Towards a Very Large Model-Based Approach for User Interface Development

Jean Vanderdonckt, Pierre Berquin
Université catholique de Louvain
Institut d'Administration et de Gestion
Place des Doyens, 1, B-1348 Louvain-la-Neuve (Belgium)
Phone: +32-10-47.85.25 - Fax: +32-10-47.83.24
E-mail: {vanderdonckt, berquin}@qant.ucl.ac.be

Abstract

This paper introduces the concept of metaphorical structurer as a way of conciliating user interface development by demonstration and by generation so that advantages of both approaches are exploited together. It basically consists of substituting part or whole of the contents of a preliminary presentation model by a set of pre-defined specifications attached to a presentation and behavior. This substitution is aimed at forming a new example from results of a previously generated user interface. This example can be reused in a programming by demonstration approach to exploit past experience. This process seems to be particularly helpful and efficient when the models used for specifying a new application become very large and are largely based on previously made designs. Advantages and shortcomings of this technique are analyzed and exemplified within an existing model-based user interface development environment called SEGUIA.

1. Introduction

Among other, four major approaches in User Interface (UI) development can be observed:

1. *Traditional approach*: a developer selects Interaction Objects (IO) from a palette in a graphical UI editor, drags them and drops them on a working surface to progressively build the presentation of a final UI. This presentation is saved in a dedicated file to be further expanded with the behavior and the call of semantic functions of the interactive application;
2. *Programming by demonstration*: a developer builds a UI by demonstrating how it will look and behave in the real world from an example [18]. The development process is similar to the traditional approach, except that the example is added with a sequence of actions to be reproduced in the future UI. This development is intrinsically reusable since a same example can serve for many different UIs. However, the developer is entirely responsible for the usability of any example, unless a critique-based tool can provide some design assistance. Most such software (e.g., KRI [15], Hydra [6]) delivers their critiques only after completing a UI rather than during the progressive UI development. This late intervention limits the potential benefits of critiques. The developer keeps the same responsibility.
3. *Model-based approach*: a developer specifies one or several models in a model editor (e.g., an application model, a data model, a domain model, a data flow model as in fig. 1) with the help of a declarative formal language from which a UI is generated [5,10,12,19]. This process is qualified as an *automated generation* of a UI if the developer does not need to cooperate during the whole production process. Once all required models have been specified, the production process is executed without any human interruption. This process is qualified as a *computer-aided design* of UI if the developer is able to explicitly control the production process by controlling parameters, by deciding design options on the fly, by accepting or rejecting options proposed by the system… Environments created to support this approach are called Model-Based Interface Development Environments (MB-IDEs).

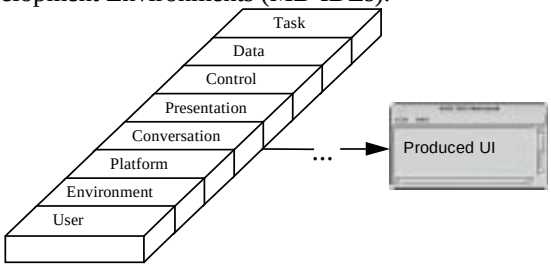

Figure 1. Models used in a model-based approach.

4. *Task-based approach*: this is approach is similar to the former, except that a task model is firstly specified. Other models can then be specified, derived, or refined depending on the task model [2,9,11,13,14]. A final UI is produced similarly to the model-based approach. The difference between UI automated generation and computer-aided design also exists here.

In the second and third categories, software can support the developer at a higher level than the one concerned by the models used in the approach. In particular, some support for addressing UI design guidelines, for deciding a correct alternative for each design option can be provided. Despite the quality of this support, software for model-based and task-based approaches are criticized on the basis of the resources (e.g., time, information, and complexity) required to specify the models. Indeed, in current versions of model-based and task-based approaches, the quality of the resulting UI heavily depends on both the quality and quantity of information captured in the models and on:

- The problem semantics expressed in the system, however complete and consistent these specifications could be.
- The expressive power of underlying models.
- The abstraction power resulting from the specifications captured in the models.
- The knowledge recorded in data base for UI building: this knowledge can take the form of design heuristics, design guidelines, production rules, first-order predicate logical formula, or semantic nets.

If these resources needed by model-based or task-based approaches exceed resources needed by a traditional approach, developers and designers are likely to reject them because of lack of these resources (mostly, the lack of time). When the problem domain for which a UI is to be built becomes **very large**, the underlying models become too large and unsustainable for developers and designers. For instance, an entity-relationship model of a data intensive system may hold up to a hundred entities and relationships, an object-oriented domain model can hold up to ten different types of relationships: these models are very-resource consuming to be acquired. This argument mainly rests on an observation stating that model-based and task-based approaches are mainly used at design time for one-shot UI building: one UI is built at a time from one or many models and, when another UI is to be designed, other models are specified. In this case, the modeling effort remains roughly the same for each new UI. We believe that these approaches are not frequently used for re-designing or maintaining an existing UI or for very large models for this reason. If we still want to use such approaches in these cases, we believe that model-based and task-based approaches as we know them today may be expanded to address questions raised by very large problem domains. Our aim is therefore to set up some first milestones towards a very-large model-based approach for UI development.

Moreover, a UI produced by a model-based approach is entirely based on the predefined set of IOs supported by the MB-IDE. If a new IO is to be added, this addition cannot be produced by the MB-IDE, but should be hand-coded. Consequently, any UI resulting from a model-based or task-based approach cannot be superior to what we can already produce with existing material. This dependence induces some lack of flexibility and expandability of the produced UI with respect to the ability of the MB-IDE to consider new aspects (such as new IOs, new icons, new interaction techniques). This conclusion is not compatible with the emergence of new IO libraries (e.g., the Borland Visual Solutions Pack, the Tea Widget Set for Java), new metaphors, new multimedia and multimodal input/output techniques, etc.

