

# Retrospective and Challenges for Model-Based Interface Development

Pedro Szekely

Information Sciences Institute, University of Southern California  
4676 Admiralty Way,  
Marina del Rey, CA 90292, USA  
Phone : +1-310-822-1511 (ext. 641) - Fax : +1-310-823-6714  
E-mail : szekely@isi.edu

**Abstract.** Research on model-based user interface development tools is about 10 years old. Many approaches and prototype systems have been investigated in universities and research laboratories around the world. This paper proposes a generic architecture for these tools, reviews the different approaches in light of this architecture, and discusses their progress towards the goals of increasing the quality and reducing the cost of developing interfaces. The paper closes with a discussion of challenges for future model-based development tools.

**Keywords.** Model-based interface development, automatic user interface generation, user interface design.

## 1 Introduction

Model-based user interface development tools trace their roots to work on user interface management systems (UIMS) done in the early 1980's [27]. UIMSs sought to provide an alternative paradigm for constructing interfaces. Rather than programming an interface using a toolkit library, developers would write a specification of the interface in a specialised, high-level specification language. This specification would be automatically translated into an executable program, or interpreted at run-time to generate the appropriate interface.

Many early UIMSs focused on dialogue specification [15]. They used state transition diagrams [18], grammars [30, 31] or event-based representations [41] to specify the interface responses to events coming from the input devices. The display aspects of the interface were typically specified outside the specification language, in call-back procedures that painted the screen as appropriate.

Some UIMSs used as their main specification the type and procedure declarations that defined the functional aspects of the application [3, 29]. Based on this information, they generated menus to invoke the procedures, and dialogue boxes to prompt users for the information needed to construct instances of the types.

Through the late 1980's and early 90's the specification languages became more sophisticated, supporting richer and more detailed representations that allowed the systems to generate more sophisticated interfaces. Today's systems use specifications of the tasks that users need to perform, data models that capture the structure and relationships of the information that applications manipulate, specifications of the presentation and dialogue, user models, etc.

The term *model-based interface development tools* refers to interface construction tools that use these rich representations to provide assistance in the interface development process. Tools range from automatic interface generation systems, generators of help systems for applications, interface evaluation tools, advisors, etc.

Even though model-based interface development tools are much more sophisticated than early UIMSs, they have not become popular in the commercial sector. Most software developers use interface builders, toolkits and a programming language to build the interfaces for interactive systems.

The main goal of this paper is to review the current progress in model-based tools, and discuss challenges for the next generation of user interface tools in general, and model-based tools in particular.

The paper is organised as follows. The next section will describe a general architecture of model-based tools that provides a way to classify model-based tools according to the components of the architecture that they emphasise. The sections after analyse the success of model-based work on automatic interface generation, high-level specification systems, help generation, and design advisors. The last part of the paper discusses new challenges for user interface software, including multi-platform support, intelligent support for the user, multi-modal interfaces and end-user tailoring. The paper closes with conclusions about the future of model-based tools.

## 2 Generic Model-Based Interface Development Architecture

Fig. 1 shows the typical components of a model-based interface development environments (MB-IDE). The rounded rectangles represent tools, the other shapes represent information produced or consumed by the various tools. The main components of the architecture are the *modelling tools*, the *model*, the *automated design tools*, and the *implementation tools*. Developers<sup>1</sup> use the modelling tools to build the model. The automated design tools are used to perform certain design activities that developers either choose or are forced to delegate to the system. The implementation tool transforms the model into an executable representation that is linked with application code, and delivered to the end-users. The following subsections discuss these components in more detail.

---

<sup>1</sup> This paper uses the term developer to refer to all the people involved in constructing an interactive application. When appropriate, the more specific terms such as task analyst, graphic designer, programmer, etc. will be used.

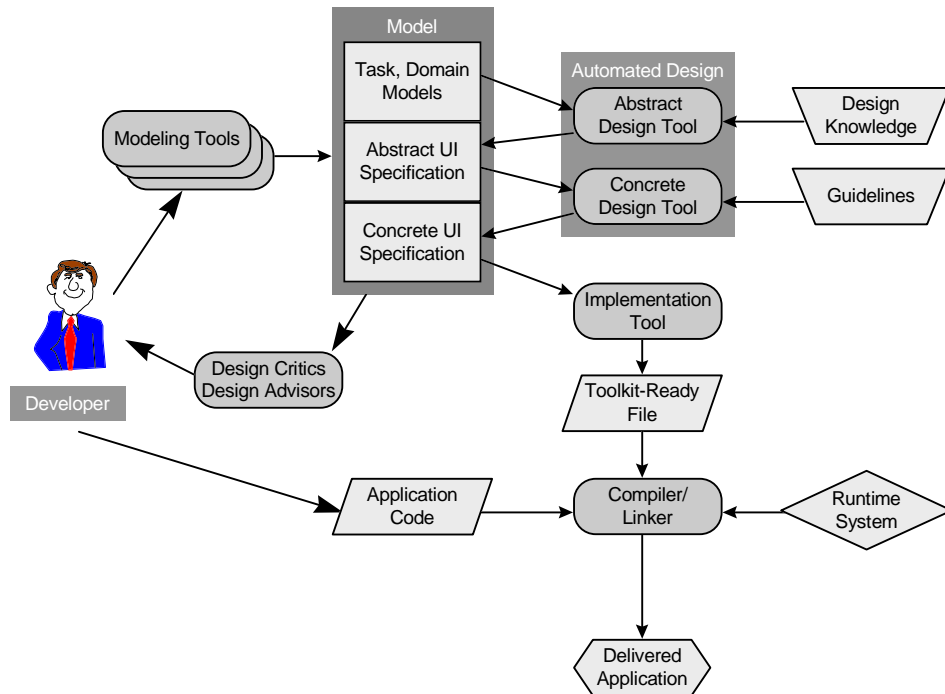


Fig. 1 Model-Based Interface Development Process

## 2.1 Model

The model is the main component of the system. The model typically organises information into three levels of abstraction. At the highest level are the task and domain model for the application. The task model represents the tasks that users need to perform with the application, and the domain model represents the data and operations that the application supports. Tasks models typically represent tasks by hierarchically decomposing each task into sub-tasks (steps), until the leaf tasks represent operations supplied by the application.

The second level of the model, called in this paper the *abstract user interface specification*, represents the structure and content of the interface in terms of two abstractions, *abstract interaction objects (AIO)*, *information elements* and *presentation units*. AIOs are low-level interface tasks such as selecting one element from a set, or showing a presentation unit. Information elements represent data to be shown, either a constant value such as a label, or a set of objects and attributes drawn from the domain model. Presentation units are an abstraction of windows. They specify a collection of AIOs and information elements that should be presented to users as a unit. In summary, the abstract user interface specification specifies in an abstract way the information that will be shown in each window, and the dialogue to interact with the information.

The third level of the model, called the *concrete user interface specification*, specifies the style for rendering the presentation units, and the AIOs and information elements they contain. The concrete specification represents the interface in terms of toolkit primitives such as windows, buttons, menus, check-boxes, radio-buttons, and graphical primitives such as lines, images, text, etc. In addition, the concrete specification specifies the layout of all the elements of a window.

The models of different MB-IDEs can differ substantially. Different MB-IDEs typically provide different modelling languages for specifying the contents of the model, and they also emphasise different levels of the model. For example, Mastermind [45] requires developers to explicitly specify all levels of the model, whereas Janus [1] only requires a data model.

