Declarative interface models for user interface construction tools: the MASTERMIND approach

P. Szekely[1], P. Sukaviriya[2]
P. Castells[3], J. Muthukumarasamy[2], E. Salcher[4]
[1]University of Southern California, Information Sciences Institute (szekely@isi.edu)
[2]Georgia Institute of Technology (noi@cc.gatech.edu, jk@cc.gatech.edu)
[3]Universidad Autonoma de Madrid (castells@lola.iic.uam.es)
[4]University of Technology, Graz (salcher@icg.tu-graz.ac.at)

Abstract

Currently, building a user interface involves creating a large procedural program. Model-based programming provides an alternative new paradigm. In the model-based paradigm, developers create a declarative model that describes the tasks that users are expected to accomplish with a system, the functional capabilities of a system, the style and requirements of the interface, the characteristics and preferences of the users, and the I/O techniques supported by the delivery platform. Based on the model, a much smaller procedural program then determines the behavior of the system.

There are several advantages to this approach. The declarative model is a common representation that tools can reason about, enabling the construction of tools that automate various aspects of interface design, that assist system builders in the creation of the model, that automatically provide context sensitive help and other run-time assistance to users. The common model also allows the tools that operate on it to cooperate. Because all components of the system share the knowledge in the model, this promotes interface consistency within and across systems and reusability in the construction of new interfaces. The declarative nature of the model allows system builders to more easily understand and extend systems.

This paper describes the modeling language of MASTERMIND, a model-based user interface development environment.

Keywords

Model-based interfaces, knowledge-based interface tools, UIMS, user interface design environments

1 INTRODUCTION

Model-based interface development is a new paradigm for constructing interfaces. In the model based approach, interfaces are automatically generated from a declarative specification (model) that describes the tasks users need to perform, the content, structure and layout of displays, and the role that display elements play in user's tasks. Developers

using the model based paradigm build interfaces by building the model that describes the desired interface, rather than by writing a program that exhibits the desired behavior.

The model-based paradigm offers many potential benefits over traditional methods of building interfaces.

- *Powerful design and run-time tools*. The declarative model is a common representation that tools can reason about, enabling the construction of tools to assist developers at design-time, and end-users at run-time. Examples of design-time tools are design critics [4, 5], which automatically analyze designs to detect questionable features, automated advisors to help developers refine designs, and automated design tools that can automatically create certain portions of the interface [9, 21]. Examples of run-time tools are automatically generated context-sensitive help [23, 31, 32], and support for end-user customization.
- *Consistency and reusability*. Because all components of the system share the knowledge in the model, this promotes interface consistency within and across systems and reusability in the construction of new interfaces.
- *Support for early conceptual design*. Models encourage designers to explicitly represent the rationale for design decisions, thus encouraging designers to think more about the artifacts they are building.
- *Iterative development*. Since models are executable even before all details of the interface have been designed, developers can experiment with designs early in the development process, catching design flaws early, before considerable coding effort has been spent, and more resistance to change has built up.

Several model-based interface development tools have been built [10, 11, 17, 18, 28, 29, 33, 34, 35, 36], but none has achieved a level of maturity to allow them to generate industrial strength applications. The main shortcomings of today's model-based tools are:

- *Lack of flexibility*. The modeling language of existing model-based tools is not expressive enough to give developers adequate ways to control all the features of the interface needed for real applications.
- *Poor performance*. Most model-based tools are experimental, and thus not tuned for performance. However, a common cause of inefficiencies is that many tools interpret the models at run-time, i.e., when the interface is being generated. Unless the models are suitably restricted, this level of interpretation leads to poor performance. Some notable exceptions are ADEPT, which compiles the models into executable code, and ITS [36] which interprets the model at run-time, but uses a model that is less expressive than those used in other tools.
- *Hard to use*. Most model-based tools are hard to use, especially when compared with interface builders. Most model-based tools require models to be specified in a specialized modeling language. Thus modeling becomes a form of programming, which is not a skill many interface developers have or wish to learn.

MASTERMIND is a new model-based interface development environment designed to address the main shortcomings of existing model-based tool. MASTERMIND represents the continuation of the work on HUMANOID and UIDE, two different but complementary model-based systems. HUMANOID's strength lies in the presentation model, modeling tools and performance, where as UIDE's strength lies in the dialogue model, the design critics, and the help generation tools. MASTERMIND is being designed to capitalize on the best features of HUMANOID and UIDE, and to try to avoid the shortcomings.