The inclusion of these novelties and the iterative UI building cycle requires that a UI produced by such approaches needs to be imported into a direct manipulation graphical editor to be adapted to these novelties and to all other techniques that are not supported by the generation. For instance, a UI generated in TRIDENT [2] needs to be imported into a graphical editor at different steps of the development process:

- at the step of IO selection [3] (e.g., an IO selected by the system can be replaced by another one chosen by the developer);
- at the step of IO positioning [1] (e.g., a developer can refine the arrangement of IOs produced by the system);
- at the step of IO attribute setting (e.g., all IOs attributes such as color, texture, font can be modified when needed);
- at the step of distributing these IOs into higher level IO (e.g., moving a group box of edit fields from one window to another dialog box).

Recording these manual modifications seems important not only at the time of the modifications themselves, but also in the future in order to reapply them on each UI redesign or regeneration, as in MECANO [19]. If these manual modifications are not saved when they are performed, the designer or the developer will be forced to reproduce this effort each time a new version of the UI is produced, for re-design or for re-generation. A re-generation may be required when one of the underlying models has been modified.

Whichever one of the four major approaches in UI building followed, a developer will necessarily go through two development steps which are not necessarily distinct: the selection of IOs and their placement into containers. In the rest of this text, we will show how programming by demonstration and model-based or task-based approaches can complement each other to address the above issues, at least at the level of these two development steps. One is considered as soon as the other one fails. For this purpose, some limits of model-based and task-based approaches are first highlighted at these two steps.

2. Some limits of model-based approaches

2.1. Some limits of the IO selection

Applying several techniques in systems has already proved the feasibility of IO selection. Among them are

1. Production rules expressed as IF… THEN… ELSE… statements, as in UIDE [5].
2. Selection trees, as in TRIDENT [21], HUMANOÏD [16] and PC-Data [20].
3. Knowledge base as in DON [5].

Both methods share an operational set of predefined IOs (e.g., an edit field, a push button, a combination box) and a rules base leading to one or several IO selected in this predefined set. These rules are fired according to the current value of various parameters such as data type, length, format, value domain, usage frequency for each data item manipulated in a UI. The value of these parameters is generally recorded in different models, mainly the data model or the domain model. The more usable a UI should be produced, the more extended and refined the predefined set of IOs should be. These extensions and refinement are performed in TRIDENT by incorporating new IOs on branches or leaf nodes of appropriate selection trees. Obviously, such an extension makes selection trees fast growing in their total amount of branches and nodes, thus increasing the depth tree and the quantity of parameters to be considered together. Some experience reveals that

1. As a selection tree is extended, specifying the related model becomes a longer task (the quantity of parameters is increasing) and a more abstract task (the representativeness of most recently added parameters is decreasing) (fig. 2). A developer may therefore loose the patience to specify all model parameters required to obtain a nearly optimal UI. This patience de-crease becomes even more obvious when identical or similar parameters are reproduced several times.

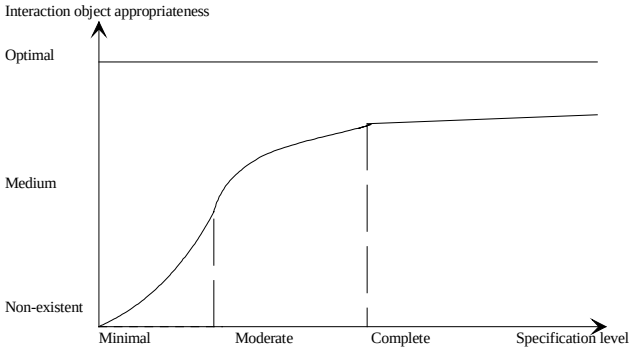

Figure 2. IO appropriateness in function of specification level.

2. The appropriateness of a particular IO in function of the specification level provided in the models follows an asymptotic curve: in the third part of the curve in fig. 2, many new specification elements are added with a only limited impact on the IO appropriateness for the considered UI.
3. The expressiveness of new parameters required to improve the appropriateness of a particular IO, and thus the UI usability, also asymptotically decreases after a given threshold (fig. 3): finally, last parameters no longer represent significant changes. Although figures 2 and 3 are mostly impressionistic, they are based on the experience we gained seeing some designers working with such a system.

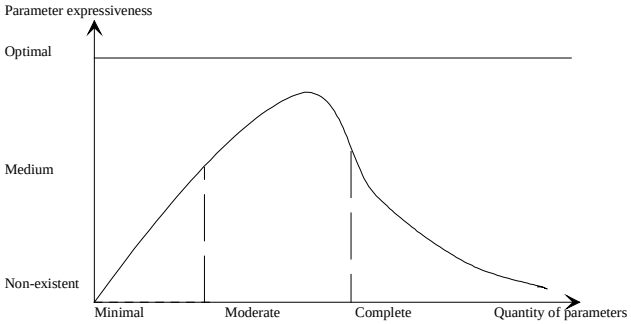

Figure 3. Parameter expressiveness in function of the quantity of parameters to be expressed.

4. The reasoning visualization, required by any artificial intelligence approach, requires more and more manipulations of the selection tree (e.g., zoom in and out operations).

We are therefore confronted with a dilemma:

1. Either we preserve a simple, yet limited, selection tree: it can therefore be easily visualized and browsed, but the produced UI will only display some limited usability for this aspect (HUMANOÏD [16], GENIUS [12] and PC-DATA [20] choose this alternative).
2. Or we prefer a complex, yet extended, selection tree: it can produce a UI where IOs are more appropriate, then more us-able, but becomes very hard to manage, to maintain and to produce a mental representation that can be transferred easily; in addition, it increases the abstraction to a level which may no longer be interpreted as significant by developers and designers (TRIDENT [21] choose this alternative).

