task of the system. However, the final decision is user-bound. This strategy appears in systems like POISE [4] and PODIUM [12] and is defined in [8] as User Controlled Self-Adaptation. Most of the adaptation dimensions of GAIA presented here follow the taxonomic works on adaptive UIs presented in [5, 8].

Concerning the adapted constituents, GAIA appears as a very complete, and complex, system. Syntactical error correction is provided, mainly based on the interaction history. Active help is provided, depending on the system/s inferred skill of

the user. Presentation input/output adjustments are provided. Incremental access to even more sophisticated application functionalities is implemented. Task analysis (on the basis of recognized event sequences or command sequences) is also implemented, allowing the system to propose auto-generated macros to the user.

Regarding the information considered for the adaptation control process, GAIA disregards any possible information about UI ergonomics, a recognized but common drawback acceptable due to the complexity involved, but takes into consideration several relevant data about each individual user of the application. Information and knowledge about each system/s user interactive characteristics, behaviour, skills, knowledge, etc., is gathered and treated by GAIA.

Concerning dialogue control of the interaction, a "dialogue dominant control strategy" [6] is assumed in GAIA, the application being abstracted away as a set of operations to be invoked through a well defined API. Therefore, the UI is only concerned with the establishment of the easiest but correct dialogue sequence needed to drive the well defined application semantics.

Considering the formalisms used to model and represent dialogues within GAIA, formally, dialogues are modelled as transition nets, being however internally represented as Extended Definite Clause Grammars (EDCG) [11] which are naturally implemented in the Prolog [13] process that is the final implementation of the ACM module.

GAIA also offers two different timings for adaptation, both under user confirmation: during the interactive session and after the interactive session.

These are the main adaptation dimensions of GAIA. In the following sections we will

explain how they are achieved and how they are really implemented.

### GAIA: Architecture

GAIA is functionally composed of two modules, ACM and ICM, and communicates with the application, as presented in figure 1.

Because the two modules have different functionalities and responsibilities within the system, they were implemented using different techniques and tools. In order to make them really independent of each other, and also to increase the degree of modularity of the system, the two modules run in two different processes and communicate using a specific, high-level protocol.

The module ACM runs as a SICStus Prolog [3] process. The module ICM is a C process which is an X-Windows [7] (with OSF/Motif [10] window-manager) client. User actions are transformed by ICM into events that are sent to ACM. ACM handles the events and derives, if needed, the corresponding output events to be sent to ICM in order to consistently update the UI presentation and behaviour.

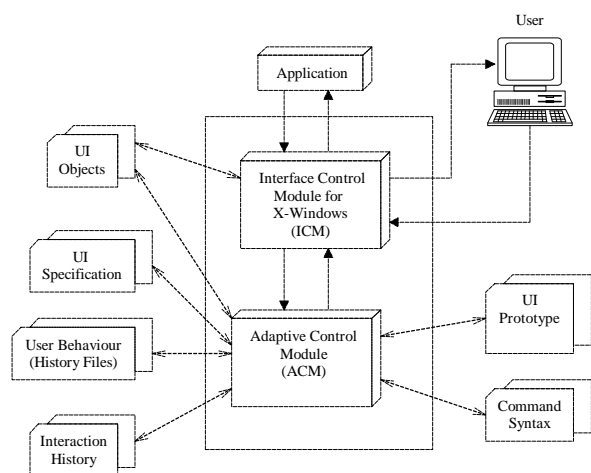


Figure 1.- GAIA/s Architecture

Although communication with the application is made by the ICM module, it

is the ACM module that generates the events that trigger the calls to application commands. The application, in turn, will send back events representing the results of the command invocation.

Both modules have their relevant information stored in several files. Most of these files, like the command-syntax file, the user interface prototype, the history of interaction, and the user/s behaviour file, store information specified by using high-level specification languages. Some examples of such specifications will be given later in the paper.

The following sections present the internal architecture of each module, their functionality, the protocol for module communication implemented and the several specification notations that were defined.

### Module ICM

The module ICM deals with the actual "look" and "feel" of the UI, as well as with the reception of the input events. In general, for each user and each particular application the ICM module starts by presenting a "default" UI previously specified. The typical UI layout built by ICM contains menus, buttons, a command-line window, an application dialogue window and a help display window. Two buttons are always presented: the HELP and the SIGNAL buttons.

The HELP button, when activated by the user, generates an help-event that is context dependent and triggers the help message that is appropriate in such context. The SIGNAL button is used by the adaptive system to notify the user that some message concerning some adaptation decision was sent to him by the system.