This paper documents the MASTERMIND modeling language in detail, discussing most modeling constructs available in MASTERMIND. The main desiderata in designing the MASTERMIND modeling language were:

- *Expressive power*. MASTERMIND is designed to give interface designers extensive control over all interface features. This goal is achieved by allowing developers to model interfaces at different levels of abstraction. The higher levels are easier to specify, but offer less control, where as the lower levels offer more control at the expense of specification cost. MASTERMIND is designed to support mainly the specification of traditional 2D graphical user interfaces.
- *Amenable to interactive specification*. MASTERMIND's modeling language was designed so that models can be easily specified using interactive modeling tools that hide the syntax of the language completely. Many aspects of the model were designed so that they can be specified by demonstration.

  To achieve this goal we did many paper designs of how the modeling tools would work, and how they could be used to construct the models expressible in the language. We built a mockup of the design environment using Macromedia Director, to concretize our vision of the design environment, and help guide the design of the modeling language. Often, we sacrificed expressivity, or provided multiple ways of expressing certain features in order to achieve this goal.
- *Compilation into efficient representation*. For model-based tools to be successful it must be possible to translate the model into an efficient representation for use at run-time. We elected to use a powerful declarative representation at design-time, that supports sophisticated reasoning about interface designs, in order to enable the creation of the design and run-time tools. The declarative representation will be translated into a partially compiled representation where many objects in the declarative model are translated into efficient procedures. However, references to the declarative model remain in the run-time representation to allow the use of sophisticated run-time tools when needed, without compromising performance.

The interface generation component (run-time system) and the modeling tools are currently being designed, and have not been implemented yet.

The rest of the paper is organized as follows. Next we briefly describe the MASTERMIND architecture in order to give some context about the role that models play in the whole system. The next sections discuss the modeling language in detail. We devote sections to discuss the models of application capabilities, tasks and presentation. We close with related work, current status and conclusions. An appendix contains an example model for an electronic mail application.

2 ARCHITECTURE

MASTERMIND uses different architectures for the design-time environment and for the delivery of applications. The design-time architecture is designed to support fast iterative development, and powerful design-time tools. To do so, the design-time architecture preserves the model in its declarative form, and maintains extensive book-keeping information so that when the models are extended, the interfaces generated from the model can be incrementally updated. The application delivery architecture is optimized for performance. It uses a compiled representation of the model that is smaller than the declarative representation, and supports fast generation of interfaces.

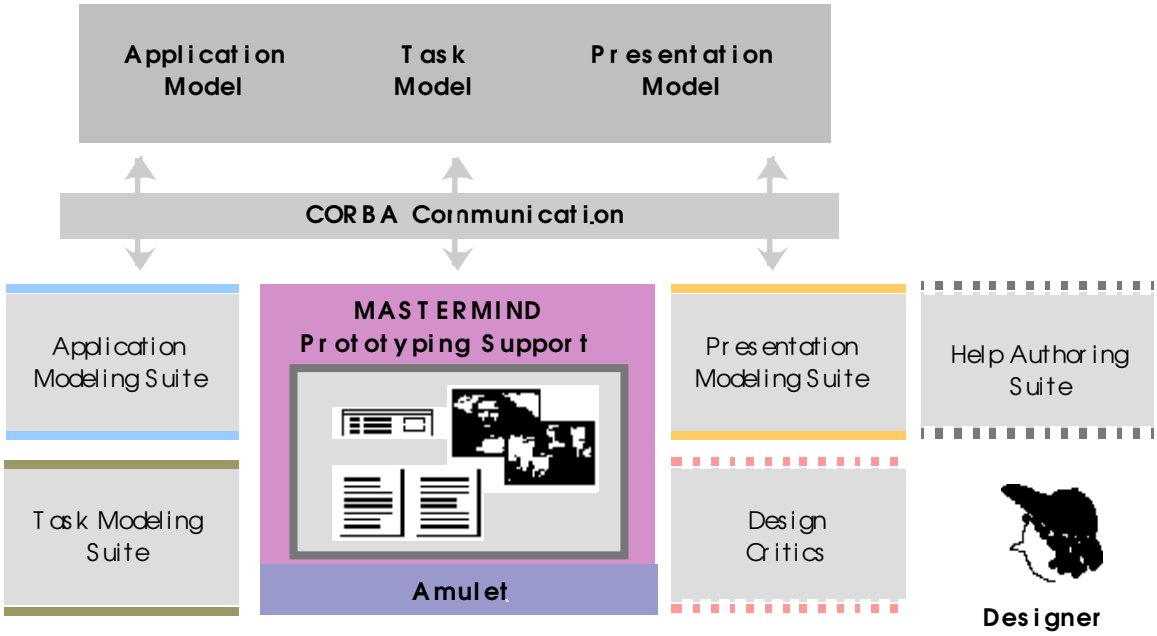

Figure 1. Architecture of the MASTERMIND design-time environment.