2.2. Some limits of the IO positioning

Many strategies already exist to provide some assistance to developers and designers to position IOs into containers. Such strategies include simple column strategy [5], double column strategy [12], balanced double column strategy [1,2], and right/bottom strategy [1]. None of them

can produce a reasonably usable layout in most case studies. Moreover, adaptations are often needed to exploit their full capacity of exploring alternative and usable layouts. And finally, some new strategies still need to be discovered where all existing strategies are inefficient or fail.

A possible remedy may lead developers and designers to benefit from an advice-giving system that proposes panoply of alter-native strategies with respect to the semantics or the configuration of the current problem. We do not see today sufficient knowledge to incorporate it into such a system. Therefore another approach might be needed.

3. User interface building with a metaphorical structurer

The above limits lead us to conclude that extending a MB-IDE supporting the two development steps of IO selection and positioning will be more expensive than the potential benefit generated by a more usable UI. To overcome this conclusion, we replace a subset of model significant specifications by a reference to a structurer.

3.1. Definition of a metaphorical structurer

A metaphorical structurer, or structurer for short, consists in a non-empty predefined set of parameters which have already been fully specified and attached to one or many IOs considered as usable in the context of interactive tasks to be carried out by the users' population. This structurer is qualified as "metaphorical" in the sense that it can integrate a visual metaphor attached to the IOs related in the structurer. This definition is of course wider than the one that is commonly admitted for traditional metaphors [17]: a metaphor covers more than its name if it can relate both presentation and dialogue aspects together in one format. By metonymy, the term "metaphor" will be used in name and place of "metaphorical structurer" in the rest of this text. Here are some examples of metaphors for information having different data types.

3.1.1. Chronometer. Its aim is to display spending time from a starting moment. Possible model specifications could be: Data type = hour; Data length = 6, Interaction = display, Domain = known; Expandable Domain = no, Continuous Domain = yes, Orientation = horizontal. Other specifications are not required since they can be deduced from the data type. The behavior is limited to the display of spent time in response to an event (e.g., a mouse click on the IO). The time can be stopped and relaunched in the same way. A possible presentation could be

3.1.2. Clock. Its aim is to display real time. Let us note that hour input with a clock could be more complex or time consuming than with a common edit field (in particular, precision is more relative). But the clock is more user friendly than the edit box. Possible model specifications could be: Data Type = hour; Data Length = 6; Interaction = input/display; Domain = known; NumberChoice Values = 1; Number Possible Values = 46656; Number-Secondary Values = 0; Expandable Domain = no; Continuous-Domain = yes; Precision = low; Orientation = circular. A possible presentation could be

3.1.3. Two states switch. Its aim is to input/display a switch between two antagonist Boolean values, i.e. a data whose negation is not necessarily the opposite of the first (for instance, "yes" and "no" are antagonist, but "alternative current" and "continuous current" are not). Possible model specifications are: Data type = Boolean; Domain = known; Number Possible Values = 2; NumberChoiceValues = 1; Antagonist Values = yes; Possible Values = {Alternate, Direct}; Orientation = horizontal. The two states of a possible presentation could be

3.1.4. Gauge. Its aim is to produce a visual and/or auditory feedback when the warning limit of a value oscillating between a low limit and a high limit is reached. Possible model specifications could be: Data type = integer; Data length = 3; Interaction = display; Domain = known; Number Possible Values = 100; NumberChoiceValues = 1; Expandable Domain = no; Continuous Domain = yes; Low Limit = 1; HighLimit = 100; Warning Limit = 80. The behavior is the native behavior induced by the IO itself, that is by a progress indicator. A possible presentation could be

3.1.5. Traffic light. Its aim is to denote an acceptation status either as input or as output. Possible model specifications could be: Data type = Alphanumeric; Data Length = 6; Inter-action = input/display; Domain = known; NumberPossibleValues = 3; Number Choice Values = 1; Possible Values = {Green, Orange, Red}. Its behavior is regulated either by a user clicking on the colors denoting the different states or by the application. The three states

of a possible presentation could be 
The model specification does not capture the constraints that a traffic light must obey (e.g., the red and amber lights can be on together, but the red and green cannot). We assume that such constraints are represented in the behavior part which can be harder to abstract. In this case, the mutually exclusive character between the values is preferred.

3.2. Building process

Traditionally in programming by demonstration, a developer builds by hand a first example, adapts it and saves it with a prototype name to reuse it whenever needed. A prototype can therefore be considered as a useful design structure for creating a UI: all IOs types and instances to be created are specified by referring to a prototypical instance that enables developers to create new IOs by copying this prototype, by instantiating it and by refining it to the specific context [18].

Traditionally in model-based or task-based design, a developer introduces the complete specifications of the underlying models (here a function chaining graph), produces a first UI to be refined to the specific context in an appropriate graphical editor. This process was reiterated each time a new UI was produced or each time a model specification has changed. The big win in this case is that, when this change has been recorded, all the rest changes automatically. We introduce here a UI building process that combines both approaches in the following process:

1. Underlying models are specified as far as they can be (sometimes, not all specifications can be known at the very first stage) not only to save these UI useful specifications, but also to maintain a logical link between these specifications and a finally produced UI. This link does not seem to exist in programming by demonstration;
2. A first UI is generated according to a model-based or task-based approach and supported by a MB-IDE;
3. The generation results are tailored to the specific needs, including all capabilities and features that are not supported by the generation process, the MB-IDE (e.g., the usage of new metaphors, fine-grained actions, alternate interaction techniques);
4. This final UI is saved with a metaphor name;
5. As soon as specifications identical or similar to a previously saved metaphor are manipulated, only minimal specifications are input: name, data type, length, format, values domain;
6. Then, a previously saved metaphor is attached to these new minimal specifications.