Each application command made accessible to the user by the adaptive system has associated a dialogue box with the set of

fields related with such a command arguments, where the argument types are taken into account for input validation.

A specific language for specifying this default UI was built which is translated into OSF/Motif widgets. The language defines four types of objects, menus, buttons, commands and arguments, and uses resources to further specify each object/s particular properties. For instance, objects of class argument have a type resource with twelve different values, for instance, numeric scale, choice\_1\_in\_N, etc.

Besides the interactive presentational objects, the UI specification also contains information about dialogue control (behaviour) and command language syntax. The UI specification is in fact divided into three different sections, namely: Objects, Behaviour and Syntax.

In the section Objects the different instances of the existing object classes are defined following the syntactical rules,

*Object = Class.*

for their class definition, and

*Object : Resource := Value*

for their specific characterization. For instance, to define a given command with two arguments one should produce the following (generic) specification,

*command1 = command*  
*command1: label := /Command 1/.*  
*command1: arguments := [arg1,*  
*arg2, arg5].*  
*command1: mnemonic := /com1/.*

A simple and usual command like *save* would be specified as,

*save = command*  
*save: label := /SAVE/.*  
*save: mnemonic := /S/.*

The section behaviour defines the state transition rules that represent the automaton that recognizes the interaction. Basically two kinds of rules are accepted. Firstly, the rules that will associate a set of active interaction objects to each transition state, which have the general form,

*State  $\rightarrow$  List of Active Objects*

Secondly, the rules that specify the transitions, which have the generic form,

*State1: Command sequence @ State2*

Finally, a set of declarations that are very important for the correct, context sensitive behaviour of the system, which are the declarations of the command pre-conditions, using the generic form,

*pre-condition(Command Name) :-*  
*Function call.*

Naturally, only commands that in a given context have their pre-conditions evaluate to TRUE are available to the user and may be selected. This implements real context-sensitive UIs.

The section Syntax specifies the correct syntax of the command language, needed at UI level for validation and for helping the user in constructing correct invocations to the application commands. An extension of the Prolog formalism called Definite Clause Grammars (DCG/s) [11] was used. However, being too technical, we will not present here the formalism which may be found elsewhere [9].

## **Module ACM**

The module ACM contains the intelligent kernel of the system, and has the internal architecture presented in figure 2.

The ACM module is responsible for the UI management and control, including the interaction between the user and the application and the analysis of the user/s

interactive behaviour. Therefore, ACM is composed of both the kernel for communication and control of the UI, and a knowledge base which gathers information about user behaviour, user interactive profile, actual UI look, interaction history, etc.

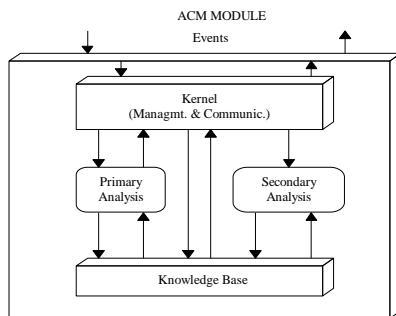


Figure 2.- ACM Module

The kernel of the ACM module is responsible for establishing the communication between the ICM module and the application. It also manages the user interactive events like menu selection, button activation and command input.

The primary analysis subcomponent does the analysis of the user behaviour during the session, being mainly aware to sequences of  $n$  events that may indicate some kind of usual, well defined behaviour from the user. The analysis of the sequence of the last  $m$  commands is also made at this level, willing to infer common sequences that might be synthesized in a macro and also trying to detect systematic syntactical errors and to record and suggest a possible user-specific command alias.

The secondary analysis component is responsible for the analysis of user behaviour throughout, and at the end, of an entire session. The decisions made at this level include the determination (by inference) of the adequate level of help, the inferred level of expertise of the user, the adequate mode of operation (menus or commands), and new application

functionality the user is now supposed to be able to correctly access.

A working session is, in practice, seen as an event or command sequence. However, it is not practical to analyse all the sequences obtained so far. Therefore the analysis mechanism will elaborate on "temporal windows" which correspond to finite sequences of  $n$  events or  $m$  commands, in particular the last  $m$  executed commands.

Each interactive session with a user is completely recorded in an event basis, by storing in appropriate files Prolog facts of the form

*input\_event([event\_code, parameters])*

The analysis mechanism will use a graph of commands (cf. figure 3) and will try to derive command sentences from such input command sequences. As valid sentences are input by the user, the graph will evolve, allowing for, after a certain level, the derivation of possible "macros" which are suggested to the user. Aliases are also suggested.