## **2.2 Modelling Tools**

The modelling tools assist developers in building the models. The main goal of the modelling tools is to hide from developers the syntax of the modelling languages, and provide a convenient interface for developers to specify the often large quantities of information that are stored in the model. A wide range of modelling tools have been developed, often specialised to the different levels of the model. These tools range from text editors to build textual specifications of models (ITS [48, 49], Mastermind), forms-based tools to create and edit model elements (Mecano [37]) and specialised graphical editors (Humanoid [25, 43, 44], FUSE [23], many others).

## **2.3 Design Critics and Advisors**

Design critics are tools to evaluate designs. The model-based approach provides an excellent platform for constructing analytic design critics because models contain a rich representation of interface designs that these tools can analyse. Most design critics work with the concrete user interface specification layer of the model because in most cases they provide evaluations about detailed features of the interface (e.g., whether the interface provides a way to access all application functionality).

Design advisors are tools that suggest how to refine the abstract layers of the model into more concrete ones. Design advisors use a knowledge-base of design knowledge, typically represented as rules. The condition part of the rules identifies some aspect of a design (e.g., an AIO), and the action part of the rule specifies a way of refining/transforming the matched design element (e.g., the CIO to use for an AIO).

## **2.4 Automated Design Tools**

Many MB-IDEs allow developers to only specify certain aspects of a model. These MB-IDEs feature automated design tools that compute the missing elements of the model from the information that developers do provide. For example, Janus [1] only requires developers to supply a domain model, and it features an automated design tool that automatically constructs both the abstract and concrete specifications of the interface. In contrast ITS and Mastermind [45] require developers to explicitly specify

all levels of the model, so these systems do not offer automated design tools. What they do offer is the capability to re-use specifications. The following section discusses automated design tools in detail.

As shown in Fig. 1, automated design tools often use a repository of design knowledge or design guidelines that control the behaviour of the design tool. In most systems developers are not expected to modify the design knowledge, which is typically specified by user interface specialists and the architects of the MB-IDE<sup>1</sup>.

## 2.5 Implementation Tools

The implementation tool translates the concrete specification of the interface into a representation that can be used directly by a toolkit or interface builder. There are essentially three kinds of implementation tools. Source-code generators (e.g., Mastermind) generate source code in a programming language, typically C++. UIMS generators (e.g., FUSE) generate a file that can be read by an existing UIMS or interface builder. Interpreters (e.g., ITS and Humanoid) do not generate an “implementation file”, but rather interpret the model directly at runtime.

The last step in the interface generation process is to link the toolkit-ready-file with application specific code and a runtime library. This is typically done using the compiler and linker for the programming language used to implement the application. Interpreter-based systems such as ITS do not use the compiler and linker, but rather feature a runtime module that reads the models during runtime, and interprets the concrete specification of the interface.

Many MB-IDEs provide implementation tools that use the model to generate more than the user interface.

For example, Janus, FUSE, UIDE [11, 12] and Humanoid can generate significant parts of the help system for an application based on the information contained in the model. Janus not only generates the interface, but also generates the database schemas for an application, and much of the data management code.

Mastermind generates code for applications that allows other processes to connect to an application, and to request to be notified when certain tasks are completed, to be sent snapshots of the application state, and to remotely invoke application tasks. This facility supports the construction of agents that can assist users in various ways. This facility was used, for example, to build a history agent that keeps a history of all the tasks that the user has completed and allows users to re-invoke previously completed tasks.

As interfaces become more sophisticated, and users expect more services from their interfaces. The ability to provide such additional run-time services for free is one of the most attractive features of the model-based technology.

---

<sup>1</sup> ITS can be viewed as an automated design tool where developers have to explicitly build the design knowledge for each application or family of applications.

### 3 Retrospective

The following sections provide a retrospective of the main user interface design and construction problems that have been addressed using the model-based approach. These sections discuss the various approaches that have been used, and how well they solve the problems.

The retrospective section is organised into five main topics:

- *Automatic interface design.* This section discusses the main approaches for automating interface design and their limitations.
- *Specification-based MB-IDEs.* This section discusses MB-IDEs that do not try to automate interface design, but rather give developers convenient languages for expressing designs.
- *Help generation.* Many MB-IDEs feature components that automatically generate help. This section reviews the different approaches and comments on their success.
- *Modelling Tools.* This section discusses various approaches to modelling tools.
- *Design critics and advisors.* This section presents a categorisation of these tools and discusses their relative benefits.

*Note.* For each topic one or two tools are discussed in some detail. The chosen tools are not necessarily the best tools according to some metric, but rather illustrate a point well, and detailed papers have been published about them. The goal of this paper is to review the main approaches, not the individual tools.

#### 3.1 Retrospective – Automatic Interface Design

The primary goal of many MB-IDEs is to automate as much as possible the design and implementation of a user interface. These MB-IDEs emphasise the domain and task models, and automatically generate the abstract and concrete user interface specifications from these models. Most MB-IDE in this category are oriented towards database applications and produce interfaces that allow the end-users to browse the database, to edit the contents of objects, to define new objects, and to delete objects.

This section argues that automating interface design is intrinsically difficult, so MB-IDEs should be very selective about the portions of the design that they choose to automate.

##### 3.1.1 Structure of Automated Design Tools

**Model Contents.** MB-IDEs whose primary goal is to automatically design use mainly two kinds of models, a *domain model* that describes the structure and attributes of the information that the application provides, and a *task model* that describes the tasks that users need to perform. For example, tools like Janus, and early versions of Mecano, use only a domain model, whereas tools like Trident [46, 47], Adept [20], DON [22] and Modest [17] use primarily a task model, but also have a domain model.

The domain models of the automatic design tools are similar. They describe classes of objects, inheritance between classes, the attributes of each class together with their types and cardinality, and relationships between objects. In addition, the models typically allow the inclusion of user interface specific information. For example the model of object attributes often includes facets to indicate whether the attribute should be shown to the user, an ergonomic name, and other information to influence the choice of abstract interaction object to be used to specify the attribute.

The task models of these tools are also similar. Tasks are usually decomposed hierarchically, and information is included to specify the sequencing between the tasks (e.g., and, or, xor, parallel). Often, the task model includes references to the domain objects needed and produced in each task. The task model is used during automatic generation to determine the interface dialogue and to determine the information that should be shown in each window.

MB-IDEs in this category typically do not require developers to specify either the abstract or concrete specifications of the interface.

**Design Process.** Most automated design MB-IDEs use the following sequence of steps to automatically design an interface:

1. *Determine the presentation units.* This step essentially determines the windows that will be used, and what information will be shown in each window.
2. *Determine the navigation between presentation units.* This step computes a graph of presentation units that defines which units can be invoked from which other units.
3. *Determine the abstract interaction objects for each presentation unit.* The abstract interaction objects specify the behaviour of each element of a presentation unit in an abstract way (e.g., select one from set).
4. *Map abstract interaction objects into concrete interaction objects.* The concrete interaction objects represent the widgets available in the target toolkit.
5. *Determine the window layout.* This step determines the size and position of each concrete interaction object.

The first three steps build the abstract user interface specification, and the last two build the concrete specification.

**Post Editing.** Once the concrete specification is built, and the implementation tool generates the “toolkit-ready” file, the developer has the opportunity to use an interface builder to beautify the layout, change fonts, colours, add decorations, and perform other cosmetic enhancements.

### **3.1.2 Difficulties With Automated Design**

Even though automatic design MB-IDEs can produce interfaces with little or no development effort, there is concern about the quality of the generated interfaces. There is substantial evidence to indicate that it is not feasible to produce good quality interfaces for even moderately complex applications from just a data and task models

(together with simple annotations of the data model, such as flags that indicate whether object attributes are relevant to the user interface).