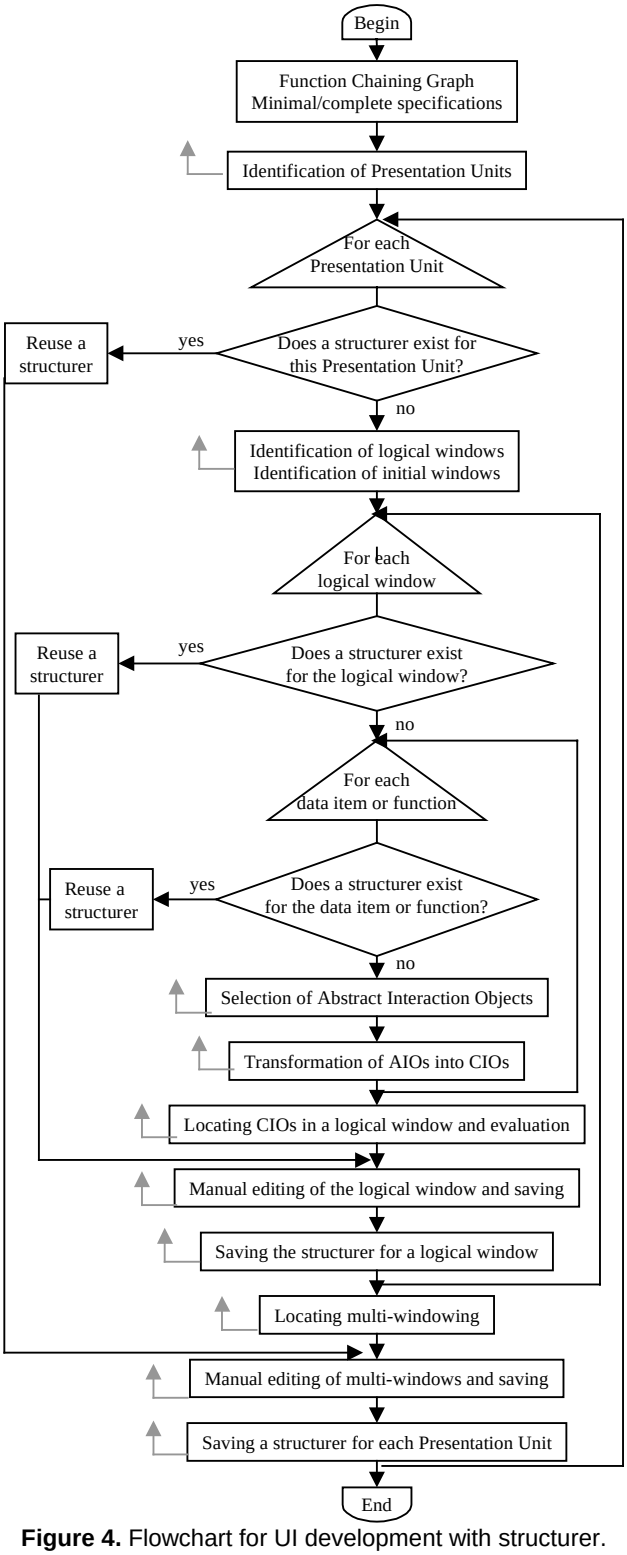

Figure 4. Flowchart for UI development with structurer.

Fig. 4 reproduces the flowchart followed to identify an existing structurer at different levels of presentation design and to reuse it. If no structurer can match the specifications, a new presentation is generated, edited and saved for future usage.