An automaton was built in order to enable the derivation, or prediction, of user/s behaviour, by associating weights to each transition. One can reach maximum or minimum weights, in the last case leading to the inhibition of such transitions. The automaton is dynamically changeable in its states and transitions as a result of the instructions for adaptation.

Each working section is recorded using a sequence of events, after performing an analysis and a filtering of the most relevant events considering the user interaction history. This user interactive behaviour history is grouped in *temporal zones* properly identified. A garbage collector deals with the periodic removal of some history fractions considered irrelevant because they belong to ancient and already analysed temporal zones.

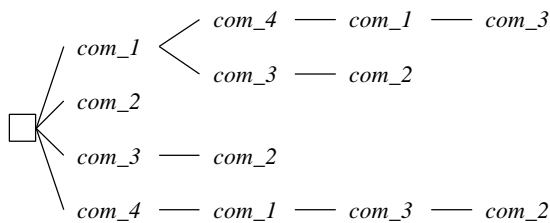


Figure 3.- The command input graph

The ACM module/s knowledge base contains a set of facts that represent the actual system/s model of the user, that is, what is the inferred knowledge the adaptive system has of each particular user, namely, the degree of expertise, the level of help supposedly needed, the user/s most typical behaviour, the user/s most common syntactical errors, etc.

A set of rules (*if-then* clauses) represent the crucial information to be used by the Prolog inference mechanism to implement the adaptation algorithm. The facts that are used to represent the actual information about the user are the sources for rule application. The consequents of the application of the rules over the facts are thus the system/s adaptation decisions suggested at each moment.

As a result of some adaptation decisions the ACM module sends several messages to the ICM module so the UI may reflect the adaptativity either in its look and behaviour. For instance when some button has to change its behaviour a message like the following is sent to ICM

*state\_button(Nmode, ButtonId)*

When some options of a context-sensitive menu are to be disabled as a result of the evaluation of the associated operation precondition, a message like

*state\_option(NMode, MenuId)*

is issued by the module ACM. An easily extendible set of predefined events like those presented above establish the

communication for adaptation between the two modules.

## Communication Protocol

During the interactive dialogue the user may only establish communication with ICM. This module, as stated before, is responsible for interpreting user instructions and generate the corresponding events to be sent either to ACM or to the application. The application also answers to events used to invoke application commands by sending other events that correspond to the communication of results. These events will then be communicated and dealt by the module ACM.

Because GAIA is a multiparadigm environment we had to define two different protocols for module communication. The protocol between ICM and ACM, because the module ACM is based on a Prolog interpreter, was implemented as a set of Prolog predicates with the general form

*event\_input([Event, Parameters])*

The communication between ACM and ICM, because the kernel of the ICM module was written in C, is made using a low level protocol representing events as a structured list of bytes which structure is irrelevant in this context.

Finally, the communication between the application and ACM is done through ICM and is based on the "callback" mechanism of the X-Windows system.

In order to clarify the communication aspects of the GAIA system and synthesize its behaviour, we present next the list of steps needed to execute an application command and how its execution may change the UI.

1.- The user, via the command-line, inputs a command;

2.- ICM translates the input into events and sends them to ACM;

3.- ACM treats the received events, validates the syntax of the input command and, if it is correct, sends back to ICM an event requiring the corresponding application execution;

4.- The application executes the command and returns an event to ACM indicating the result of the execution;

5.- According to the result of the command execution, ACM sends to ICM an event either of error or result;

6.- If the execution succeeds, ACM may change its internal state and, if it is the case, as a result of the internal analysis for adaptation, notifies the module ICM of the possible UI changes;

Apparently, this may induce the idea that there is a great deal of overhead in the system due to such a complex and low level communication. However, the experiences so far conducted using GAIA do not confirm this initial concern.

### **GAIA as an Analytical Tool**

In addition to all the functionality and characteristics that were implemented in GAIA and have just been presented, there is another characteristic of the system that, since the beginning of its design, was considered very important and deserves to be stressed here.

It is well known by those who experienced the area and by those who read the literature on adaptive UIs (for instance [2]) and on user models (for instance [1]), that adaptive UIs are the most complex UIs to design. The reason is well known too. They are UIs that change dynamically and their changes are dependent on the inferred knowledge they may assume about their

user/s interactive knowledge. However, while the "real" user knowledge is continuously changing, that is, is a continuous variable, the representation of this user knowledge by the adaptive systems is discrete, that is, evolves by steps, layers or pre-defined levels.