The chapters by Morten Harning [16] and by Stephanie Wilson and Peter Johnson [50] describe critical decisions that must be made in the design of an interface, which the automated design tools cannot currently make appropriately, and which do not seem feasible to automate.

Harning's paper contains an excellent example that illustrates the difficulty of automating steps 1 and 3. Harning's example is about a project management application where users want an interface to monitor progress in the various activities involved in a project. In this application there are four classes of objects represented in the data model: Employee, Project, Activity, Weekly Estimate, and Time Entry. Harning demonstrates using examples that of a good interface must satisfy the following properties:

- *Users need windows that show information drawn from multiple objects.* In the project monitoring example, the project display is based mostly on the Project object, but also shows attributes of the Employee and Activity objects. Furthermore, the example shows that the choice of attributes is task-dependent, and required developers to have a deep understanding of the user's tasks. This means that step 1 of the abstract design tool is hard, if not impossible to automate.

This property is achieved in the interfaces generated using Trident. The Trident task model captures the information needed for each task, and the generation algorithm calculates how the information flows between tasks in order to determine what information to show in each presentation unit, and where to place it. Systems like Janus, which only use the data model do not satisfy this property.

- *Users do not want the raw information, but rather they need the information to be re-structured and summarised.* In the project monitoring example, users want a weekly report display that essentially combines the Activity and Weekly Estimate objects on a weekly calendar display that shows how much effort was spent on each activity during a specific week. Re-structuring and summarisation cannot be done without a deep understanding of the user's tasks, and again points to the difficulty of automating step 1.

Another restructuring problem is that users want to see the names of people in the Project Leader field as "name (initials)". This means that rather than using two AIOs to present two different attributes, a single one should be used to present a combination of two attributes. This simple example suggests that the assignment of object attributes to AIOs (step 3) is also a hard problem.

- *Graphical displays are often more effective than tables and forms.* Harning's paper has an example of a graphical display that uses a plot with two curves to show how much time has cumulatively been spent on a project compared to the estimate of the time remaining to complete the project. This example shows that the set of AIOs need to be expanded to include more sophisticated elements such as plots. Of course, then the problem is how to select the appropriate one (step 3),

how to set all its parameters, and then how to map it to concrete interaction objects (step 4).

There are two main approaches to automatic design, one based on task models, and the other based on the domain model. The task model approach performs better because task models have some of the information to satisfy the properties listed above. The domain model approach does not have access to such information, and can only produce simple interfaces, typically with one object per presentation unit.

The requirements listed above point to deep issues of interface design, and raise questions about the utility of completely automating the design process, especially steps 1 and 3. Even a small amount of developer involvement can have a huge difference. A simple calculation reveals the economics of the situation.

Most of the automatically designed interface force users to bring up several windows to view the information they need to perform a task, rather than a single window with all the information. Ignoring issues about time to assimilate improperly structured information and the error rates that can result, bringing up several windows and closing them can easily take 3 additional seconds. If users do this 20 times a day, in a year, one full day will be lost per worker. If an organisation has 40 users, 2 man months will be lost per year. Surely it is worth to have developers spend several weeks working on a design.

### **3.1.3 Discussion**

The conclusion of this section is that none of the 5 steps should be completely automated. Rather, collaboration between developers and tools should be built in from the start. Tools should offer suggestions and alternatives. Developers make the decisions, accepting suggestions, choosing between alternatives or entering their own solutions.

This means that the abstract and concrete specification layers of the models should be available to the developers. The specification languages for these layers must allow developers to control all features of the interface that they want to control, no matter how low level. Emphasis should shift from automation to computer aided design.

A simple, and commonly used approach to computerised design aids is the post-editing approach. An automated generation tool generates a first draft of the design, and then the developer edits the draft to produce the final design. This approach has a serious shortcoming, namely that when developers change the model, they need to run the generator tool again, and the post-editing changes will be lost.

The post-editing approach has been used mainly to allow developers to beautify layouts. However, many MB-IDEs such as FUSE feature automatic generator of higher levels of abstraction, and run the risk of running into the same post-editor problems.

One solution to the post-editing problem is to record the changes performed during post-editing, and to reapply them to the output of the generation tools. This approach was used in early versions of Mecano, but it proved difficult to apply the changes

reliably, especially when new elements were introduced to a design, or old elements were deleted.

A more robust solution requires a deep integration of the computerised advisor and the modelling tools. In this approach the advisor tools produce design alternatives and suggestions that developers can incorporate into an evolving design via the modelling tools. There is no batch generation process followed by a refinement phase, but rather an incremental evolution of the design, where the computerised advisors and the developers incrementally build the design.

Several MB-IDEs are moving away from automation in the direction of computerised advisors. For example, the Tadeus [38] system requires developers to specify steps 1 and 2 in a structure called a dialogue graph. Steps 3 and 4 are table driven. The system builds default tables with default entries, but developers can edit these tables and override any entry. Step 5 is done automatically, but Tadeus supports post-editing of the generated implementation file.

The FUSE system described in [23] also provides a specification language and tool (BOSS [39]) that lets developers specify the abstract interface specification, and many aspects of the concrete specification. In addition, FUSE provides a tool (FLUID [2]) that uses the task and domain model to produce specifications that can be fed to the BOSS tool to refine and produce an interface. It is unclear for the published papers whether and how FUSE avoids the post-editing problem.

Trident is perhaps the most sophisticated and robust system that combines automatic generation and computerised advice. Trident developed many different strategies and algorithms for performing each of the 5 steps listed above. For example, they developed six strategies for defining presentation units, and have tools that can automatically select and apply a strategy based on information contained in the task and domain model. Trident also offers developers the option of choosing a strategy, or performing the step by hand. However, it is unclear from the published literature on Trident whether it uses an integrated approach as described above.

### **3.2 Retrospective – Specification-Based MB-IDEs**

MB-IDEs in this category seek to provide powerful interface specification languages. These languages provide effective layering or abstraction mechanisms that allow developers to express interface properties at a convenient level of abstraction to facilitate reuse and design modifiability. These languages also seek to give developers extensive control over all features of the interface, so that developers can express any design that they can think of. The goal is not to automate design, but rather to make it easy for developers to express designs, change designs, retarget designs to new platforms, new classes of users, new tasks, etc.

MB-IDEs in this category are oriented towards data management applications. Most business-oriented applications fall in this category, but many engineering and data visualisation applications do not, because they have interfaces whose graphical components are too complex to be expressed in their interface specification languages.

### 3.2.1 Structure of Specification-Based MB-IDEs

The structure of specification-based MB-IDEs is also compatible with the architecture shown in Fig. 1. They emphasise the model and the implementation tool, and typically do not have an automated design tool.

The modelling language of these MB-IDEs have facilities for developers to express models at the three different levels of abstraction shown in Fig. 1. The models of these MB-IDEs typically feature a data model, but not always a task model. The data model is used mostly in the implementation tool to generate the binding between the interface objects and the application data, so that the interface objects can access the application objects to retrieve the pieces of information that will be displayed (e.g., access the name field of a person object).

The modelling languages to specify the abstract and concrete user interface specifications are designed to maximise reuse. Even though the goals of the different MB-IDEs in this category are the same, the features of the modelling languages are different. For this reason, this section will not attempt to describe these languages in general terms, but rather uses the well known ITS system as an example. Other MB-IDEs in this category include BOSS, Humanoid and Mastermind.

### 3.2.2 ITS

The ITS system was developed by IBM research, and was used to construct several large applications such as the information kiosks for the Seville world fair, a purchasing system for a large corporation, an insurance industry application, and many others.