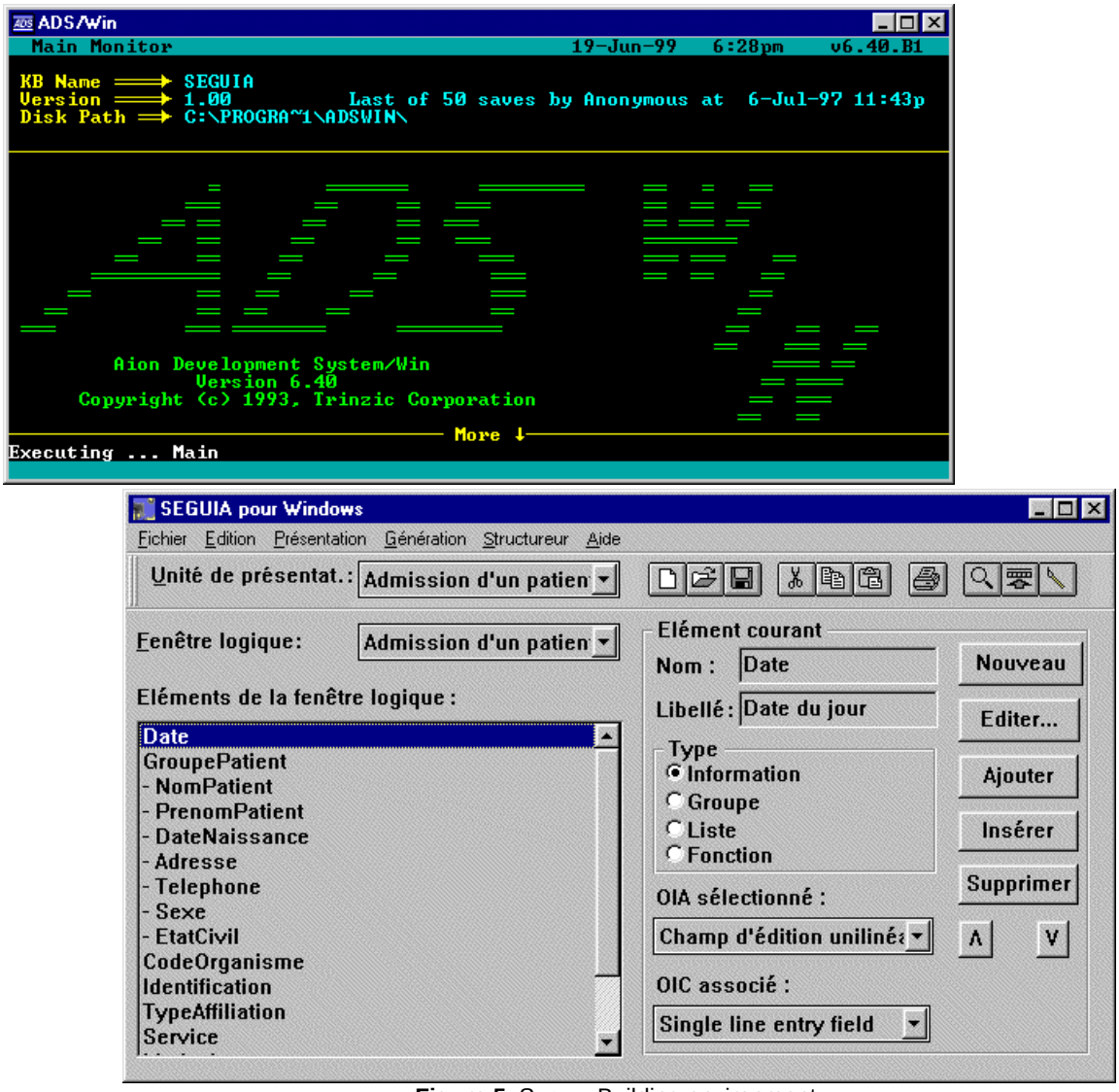

Figure 5. SEGUIA Building environment.

The attachment of a structurer not only completes missing specifications by filling them in, but also directly links these specifications to one or many known IOs. These IOs have been previously saved, thus it is assumed that they successfully went through the selection and positioning process (step 3). This process will generate an example extracted from a metaphor base. For instance, if a developer wants to specify parameters related to a data item "Indicator of free memory", rather than introducing again the complete specifications, the developer can resume this task to a reference to an existing metaphor by telling the software Metaphor = gauge. This imports all saved specifications along with presentation and behavior. The developer is then free to tailor these specifications, this presentation and this behavior to specific needs and to save these results as a new metaphor called "Memory indicator". This metaphor will be a sub-type of "gauge".

4. Concept implementation

4.1. An environment for UI building by metaphor

A prototype environment supporting the process described in section 3.2, SEGUIA, has been implemented as an UI generator on top of an expert system shell AION/DS marketed by Trinzic Software under Microsoft Windows (fig. 5). In the list box labeled "Hierarchy", a hierarchical structure of information and actions to be manipulated in a task unit is decomposed. Each item is minimally specified in the right group bow labeled "Hierarchy Current Item". Pressing the "Edit…" push button gives access to complete specifications depending on the data type and other parameters.

Let us exemplify how SEGUIA supports the metaphor-based UI process for the "Sex" information. Rather than

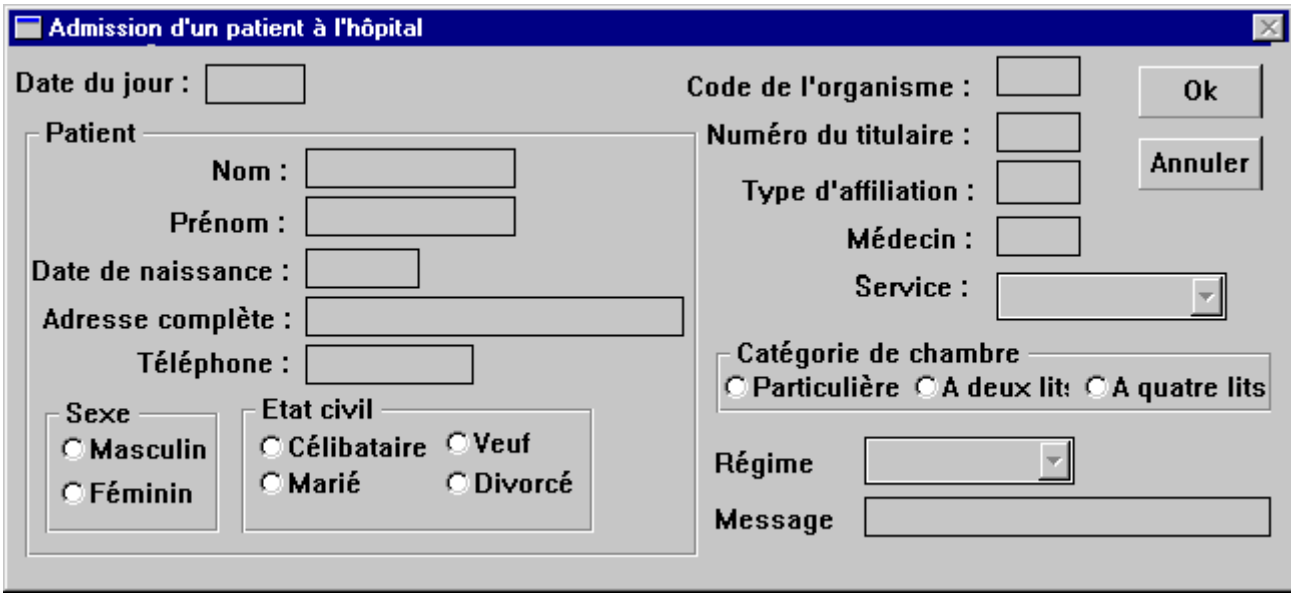

Figure 8. Window for patient admission to a hospital.

entirely specifying all parameters in the data model for this item, the developer introduces the minimal specifications: here, a data type (alphanumeric) and a multiplicity (simple) to state a simple choice mechanism. Then, the "Metaphor" menu (fig. 6) enables the developer to look for an appropriate metaphor to complete these minimal specifications through six menu items. If the developer has experience enough to know what type of metaphor could be here applied, he or she can open an existing metaphor, apply it to the current item, refine it if necessary and save it for future purposes in a straightforward manner. Contrarily, if the developer does not remember a type of metaphor that could be here applied, he or she can search the metaphor base for candidate metaphors. Two mechanisms have been implemented so far to support this search:

1. If the minimal specifications are identical or similar to the ones saved in an existing metaphor, this metaphor is selected and can be viewed separately. This mechanism basically consists in comparing specifications items and reporting the results of comparison to the developer;
2. If the minimal specifications are only very partially identical or similar to the ones saved in an existing metaphor, this metaphor is selected and can be viewed separately. This mechanism essentially consists in applying pattern-matching algorithms in a case-base library [8].