Therefore, these representations only have information to produce a discrete simulation of the reality. So, in order to design a fine-grained discrete simulation of the knowledge of the user, by some means we need to know very well the user and to be able to reflect this kind of knowledge in the adaptation process. In technical terms this means that the final user model to embed in the adaptive system has to be iteratively refined towards its final definition. To be able to do so, not only we need a clear definition of the adaptation level of abstraction (the adaptation dimensions) but also a clear understanding of the adaptation process.

In GAIA, since the beginning of its design, a requirement for system adaptation understanding and iterative design of each user model was emphasized, and, subsequently, really implemented.

As a result, GAIA is able to always start from a default UI and a default user model. However, and starting from this initial state, the system is able to produce, if working in the so-called *inspect mode*, a set of events called *demo outputs* that may help to analyse not only user/s behaviour but also the system/s decisions or adaptation. This huge set of informations, gathered user by user and session by session, is registered in the GAIA system with the main aim of allowing for the analysis of the adaptation criteria and model of the user by experts in the human factors area. In this way GAIA may be seen as an analytical tool that starts to help designing adaptive user interfaces but may end up helping to design adaptable UIs from adaptive ones.

During the execution of this project we were not yet able to get the supposedly and justifiable needed, contribution from the human factors engineers and psychologists. I hope that the capabilities of the software architecture presented may trigger some interest from them, towards a very much needed, possible and indispensable symbiosis.

## CONCLUSIONS

The GAIA system presented in this paper is technically a very complex system covering different areas. Because of its immaturity it was not yet possible to submit it to a real testing, which could allow for a measurement of its degree of success and effectiveness. However, given its above shown characteristics it is, undoubtedly, a good iterative tool for the analysis and the design of adaptive UIs.

Prototyping and experimentation are also characteristics that are supported by GAIA. Using these characteristics of the system, and the expertise of the human factors engineers and the knowledge of the psychologists, we believe that much better user models may be achieved, and better adaptive UIs may be built using a multidisciplinary and step-by-step refinement process.

The GAIA/s software architecture is open and modular which facilitates the introduction of changes in some of the functionalities of its components. The system is running and a user/s manual is available [9].

## REFERENCES

1. Barnard, P and Harrison, M. Towards a framework for modelling human-computer interaction, Working Paper

RP3/WP5, ESPRIT BRA Project 3066-Amodeus, May, 1991.

2. Browne, D., Totterdell, P. and Norman, M. (Eds.), *Adaptive User Interfaces*, Academic Press, London, 1990.
3. Carlsson M. et al. *SICStus Prolog User/s Manual*, Swe-dish Institute of Computer Science, SICS Tech. Rep. T91:11B, October, 1991.
4. Croft, W.B. The role of context and adaptation in user interfaces, *Int. J. Man-Machine Studies* 21 (1984), 283-292.
5. Edmonds, E.A. Towards a Taxonomy of user interface adaptation. In *Proceedings of the IEE Colloquium on Adaptive Man-Machine Interfaces* (London, 1986), Digest 110, 3/1-3/6.
6. Hartson, H.R. and Dix, D. Human-Computer Interface Development: Concepts and Systems for its Management, *ACM Computer Surveys*, 21, 1 (1989), 5-92.
7. Jones, O. *Introduction to the X-Windows System*, Pren-tice-Hall, Englewood Cliffs, N.J., 1989.
8. Kuhme, T., Dieterich, H., Malinowski, U. and Schneider-Hufschmidt, M. Approaches to Adaptivity in User Interface Technology: Survey and Taxonomy. In *Proceedings of the IFIP TC2/WG2.7 Working Conference for Human-Computer Interaction* (Ellivuori, Finland, Aug. 10-14), Elsevier Science Publishers, North-Holland, 1992, pp. 225-250.
9. Martins, F.M. and Gouveia, P. *GAIA: Technical, Reference and User/s Manual*, Tech. Report DI/INESC, August, 1995.
10. Open Software Foundation, *OSF/Motif Programmer/s Guide Toolkit*, January, 1989.

11. Pereira, F. and Warren, D. Definite Clause Grammars for Language Analysis - A Survey of Formalisms and a Comparison with Augmented Transition Networks, *Artificial Intelligence 13*, (1980), 231-278.
12. Sherman, E.H. A User-Adaptable Interfaces to Predict User/s Needs, Report KSL-90-56, Knowledge Systems Laboratory, Stanford University, August, 1990.
13. Sterling, L. and Shapiro, E. *The Art of Prolog*, MIT Press, Cambridge, UK, 1986.