ITS has modelling components corresponding to the three levels of modelling shown in Fig. 1. The domain model is called a *data pool*, there is no task model, the abstract specification is called *content specification*, and the concrete specification is called a *style specification*.

The data pool definition language (domain model) supports the specification of structured objects and sequences of objects, like the domain model in many other MB-IDEs. The following is an example of the data pool specification for an airline reservation system.

```
list listname = flights, numrecords = 10
  field destination, rangename = cities, size = 20
  field departure_time, size = 10
  field departure_date, size = 20
  field airline, rangename = airlines, size = 20
  field number_stops, size = 5
```

The content specification (abstract user interface specification) of an interface consists of a collection of frames. Frames can contain lists, forms, choices, information blocks, and nested frames. These elements specify the information that will be presented to the user. Top-level frames correspond to presentation units. Lists and forms specify which elements of the data pool are to be shown in a frame. Information blocks

specify static pieces of information to be shown in a frame. Choices indicate sets of alternatives that can be chosen by the user, and correspond to AIOs. Each element specification can be elaborated using an extensive set of attributes that specify the interface content in detail. The following is a fragment of the content specification for the airline reservation example. This frame specifies that five flights are to be displayed, and specifies which fields of the flights object to display.

```
frame id = check_today, action = getlist, listname = flights, value = flights.data
  list listname = flights, number = 5
    list-item field = destination, message = "To"
    list-item field = departure_time, message = "Departure"
    list-item field = departure_date, size = 20
    list-item field = airline, message = "Carrier"
  frame message = "To search for selected flights"
...
```

The style specification (concrete user interface specification) specifies the mapping from AIOs to CIOs.

To quote from Wiecha's paper, "a style is a co-ordinated set of decisions on the appearance and behaviour of the interaction techniques used in a family of applications". Styles are specified using rules. The condition part of the rule can test any of the attributes of a frame or its children. The action part of the rule selects the CIO to use, and specifies values for the attributes.

Typically, the rule set for an application consists of general rules that apply to families of frames (e.g., there could be a rule for displaying choices as radio buttons), and specific rules that match specific frames defined in the content (e.g., a rule for the check\_today frame defined above). General rules are reused in multiple applications and within a single application. Specific rules are used to specify the features of a particular interface that make it different from the generic case.

The following is an example of a style rule. It specifies that if the content is a choice, then construct a vertical group of a title, and something else, depending on which of the nested conditions match. If only one element can be chosen, then the second component is a vertical group, or a collection of horizontal groups, one for every choice. The horizontal group consists of a dingbat to indicate radio buttons, and a message. Note that this rule does not completely specify the display of choices. Other rules may be used to determine the attributes of the unit types used within this rule (VertGroup, HorzGroup, Dingbat and Message).

```
:conditions source = choice
  unit type = VertGroup
  unit type = Title
  :eunit

:conditions kind = 1_and_only_1
  unit type = VertGroup
    unit type = HorzGroup, replicate = all
      unit type = Dingbat
      :eunit
```

```

:unit type = Message
:eunit
    :eunit
        :econditions
        ...
        ...
        :eunit
:econditions

```

The implementation tool of ITS consists of the rule interpreter and the run-time support system that fires the rules appropriate rules when actions are invoked and the contents of the data pool change.

### 3.2.3 Discussion

The main difference between specification-based systems such as ITS, and automated design tools such as Janus is one of philosophy. In specification-based MB-IDEs the modelling language is open, whereas in automated design tools it is closed. In automated design tools, developers can only control the design using a few attributes that the tool developers chose to export for that purpose, limiting the developers' ability to control the design of interfaces, and ultimately limiting the quality of the interfaces that can be generated.

Even though ITS is a specification-based MB-IDE, developers do not specify all the features of every individual window. The main point of ITS is that developers should not have to do that. Developers using ITS must specify the abstract user interface specification completely, that is, they have to specify the abstract interface for every different *kind* of window. As argued in the previous section, this is good because the abstract interface is precisely the hardest aspect to generate automatically. However, developers using ITS do not have to specify the concrete user interface specification completely. There is no automated designer to do it, but developers can reuse rule sets from libraries that contain the abstract to concrete mapping for significant portions of the interface specification. This reuse capability enables specification-based MB-IDEs to incorporate many of the cost savings capabilities of automated designers, while overcoming the most serious problems.

Other specification-based MB-IDEs such as Humanoid and Mastermind share the design philosophy of ITS, but differ in the nature of the modelling language. In a large logistics application developed using Humanoid, the developers were able to identify about 13 different families of windows to account for the more than 100 different windows that the system provided. Developers modelled those 13 windows so they did not have to specify each window separately, as would appear to be necessary with a pure specification-based system. However, the design of the 13 windows was according to user requirements, and it would not have been possible to design those windows automatically.

The BOSS system, briefly described in Loczewski's and Schreiber's chapter, is another example of a specification-based MB-IDE. BOSS is also a module of the

FUSE system, which is a mixture between automated designer, as implemented in its FLUID module, and a specification system.

### **3.3 Retrospective – Help Generation**

Many MB-IDEs [23, 26, 32, 33, 42] have the ability to automatically, or semi-automatically generate a help system for an application based on model information used to construct the user interface in the first place.

Cartoonist [42] was the first system to provide a compelling demonstration of help generation. Cartoonist allowed the user to ask “how do I do X?” questions, where X could be any of the actions of an application. In response, it would show an animation showing the exact actions that the user needed to perform with the mouse and keyboard to invoke the action. A typical example would show the mouse selecting an object (if one was not selected), then pulling down the appropriate menu, filling out a dialogue box, and finally clicking the OK button.

Cartoonist used the UIIDE interface models. The abstract interface specification of UIIDE describes the actions that users can perform. The action specification contains pre-conditions that specify the contexts in which the action can be performed, and post-conditions that specify how actions modify the context. The concrete specification models the mapping between actions and concrete interaction objects. Using this information, Cartoonist was able to construct a plan with the sequence of interaction techniques that needed to be invoked in order to perform an action. Cartoonist could even determine what other actions need to be invoked before in order to modify the context to satisfy the preconditions of the action being explained. This allowed the user to ask for help at any time, even when the context was not appropriate to perform the action.

Humanoid also generated a help system for an application based on the model [26]. The help system provided hypertext help to explain the information displayed in a region selected by the user (e.g., paper.txt represents a file), and explain all the commands that the user could issue (e.g., paper.txt can be selected by clicking with the left button, and then the commands delete, and grep can be applied to it). An important contribution of the Humanoid help system is that it used an example-based technique to assist developers in specifying the text of the help windows. Humanoid first generated text automatically, but developers could select text fragments to edit the wording, and then Humanoid would interact with the developer to find an appropriate place in the model to store the edited text fragment. Placement in the model determined the contexts in which the text fragment would appear.

The chapter by Contreras and Saiz in [8] illustrates how the knowledge in the models can be used to automatically generate software tutors, and how the tutors can be customised to different classes of users with different tutoring needs and preferences.

The chapter on the FUSE system, also describes how the information in a model can be used to construct a help system. FUSE, like Cartoonist, produces context sensitive help using the model information. It uses a different style of modelling and also delivers the help in HTML pages rather than using animation.

### **3.4.1 Discussion**

The ability to generate help systems using the information contained in the model is one of the main benefits of the model-based technology. All of today's applications feature a help system, and significant development effort must be devoted towards implementing it. Context-sensitive help is especially difficult to implement because it must reference internal data structures of the interface in order to query the current context of the interface.