Once an existing metaphor has been applied, missing specifications are either filled or proposed to confirmation to the user on demand, the attached presentation and behaviour are transferred and adapted to local peculiarities.

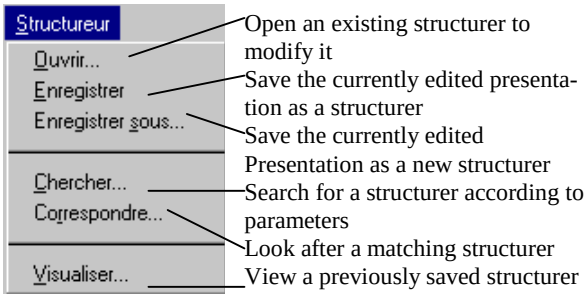

Figure 6. Menu items allowing actions on metaphors.

Fig. 7a shows the presentation of a textual metaphor for the "Sex" information consisting of a group box surrounding two radio buttons, whereas fig. 7b shows an alternative graphical metaphor which replaces the two radio buttons by a radio icon depicting the international pictogram.

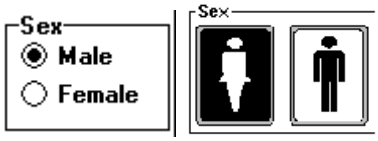

Figure 7. Textual metaphor for "Sex" information (a); Graphical metaphor for "Sex" information (b).

In some cases, the second metaphor might be considered as more appropriate than the first one. Fig. 8 reproduces the final dialog box that has been produced for the example given in fig. 5, second part. In this dialog box, the textual metaphor was kept, but nothing hinders the developer from replacing it by the graphical metaphor and to regenerate the layout according to this change. The final layout could also be saved as a global metaphor.

4.2. Advantages

The limited experience we gained with this prototype revealed the following potential benefits:

1. The mental cognitive load is reduced as a developer can think in high level terms of metaphors rather than in lower level terms of complete specifications, which can be resource consuming. Only fundamental attributes such as name, data type, format, values domain, were here specified. This seems more reasonable.
2. The operation work load for selecting appropriate IOs also decreased: it is no longer required to entirely traverse the complete selection cycle: browsing a complete selection tree becomes superfluous, selecting an IO, generating it, adapting it are not useful as soon as an existing metaphor can be reused.
3. The operation workload for positioning IOs is also shortened. Since all manual operations have been saved with the final results, they are no longer need to be reapplied for a new case; moreover, no generation is needed since the final results are directly retrieved for the definition of the metaphor for the current information. For each metaphor, the developer is not limited to one single, unique presentation. It is indeed allowed to define a preference list of alternate metaphors depending on circumstances. These circumstances are dictated by the current values of parameters and parameters coming for the problem domain or the software and hardware constraints (e.g., the availability of a particular interaction technique, the screen density). In this preference list, metaphors can be sorted by decreasing order of usability. The first level proposes an IO considered as optimally usable. Each subsequent level represents an inferior level of usability with respect to the optimal solution. For each level, several different presentations can feed in the current selection. For example, a scale can be preferably displayed vertically, then horizontally if nothing comes to defeat this strategy. By definition, a metaphor will consist

   - At the conceptual level: in complete specifications where each required parameter has been appropriately specified.
   - At the logical level: in a preference list of Abstract Interaction Objects (AIOs) [9,21] which are a kind of generalization of physical IOs belonging to the various physical computing platforms.
   - At the physical level: in a preference list of Concrete Interaction Objects (CIOs) with, for each of them, a precise value for each attribute, property (e.g., a specific colour scheme, a given position).

3. Building a UI is no longer performed by abstraction of a metaphor, but also by concretization: such a metaphor could be more expressive to the eyes of a developer rather than a long series of poorly significant specifications.
4. The manipulation of a metaphor opens a door to the embodying of nearly all kind of new IOs to represent a metaphor. For instance, a traffic light may represent the acceptation status of an administrative folder, a pictogram can depict the sex of a person in a file, a label can represent the address of people in a multicast task.
5. Promulgating metaphors may improve the usability of a new UI by referring to past experience in order not to replicate what has been done before and in order to reuse this work to ensure consistency. This property can be intrinsically verified if the same metaphors are used whenever they are compatible with the same context.
6. Building a UI by metaphor can cover both selection and positioning: where selection rules and guidelines were separated from positioning rules and guidelines, this distinction is no longer a concern. Selection and positioning can work hand in hand and independently. A selected IO can always be replaced by another and the positioning of IOs with respect to their container can always be revisited in a logical manner.
7. The metaphor is not limited to a unique information item. Of course, it could cover a simple data item (as the “Sex” of a person in a file), but also on a list of different data items (as the set of lines of a customer’s order), a group of data items (as the full address of a customer), a set of data items which can be as extended as needed (as the complete form of a customer), an action (as the recording of an insurance), a status message (as the printing status of a job). Consequently, a same structurer can be reused in input or output within a same UI (thus ensuring intra-application consistency) or across a wide family of different UIs in the same problem domain (thus ensuring inter-application consistency).
8. A structurer is still modifiable as needed. If a new metaphor should be introduced, it is possible to initiate a first design with an existing metaphor and to add required specifications ; if such a metaphor does not exist, part of existing metaphors can be imported (thanks to the pattern-matching mechanism); if none of the above work, the traditional model-based or task-based approach can be adopted instead.
9. The structurer base can be viewed as a case repository, which is perpetually maintained and enriched as the organization's experience grows. In this way, the structurer base may initiate a design memory for a particular problem domain, for a specific company, for a specific type of interactive task, or any combination of these parameters. It is expected that the presentation knowledge developed during continuous development can be better captured and reused throughout the use of structurers.