The next sections argue that it is precisely the ability to generate runtime services such as help, that give the model-based technology an edge over conventional technologies for implementing interfaces. Using conventional technologies, each runtime service must be separately designed and implemented. Using the model-based technology the services are generated for free, or for a small incremental cost. The reason is that the services use the same information that is used to build the interface in the first place. In addition, as an interface design evolves, the services automatically evolve with it to remain consistent with the design.

### **3.5 Retrospective – Modelling Tools**

Interestingly, ITS, the most widely used MB-IDE does not have a graphical modelling tool. Developers must learn the syntax of the modelling language, and enter the models using a text editor. The creators of ITS found that developers learn the syntax of the language quickly, and that the lack of a modelling tool is not an obstacle to using the tool. They also report (personal communication) that a syntax directed editor was built, but developers refused to use it.

The lesson to be learnt from this experience is that it is false that some tool is better than no tool. A text editor is a powerful tool that is always available. Its most attractive features are users know how to navigate with it, that it is very fast, that it provides cut and paste, effective search mechanisms, global replace, the ability to easily comment out pieces of a design, etc.

However, experience with widely used CASE tools, and expert system shells such as Nexpert Object [28] and Kappa [36] suggest that well engineered graphical tools for building models are useful for the development of large applications. They can be better than text editors, but they must be well engineered, and designed to support large applications.

Most MB-IDEs feature simple forms-based interfaces for creating and editing model entities. Some MB-IDEs such as FUSE and Adept provide visual modelling tools. These tools have not been extensively used, so it is early to comment about their usability for developing large applications.

An interesting approach to modelling tools is embodied in a tool called Grizzly Bear [13]. This tool tries to hide from developers the intricacies of the models by providing an interface that looks like a traditional interface builder or a drawing editor. The interface provides a palette of building blocks and a drawing area where developers can draw pictures of the interface. Grizzly Bear builds models by demonstration. It

extracts model entities from the example interfaces that developers draw. It can generalise different pictures into different classes, and most importantly, it can infer dialogue fragments from before and after snapshots of an interface. Grizzly Bear was used to completely build the model for a simple drawing editor based on demonstrations of how the editor should work. An interesting feature of this tool is that it shows developers a textual view of the model as it is being constructed. This view helps novice developers learn the modelling language, and allows experienced developers to edit the textual representation directly. Grizzly Bear represents the first step towards this kind of tool, and further progress needs to be made before such a tool is ready for serious application development.

### **3.6 Retrospective – Design Critics and Advisors**

Much work on design critics and advisors has been done in the context of model-based tools [4, 10]. The reason is that in order to evaluate a design, and automated critic has first to analyse the design to determine what it does. The models provide rich information for critics and advisors to do their work.

The following kinds of evaluation tools have been investigated.

*Property verification.* The tool verifies that a design satisfies certain properties (e.g., all application functionality is reachable). Some tools [11, 32] can only verify a set of pre-defined properties encoded in a knowledge-base. More powerful tools [35] allow developers to specify the properties to be verified.

*End-user simulation.* These tools [21] simulate a user interacting with an application, and make predictions about times to perform tasks, learning times and likely errors.

*Summative evaluation.* These tools produce numbers that can be used to rank designs. An example of such a tool is AIDE [40], a tool to compute metrics based on a theory of layout quality. Work on such tools is still very preliminary. The chapter by Comber and Maltby [7] describes experiments designed to validate the results of some of these tools.

Many property verification tools [24] are designed to detect violations of standard user interface guidelines (e.g., File menu should have the mnemonic F). These tools play a similar role to spelling checkers in word processors: they detect surface problems that show a lack of professionalism. They do not detect problems related to the semantics of the interface, but nevertheless, they are very useful.

Most style-guide verification tools are not model-based, but rather take as input the toolkit ready file used in well known toolkits (e.g., resource files for Windows, UIL files for Motif). The limitations of these tools are discussed in the paper by Farenc et. al. [9]. The problem is that the toolkit-ready file does not contain enough information about a design to verify many of the style rules. In the context of ERGOVAL, 44% of the rules can be automatically verified using the toolkit-ready file, and up to 78% could be automated if the evaluation tool had access to appropriate information. The model-based approach to interface development should allow tools to get closer to the 78% limit. For example, Farenc et. al. illustrate the limitations of toolkit-ready files

with the rule that states that “for any input, if there are any acceptable values, such values must be displayed.”. Such a rule can be automated in the context of most MB-IDEs because their models contain information about the acceptable values for inputs, and information about how the inputs are displayed.

There are a few notable examples of design critics aimed at more fundamental design issues, addressing issues similar to grammar and document content in word processing. These critics require very detailed models, more detailed than the models currently being used in most MB-IDEs. One example of such a tool is NGOMSEL [21], which belongs to the end-user simulation category of design critics. NGOMSEL takes as input a detailed task model where the leaf tasks represent interaction techniques (CIO). It can simulate a user interacting with the application, and predict how long it will take an expert user to complete a high level task. NGOMSEL can also make predictions about features of an interface that users will find difficult to learn.

Another example of a sophisticated design critic is embodied in the work of Fabio Paterno [35]. His critic is a property verification critic that uses detailed models of an application specified using the LOTOS [34] notation. His system allows developers to specify complex properties using a notation based on temporal logic. One of Paterno’s papers [35] discusses an interesting example about an air traffic controller application that uses a message area to display messages to the user. The last message to arrive is shown in the message area, and the previous ones are queued until the operator gets around to view them. This design could lead to subtle timing problems where operators delete the wrong message, skip viewing a message, etc. His paper shows how required properties of this interface can be verified (e.g., the user can read a message several times), or how undesirable effects can occur (e.g., user unwittingly deletes the wrong message). The expense of building the complex models required by this critic can be justified in safety critical applications such as air-traffic control.

Much work remains to be done before these advanced critics become a useful tool for developers. These critics require detailed models that are time-consuming to build, and expressed in specialised notations that most developers do not know. However, work is in progress to integrate these tools with MB-IDEs (NGOMSEL with Mastermind [5], Paternó is working on an implementation tool for his notation). Once this work is complete these design critics will have a more substantial impact on the design and development of interfaces.

An interesting question is the extent to which MB-IDEs can render style-guide verification tools unnecessary because the kinds of errors that they detect cannot be committed when using an MB-IDE.

Automatic generation MB-IDEs provide one answer to this question. The design algorithms of these tools are based on style-guides, so they will automatically be obeyed. Most violations will be due to exceptions specifically coded in the design algorithms.

Design advisors provide a different answer to this question, in the context of specification-based MB-IDEs. Design advisors can be viewed as pro-active critics. Rather than telling designers what they did wrong, they try to steer designers away

from poor design choices. The most attractive feature of automated design advisors is that they complement specification-based MB-IDEs so that developers do not have to construct specifications on their own, but are assisted by advisors whose knowledge-bases codify expert knowledge and wisdom about interface design.

The work on design advisors has not yet reached a level of maturity that allows a critical discussion of their approach and effectiveness. Two well known systems are Trident and Expose [14].

## 4 Challenges and Opportunities

The main opportunities for model-based interface technology lie ahead because it is better suited than traditional technology to meet the new interface challenges that technology is creating.

Faster machines and networks enable more and more sophisticated applications, providing users with more capabilities and more information, but at the same time overwhelming them with more commands and options. Interfaces will need to become more intelligent to assist users in performing their tasks, to help them come up to speed in a new application, to allow users to customise them to make them effective for the particular tasks that users perform most often.

Laptops are commonplace. Smaller portable devices such as PDAs and pagers are getting linked to the networks and provide the ability to access the same information that is available via workstations and laptops. The need will arise for applications that scale across a wide range of devices to provide users with the same or a scaled down version of the workstation functionality. Scaled up versions will also be needed to take advantage of wall-sized displays.

New modalities such as speech, natural language, hand-writing recognition are maturing. Applications will need to reconfigure their interfaces to take advantage of whatever modalities are available on the user's platform.

The following sections discuss why the model-based technology is well positioned to meet these challenges, and give some suggestions on how MB-IDEs need to evolve.

### 4.1 Challenge 1 – Task-Centred Interfaces

The main difficulty that users face when interacting with an application is to figure out how to use the capabilities of the application to perform desired tasks. Applications often offer many dozens of commands and options, so it is difficult for users to learn and remember the sequence of commands needed to perform a task. Many of the most popular and complex applications such as Microsoft Office and its competitors attempt to cope with this problem by offering *task assistants*. For example, Microsoft Excel has assistants to construct charts, to pivot tables, to create templates, etc. Microsoft Word has assistants to format tables, to format documents, to correct spelling, to do mail merge, etc. The typical behaviour of an assistant is to analyse the current context (e.g., the array of selected cells in a spreadsheet), and then ask users a

sequence of questions about how they want the task performed, and finally perform the task for the user. Assistants make certain tasks easy to perform, even if they limit the set of options that users have.

Related to task assistants are *guidance systems*. Guidance systems have two main components, an indexing component that helps users find the topic they need guidance on, and a component that guides the user in performing the task. For example, Microsoft's answer wizard, a kind of guidance system, allows users to index in several ways: they can use keywords to find topics, or they can browse the hierarchy of topics. Guidance is given to users using the task assistant technology, traditional hypertext help windows, enhanced hypertext windows with buttons to invoke relevant application functionality.

Today's task assistants and guidance systems are implemented separately from the interface, most surely at a significant development cost. Developers of these systems must, at least informally, build a model of the tasks that users are expected to perform. For the task assistants they must encode in detail all the steps for performing the task, taking into account all the contingencies that arise from the different contexts in which the assistant is invoked. For the guidance systems, developers must encode a comprehensive model of the tasks, including the words that can be used to index them, the steps for each task, pointers to application commands that perform particular steps, etc.

One of the challenges and opportunities for model-based technology is to partially automate the generation of task assistants and guidance systems. Many MB-IDEs use a task model to assist with the design of the interface, and also to control the dialogue at runtime. Such task models already contain much of the information needed for task assistants and guidance systems. They already contain a representation of all the tasks, the steps to perform each task, sequencing constraints, information needed for each task, etc. The abstract and concrete interface representation contain the information that links tasks to the interaction techniques that invoke the various steps of a task. It seems quite sensible to enhance the task model representation to include any additional information needed for the task assistants and guidance systems, and to generate these services from the model.

As mentioned in a previous section, significant progress has been made in this direction. What is needed is to make the transition from an interesting feasibility demonstration, to a robust, high quality implementation. Current demonstrations of help generation work for some of the tasks, not all, generate poor quality text full of the internal names of objects (e.g., `start1stConnection`), and for the most part, have never been user tested or formally evaluated.

The comparison between automated design tools and specification-based tools is relevant here. Automatic interface generation systems offer interesting demonstrations, but only systems like ITS become successful, because they provide developers with appropriate control over the design. Likewise, model-generated task assistants and help generation systems will achieve high enough quality only if developers of these systems can exert *complete* control over the text that is produced, and significant control over the format.

## **4.2 Challenge 2 – Multi-Platform Support**

Most of the user interface tools developed during the late 1980's and early 90's were designed for a canonical platform featuring a mouse, a keyboard, and a 13 inch colour monitor. Today's platforms are stretching the limits of the canonical platform, often yielding hard to use interfaces. Large, high resolution monitors cause the displays of some applications to become unusable because the icons and text become hard to see, and hard to point at with the mouse. Smaller displays, such as laptop 9 inch displays result in some applications using almost all the screen space for menus, toolbars, dialogue boxes, leaving users a tiny window to perform their work. As argued before, the situation will get much worse once radically smaller (PDA, pager) and larger devices become popular (wall displays).

Interfaces developed using traditional interface builders and toolkits are hard to adapt to different platforms because developers must redesign each window for each new platform. As the set of platforms proliferates, this becomes expensive.

Model-based technology offers a much better approach. For qualitatively similar devices (e.g., workstation and laptop), changes in the AIO to CIO mapping, and the CIO parameters are typically enough to appropriately scale the interface. More radical changes can be done by redesigning the abstract interface specifications. The important point to bear in mind is that in a system like ITS the amount of work is proportional to the number of style rules, which is typically much smaller than the number of windows.

ITS has demonstrated the usefulness of this approach by refining style rules to port interfaces to use a touch screen rather than a mouse. The change involves making the target areas larger, and increasing the spacing between adjacent target areas.

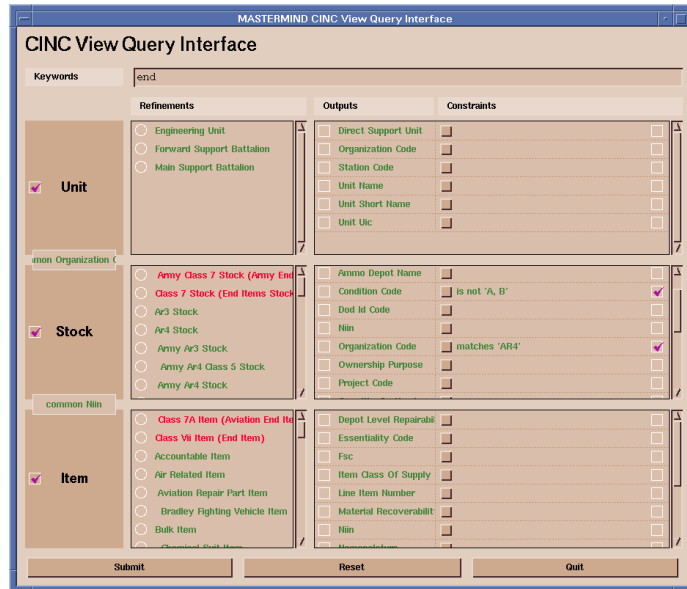
One of the interesting challenges in this area is to develop techniques to scale interfaces to radically different platforms, such as PDAs and pagers. Fig. 2 shows an example of a first step in this direction. The figure shows simple adaptations explicitly represented in Mastermind's model that cause an interface to adapt to changes in screen size by progressively removing less important information as the available space becomes smaller. The first adaptation causes the first column of scrolling areas to be replaced by buttons that bring up pop-up windows with the same information. The second adaptation causes some headings to disappear and remaining heading fonts to become smaller.

## **4.3 Challenge 3 – Interface Tailoring**

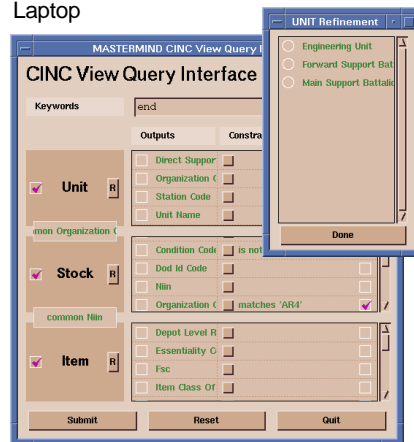
Interface tailoring refers to the ability to customise and optimise an interface according to the context in which it is used. Interfaces can be tailored to tasks that different segments of the user population need to perform most often, to the level of use and experience of users, to the physical abilities of users, to platform characteristics, etc.