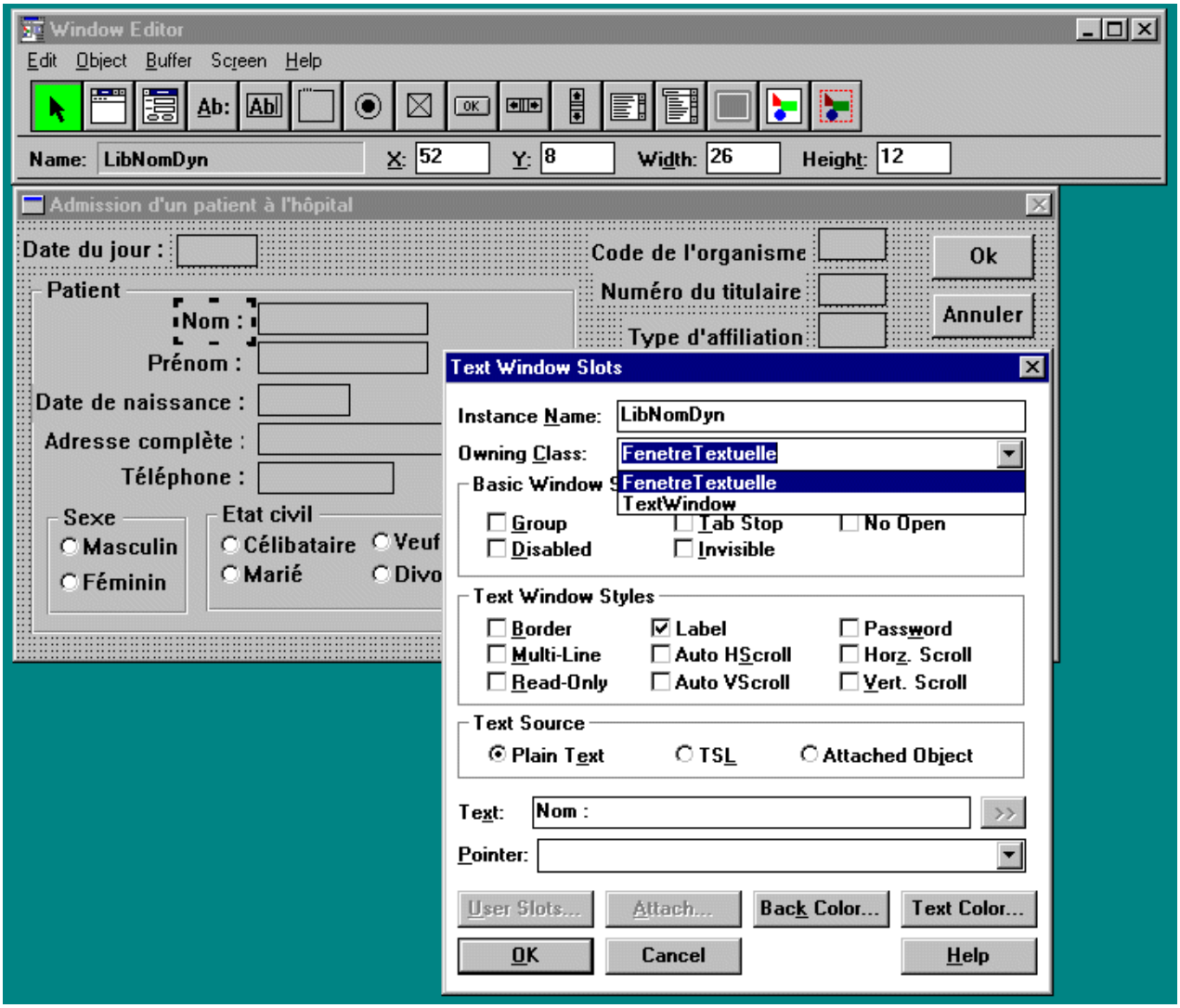

Figure 8. Manual editing of object properties in AION/DS

5. Conclusion

As we just saw, the structurer is intended to significantly reduce the specification workload; in this sense, it could be considered as a good mean to foster a lightweight model. Gray et al. have for instance developed this idea for presentation models [9]. This approach has several potential advantages that should be moderated with respect to equally potential shortcomings.

First, it is not always easy to talk in terms of metaphors in all case studies because imagining an appropriate metaphor is a complex task when there is no concrete, physical counterpart in the real world. For instance, finding a metaphor for a set of edit fields, identification labels and prompts is not obvious.

Second, UI building by metaphor is compatible with model-based and task-based approaches, but does not release designers from the need to be assisted by knowledge provided by MB-IDEs, especially for novice.

Third, the developer may be invited to define too narrow metaphors, which only vary with some very limited parameters. For instance, the temperature, the temperature expressed in Celsius degrees, the temperature of a human body expressed in Celsius degrees, the temperature expressed in Fahrenheit degrees, the temperature of a melting metal in Fahrenheit degrees may be interpreted as too close. Strategies for semantically indexing structurers to assist the designer in the choice of metaphor for the current design are needed. Further research is required for this purpose.

Fourth, the developer can perform any manual editing he or she wants in the AION/DS graphical editor. For instance, fig. 8 reproduces a situation where any property of any CIO such as an edit field can be manually modified. These manual operations are only reflected on the current example if they are not recorded in a structurer.

Finally, reusing the presentation of an existing metaphor is straightforward, but reusing the behaviour of this metaphor may require more than a copy of the programming code realising this behaviour and an adaptation to new parameters. This can seriously reduce the easiness of reusing a metaphor, especially when the underlying models are very large.