There is a whole spectrum of tailoring possibilities. Interface tailoring can happen at the factory, that is, developers produce several versions of an application tailored according to different criteria. Tailoring can also be done at the user's side, for

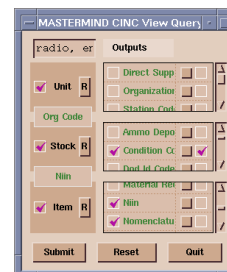
### Workstation



### Laptop



### PDA



**Fig. 2** Scaling an Interface to Multiple Platforms in MASTERMIND

instance, by system administrators or experienced users. In the extreme, individual users might tailor the interfaces themselves, or the interface could adapt on its own by analysing the user's patterns of use.

No matter when tailoring happens, and what interface features are tailored, tailoring involves modifying the interface design. The simplest level of tailoring happens at the concrete level of an interface specification where features such as the layout, colours and fonts of an interface are changed. More sophisticated tailoring can happen at the

abstract interface specification where the dialogue gets modified, for example to shortcut certain steps, to rearrange the order for performing steps, etc. At the highest level, new tasks might be defined by composing existing tasks.

Many model-based interface tools address some aspects of interface tailoring. For example, the FUSE system presents examples of how an interface can be tailored according to the user's level of experience. However, most of the examples are about factory tailoring, where developers construct the rules that define how the interfaces should adapt depending on certain contextual information such as a simple user model.

However, it should be possible to use the automatic interface generation capabilities of many MB-IDEs to support end-user, or administrator-level tailoring of interfaces. Such a facility would be a compelling example of the benefits of the model-based technology.

#### **4.4 Challenge 4 – Multi-Modal Interfaces**

New input modalities such as speech, natural language and pen gestures have matured to the point where they can be effectively used in practical applications. Currently, applications that take advantage of these modalities are custom built without much tool support.

Building interfaces that combine these modalities with traditional graphical elements is hard for several reasons:

- To incorporate speech and natural language developers must define the lexicon and perhaps the grammar for parsing and interpreting natural language sentences.
- Speech, natural language and pen input are intrinsically ambiguous. No matter how good the recognisers get, they will always produce a set of alternative interpretations with levels of confidence, rather than a single certain interpretation. Interfaces must be designed to cope with this uncertainty.
- Users can speak and point at the same time, and the interpretation of the inputs depends on their relative timing. In addition, the inputs from the various modalities can refer to each other (e.g., put this file <click> there <click>).

New output modalities such as 3D graphics are also becoming cheaper and commonplace.

Currently, not many model-based interface systems are addressing the construction of interfaces that use these modalities. The model-based interface community runs the risk that the architectures and tools that are being developed will not work with these modalities.

One notable exception is the work by Phil Cohen [6]. His system provides an open architecture for the development of multi-modal user interfaces. The system uses a blackboard architecture that allows an open-ended set of agents to collaborate. Agents collaborate to perform user tasks, to disambiguate natural language requests, etc.

## 5 Conclusion

Much progress has been made towards demonstrating that the model-based approach provides a viable and effective new technology for developing user interfaces. Model-based systems have evolved from simple proof of concept prototypes that were used on toy applications, to powerful systems that address the construction of interfaces for realistic applications (ITS, Trident, Janus, Mastermind, etc.) The models of many MB-IDEs have been integrated with mainstream software engineering modelling techniques such as OOA (Janus), ER models (Trident, GENIUS [19]), making it easier to use these tools together with other well established software engineering methodologies.

As a community, we need to make progress in two fronts. The first is to build compelling demonstrations of the benefits of model-based tools. The interesting demonstrations of help generation, platform scalability, design critics need to be proven in more realistic settings with realistic applications.

The second front is to address the challenges being posed by new technology developments. As discussed in the last section, these challenges are in fact opportunities for the model-based technology. The challenges point towards solutions where models play an important role, so the model-based technology is well positioned to address them.

## References

1. Balzert, H., Hofmann, F., Kruschinski, V., Niemann, C.: *The JANUS Application Development Environment-Generating More than the User Interface*. In: Vanderdonckt J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 183-207).
2. Bauer, B., *Generating User Interfaces from Formal Specifications of the Application*, In: Vanderdonckt J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 141-157).
3. Beshers, C.M., Feiner, S.K.: *Scope: Automated Generation of Graphical Interfaces*. In Proceedings of UIST'89. New York: ACM Press 1989 (pp. 76-85).
4. Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Vanderdonckt, J.: *Computer-Aided Window Identification in TRIDENT*. In Nordbyn K., Helmersen P.H., Gilmore D.J., Arnesen S.A. (eds.): Proceedings of INTERACT'95, London: Chapman & Hall 1995 (pp. 331-336).
5. Byrne, M.D., Wood, S.D, Sukaviriya, P., Foley, J.D, Kieras, D.E.: *Automating Interface Evaluation*. In Adelson, B., Dumais S., Olson J. (eds.): Proceedings of CHI'94. New York: ACM Press 1994 (pp. 232-237).
6. Cohen, P.R., Cheyer, A., Wang, M., Baeg, S.C.: *An Open Agent Architecture*. In AAAI Spring Symposium (pp. 1-8).

7. Comber, T., Maltby, J.: *Investigating Layout Complexity*. In: Vanderdonck J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 211-229).
8. Contreras, J., Saiz, F.: *A Framework for the Automatic Generation of Software Tutoring*. In: Vanderdonck J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 171-182).
9. Farenc, Ch., Liberati, V., Barthet, M.-F.: *Automatic Ergonomic Evaluation: What are the Limits?* In: Vanderdonck J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 159-170).
10. Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., Sumner, T.: *Embedding Computer-Based Critics in the Context of Design*. In Ashlund S., Mullet K., Henderson A., Hollnagel E., White T. (eds.): Proceedings of INTERCHI'93. New York: ACM Press 1993 (pp. 157-164).
11. Foley, J.D.: *History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation*. In Paternó F. (ed.): Proceedings of DSV-IS'94. Berlin: Springer-Verlag 1995 (Focus on Computer Graphics Series, pp. 3-14).
12. Foley, J.D., Kim, W.C., Kovacevic, S., Murray, K.: *UIDE - An Intelligent User Interface Design Environment*. In Sullivan J.W., Tyler S.W. (eds.): Intelligent User Interfaces. New York: ACM Press 1991 (pp. 339-384).
13. Frank, M.: *Grizzly Bear: A Demonstrational Learning Tool For A User Interface Specification Language*. In van der Veer G.C., Bagnara S., Kempen G.A.M. (eds.), Proceedings of UIST'95. New York: ACM Press 1995 (pp. 75-76).
14. Gorny, P.: *EXPOSE - An HCI-Counseling for User Interface Design*. In Nordbyn K., Helmersen P.H., Gilmore D.J., Arnesen S.A. (eds.): Proceedings of INTERACT'95, London: Chapman & Hall 1995 (pp. 297-304).
15. Green, M.: *A Survey of Three Dialogue Models*. ACM Transactions on Graphics, Vol 5, No. 3, 244-275 (July 1986).
16. Harning, M.: *An Approach to Structured Display Design - Coping with Complexity*. In: Vanderdonck J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 121-138).
17. Hinrichs, T., Bareiss, R., Birnbaum, L., Collins, G.: *An Interface Design Tool based on Explicit Task Models*. In Tauber M.J., Bellotti V., Jeffries R., Mackinlay J.D., Nielsen J. (eds.): Companion Proceedings of CHI'96. New York: ACM Press 1996 (pp. 269-270).
18. Jacob, R.J.K.: *A Specification Language for Direct-Manipulation User Interfaces*. ACM Transactions on Graphics, Vol. 5, No. 4, 283-317 (October 1986) .
19. Janssen, C., Weisbecker, A., Ziegler, J.: *Generating User Interfaces from Data Models and Dialogue Net Specifications*. In Ashlund S., Mullet K., Henderson A.,