The metaphorical structurer has been introduced to mainly address questions raised by very large model-based or task-based approaches, but is only a first step towards such a goal. It is greatly expected that the complexity, the size of very large models specification can be reduced as parts of the models have been already specified elsewhere.

6. References

[1] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, and J. Vanderdonckt, « Towards a Dynamic Strategy for Computer-Aided Visual Placement », *Proc. of 2nd Workshop on Advanced Visual Interfaces AVI'94* (Bari, 1-4 June 1994), T. Catarci, M.F. Costabile, S. Levialdi, and G. Santucci, G. (eds.), ACM Press, New York, 1994, pp. 78–87.

[2] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, J. Vanderdonckt, and G. Zucchinetti, « Key Activities for a Development Methodology of Interactive Applications », Chapter 7 in *Critical Issues in User Interface System Engineering*, D. Benyon. & Ph. Palanque (eds.), Springer-Verlag, Berlin, pp. 109–134.

[3] F. Bodart and J. Vanderdonckt, « On the Problem of Selecting Interaction Objects », *Proc. of Conf. on Human-Computer Interaction HCI'94* (August 23-26, 1994, Glasgow), Cambridge Univ. Press, Cambridge, pp. 163–178.

[4] Borland International, Inc., *Borland Visual Solutions Pack*, Scotts Valley, California, 1994

[5] D.J. de Baar, J.D. Foley, and K.E. Mullet, « Coupling Application Design and User Interface Design », *Proc. of CHI'92* (Monterey, 3-7 May 1992), ACM Press, New York, pp. 259–266.

[6] G. Fischer, G. Nakakoji, J. Ostwald, G. Stahl, and T. Sumner, « Embedding Computer-Based Critics in the Context of Design », *Proc. of InterCHI'93* (Amsterdam, 24-29 April 1993), ACM Press, New York, pp. 157–164.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, « Design Patterns », Addison-Wesley, Reading, 1992.

[8] A.K. Goel, J.L. Kolodner, M. Pearce, R. Billington, « Towards a Case-Based Tool for Aiding Conceptual Design Problem Solving », in *Proc. of Case-Based Reasoning Workshop* (Washington, 8-10 May 1991), R. Rareiss (ed.), Morgan Kaufmann, Los Altos, pp. 109–120.

[9] P. Gray, R. Cooper, J. Kennedy, J. McKirdy, P. Barclay, and T. Griffiths, « A Lightweight Presentation Model for Database User Interfaces », Proc. of 4th ERCIM Int. Workshop on User Interfaces for All (Stockholm, 19-21 October 1998), C. Stephanidis and A. Waern (eds.), ERCIM. Accessible at http://www.ics.forth.gr/proj/at-hci/UI4ALL/UI4ALL-98/gray_ps.zip

[10] T. Griffiths, J. McKirdy, G. Forrester, N. Paton, J. Kennedy, P. Barclay, R. Cooper, C. Goble, and P. Gray, « Exploiting Model-Based Techniques for User Interfaces to Databases », *Proc. of VDB'4* (May 1998), Y. Ioannidis and W. Klas (eds.), Chapman & Hall, London, pp. 21–46.

[11] T. Griffiths, J. McKirdy, N. Paton, J. Kennedy, R. Cooper, P. Barclay, C. Goble, P. Gray, M. Smyth, and A. Dinn, « An Open Model-Based Interface Development System: The Teallach Approach », *Comp. Proc. of DSV-IS'98* (Abingdon, 3-5 June), P. Markopoulos and P. Johnson (eds.), ERCIM Press, Aire-la-Ville, 1998.

[12] C. Janssen, A. Weisbecker, and J. Ziegler, « Generating User Interfaces from Data Models and Dialogue Net Specifications », *Proc. of InterCHI'93* (Amsterdam, 24-29 April 1993), ACM Press, New York, pp. 418–423.

[13] P. Johnson, S. Wilson, P. Markopoulos, J. Pycock, « ADEPT-Advanced Design Environment for Prototyping with Task Models », *Proc. of InterCHI'93* (Amsterdam, 24-29 April 1993), ACM Press, New York, p. 56.

[14] J. Löwgren and T. Nordquist, « Knowledge-Based Evaluation as Design Support for Graphical User Interfaces », *Proc. of CHI'92* (Monterey, 3-7 May 1992), ACM Press, New York, pp. 181–188.

[15] P. Luo, P. Szekely, and R. Neches, « Management of Interface Design in HUMANOÏD », *Proc. of InterCHI'93* (Amsterdam, 24-29 April 1993), ACM Press, New York, pp. 107–114.

[16] A. Marcus, « Human Communications Issues in Advanced UIs », *Communications of the ACM*, 36, 4, April 1993, pp. 100–109.

[17] B.A. Myers and W.A.S. Buxton, « Creating Highly Interactive and Graphical User Interfaces by Demonstration », *Proc. of SIGGRAPH'86* (Dallas, 18-22 August 1986), Computer Graphics, 20, 4, August 1986, pp. 249–258.

[18] A.R. Puerta, E. Cheng, T. Ou, and J. Min, « MOBILE: User-Centered Interface Building », *Proc. of CHI'99* (Pittsburgh, April 1999), ACM Press, New York, 1999.

[19] B. Suck, « Presentation Dictionary: Vom Daten zum Dialogelement bei DATEV », E&I Forum, 1983, pp. 34–39.

[20] J. Vanderdonckt and F. Bodart, « Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection », *Proc. of InterCHI'93* (Amsterdam, 24-29 April 1993), ACM Press, New York, pp. 424–429.

[21] S. Wilson and P. Johnson, « Bridging the Generation Gap: From Work Tasks to User Interface Designs », *Proc. of CADUI'96* (Namur, 5-7 June 1996), J. Vanderdonckt (ed.), Presses Universitaires de Namur, Namur, 1996, pp. 77–94.