- Hollnagel E., White T. (eds.): *Proceedings of INTERCHI'93*. New York: ACM Press 1993 (pp. 418-423).
20. Johnson, P., Johnson, H., Wilson, S.: *Rapid Prototyping of User Interfaces Driven by Task Models*. In J. Carroll (ed.): *Scenario-Based Design: Envisioning Work and Technology in System Development*. London, John Wiley & Sons 1995 (pp. 209-246).
  21. Kieras, D.E.: *A Guide to GOMS Model Usability Evaluation Using NGOMSL*. In Helander M., Landauer T. (eds.): *The handbook of human-computer interaction*. Amsterdam: North-Holland 1996.
  22. Kim, W.C., Foley, J.D.: *Providing High-level Control and Expert Assistance in the User Interface Presentation Design*. In Ashlund S., Mullet K., Henderson A., Hollnagel E., White T. (eds.): *Proceedings of INTERCHI'93*. New York: ACM Press 1993 (pp. 430-437).
  23. Lonczewski, F., Schreiber, S.: *The FUSE-System: an Integrated User Interface Design Environment*. In: Vanderdonckt J. (ed.): *Proceedings of CADUI'96*. Namur: Presses Universitaires de Namur 1996 (pp. 37-56).
  24. Löwgren, J., Nordqvist, T.: *Knowledge-Based Evaluation as Design Support for Graphical User Interfaces*. In Bauersfeld P., Bennett J., Lynch G. (eds.): *Proceedings of CHI'92*. New York: ACM Press 1992 (pp. 181-188).
  25. Luo, P., Szekely, P., Neches, R.: *Management of Interface Design in Humanoid*. In Ashlund S., Mullet K., Henderson A., Hollnagel E., White T. (eds.): *Proceedings of INTERCHI'93*. New York: ACM Press 1993 (pp. 107-114).
  26. Moriyón, R., Szekely, P., Neches, R.: *Automatic Generation of Help from Interface Design Models*. In C. Plaisant (ed.): *Proceedings of CHI'94*. New York: ACM Press 1994 (pp. 225-231).
  27. Myers, B.A.: *User Interface Software Tools*. *ACM Transactions on Computer-human Interaction*, Vol. 2, No. 1, 64-103 (March 1995).
  28. *Neuron Dataelements Environment*. <http://www.neurondata.com/>
  29. Olsen, D.R.: *A programming language basis for user interface management*. In Bice K., Lewis C. (eds.): *Proceedings of CHI'89*. New York: ACM Press 1989 (pp. 171-176).
  30. Olsen, D.R.: *SYNGRAPH: a Graphical User Interface Generator*. *Computer Graphics*, Vol. 23, No. 3, 43-50 (July 1983).
  31. Olsen, D.R.: *MIKE: The Menu Interaction Kontrol Environment*. *ACM Transactions on Information Systems*, Vol. 5, No. 4, 318-344 (1986).
  32. Palanque, P., Bastide, R.: *Contextual Help for Free with Formal Dialogue Design*. In Alty J.L., Diaper D., Guest S. (eds.): *Proceedings of HCI'93*. Cambridge: Cambridge University Press 1993.

33. Pangoli, S., Paternó, F.: *Automatic Generation of Task-oriented Help*. In van der Veer G.C., Bagnara S., Kempen G.A.M. (eds.), Proceedings of UIST'95. New York: ACM Press 1995 (pp. 181-187).
34. Paternó, F., Faconti, G.: *On the Use of LOTOS to Describe Graphical Interaction*. In Monk A., Diaper D., Harrison M.D. (eds.): Proceedings of HCI'92. Cambridge: Cambridge University Press 1992 (pp. 155-174).
35. Paternó, F., Mezzanotte, M.: *Formal Verification of Undesired Behaviours in the CERD Case Study*. In Bass L., Unger C. (eds.): Engineering for Human-Computer Interaction, Proceedings of EHCI'95. London: Chapman & Hall 1995 (pp. 213-226).
36. *PowerModel® The Object Power Tool*. <http://www.intellicorp.com/power-model.html>
37. Puerta, A.: *The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development*. In: Vanderdonck J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 19-36).
38. Schlungbaum, E., Elwert, T.: *Automatic User Interface Generation from Declarative Models*. In: Vanderdonck J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 3-18).
39. Schreiber, S.: *Specification and Generation of User Interfaces with the BOSS-System*. In Blumenthal B., Gornostaev J., Unger C. (eds.): Proceedings of EWHCI'94. Berlin: Springer-Verlag 1994 (Lecture Notes in Computer Sciences, vol. 876, pp. 107-120).
40. Sears, A.: *AIDE: A Step Toward Metric-Based Interface Development Tools*. In van der Veer G.C., Bagnara S., Kempen G.A.M. (eds.), Proceedings of UIST'95. New York: ACM Press 1995 (pp. 101-110).
41. Singh, G., Green, M.: *Automating the Lexical and Syntactic Design of Graphical User Interfaces: The UofA\* UIMS*. ACM Transactions on Graphics, Vol. 10, No. 3, 213-254 (July 1991).
42. Sukaviriya, P., Foley, J.D.: *Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help*. In Proceedings of UIST'90. New York: ACM Press 1990 (pp. 152-166).
43. Szekely, P., Luo, P., Neches, R.: *Facilitating the Exploration of Interface Design Alternatives: The Humanoid Model of Interface Design*. In Bauersfeld P., Bennett J., Lynch G. (eds.): Proceedings of CHI'92. New York: ACM Press 1992 (pp. 507-514).
44. Szekely, P., Luo, P., Neches, R.: *Beyond Interface Builders: Model-Based Interface Tools*. In Ashlund S., Mullet K., Henderson A., Hollnagel E., White T. (eds.): Proceedings of INTERCHI'93. New York: ACM Press 1993 (pp. 383-390).
45. Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J., Salcher, E.: *Declarative interface models for user interface construction tools: the Mastermind*

- approach*. In Bass L., Unger C. (eds.): Engineering for Human-Computer Interaction, Proceedings of EHCI'95. London: Chapman & Hall 1995 (pp. 120-150).
46. Vanderdonckt, J.: *Automatic Generation of a User Interface for Highly Interactive Business-Oriented Applications*. In Plaisant C. (ed.): Companion Proceedings of CHI'94. New York: ACM Press 1994 (pp. 41 & 123-124).
47. Vanderdonckt, J.: *Knowledge-Based Systems for Automated User Interface Generation: the TRIDENT Experience*. Technical Report RP-95-010. Namur: Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique 1995. Available at <http://www.info.fundp.ac.be/cgi-bin/pub-spec-paper?RP-95-010>.
48. Wiecha, C., Bennett, W., Boies, S., Gould, J., Green, S.: *ITS: A Tool for Rapidly Developing Interactive Applications*. ACM Transactions on Information Systems, Vol. 8, No. 3, 204-236 (July 1990).
49. Wiecha, C., Bennett, W.: *Generating Highly Interactive User Interfaces*. In Bice K., Lewis C. (eds.): Proceedings of CHI'89. New York: ACM Press 1989 (pp. 277-282).
50. Wilson, S., Johnson, P.: *Bridging the Generation Gap: From Work Tasks to User Interface Designs*. In: Vanderdonckt J. (ed.): Proceedings of CADUI'96. Namur: Presses Universitaires de Namur 1996 (pp. 77-94).