Figure 1 shows the architecture of the design-time environment. The MASTERMIND models are represented using the CORBA (Common Object Request Broker Architecture) object model, and run in a separate process called a *model server*. All the MASTERMIND tools and the MASTERMIND prototyping environment also run as separate processes, and can access and modify the model by communicating with the model server using the CORBA communication support layer[1]. This multi-process architecture allows new tools to be integrated without needing to modify or recompile the complete system, supports teams of designers working simultaneously, and supports remote collaborations too.

The model server uses a remote procedure call architecture to communicate with its clients. It provides procedures to create the large variety of structures that compose a model, procedures to modify and destroy these structures, and procedures to query the contents of the model. The model server can also save and restore models to and from text files. The complete set of procedures is specified using CORBA IDL.

MASTERMIND will provide tools for authoring the model (application, task and presentation modeling tools), tools for critiquing designs, tools for authoring the help systems, tools for generating portions of the interface automatically (e.g., generating menus from the task model) and a prototyping environment that can generate executable interfaces from the model, even before the model is completely specified. The tools are integrated by sharing the model via the model server. Whenever a model element in the model server is modified (by request of any tool), all tools that depend on the modified element are informed so that they can update their state. In particular, the prototyping environment is always notified about model changes so that it can update the interface prototypes "on the fly" whenever the design specifications change.

The application delivery environment does not use CORBA (unless the application itself uses CORBA). For delivery, the models will be translated into C++ source code that can be compiled and linked in with the rest of the application code, making delivered applications more compact and efficient. Delivered applications will retain the ability to contact the model server in order to access the declarative representations of the models

[1] Models can be partitioned into parts that execute in separate model servers running in separate nodes in a network. This makes it possible for teams of designers distributed in remote sites to collaborate in an efficient manner.

they use. This will allow tools like the animated help system, that are invoked during application execution, to access and analyze the models in an incremental fashion.

3 APPLICATION MODEL

The application model defines the capabilities of the application. MASTERMIND's application model is an extension of the CORBA object model. MASTERMIND uses CORBA because it is a widely emerging standard, whose modeling language provides the basic facilities needed to model applications. CORBA's strength is its support for distributed heterogeneous applications, so by using CORBA, MASTERMIND will be able to support interfaces for distributed applications. This section briefly summarizes the CORBA object model, and the MASTERMIND extensions.

The CORBA object model is very similar to C++ and Smalltalk classes. CORBA supports the definition of classes (called *interfaces* in CORBA) using multiple inheritance. Classes can have attributes and methods. In addition, CORBA has a model of exceptions. A class can declare a set of exceptions, which consist of a name and parameters describing data that will be communicated with the exception. The model of methods lists the exceptions that methods can raise.

MASTERMIND currently supports two extensions to the CORBA IDL language. The first one is the notion of method preconditions that allow developers to model when it is legal to call methods. The second extension is the notion of reports: other objects, including presentations and tasks, can register themselves as consumers of reports, to be informed when certain changes occur in an object, and update their state accordingly. The reports mechanism works even when the objects and the consumers are in different processes.

The example below shows a partial MASTERMIND model for an Email application. The bold keywords represent CORBA IDL modeling constructs. The bold, underlined keywords represent the MASTERMIND extensions. The Message object provides attributes that represent the information typically stored in an Email message. We show examples of two exceptions, and only three methods. The precondition of the send method specifies that send can only be invoked if the message_ready_to_send returns true. We show some of the reports that messages can generate in order to inform clients about changes.

```
interface Message {
    attribute Address sender;
    attribute Date arrival_time;
    attribute String subject;
    attribute sequence<Address> recipients
        raises recipient_incorrect;
    attribute String body;

    exception recipient_incorrect (String recipient);
      This exception is raised by the methods that modify the recipients attribute of a
      message.
    exception undeliverable_message;
      This exception is raised when a message cannot be delivered for whatever reason.

    boolean message_ready_to_send ();
    void send ()
        preconditions message_ready_to_send;
    void refile (in Folder where);
```

```
    report sender_changed;
    report subject_changed;
    report recipient_added (Address new_recipient);
    report recipient_removed (Address old_recipient);
    report recipients_changed;
}
```

The application model is not restricted to only contain objects representing the data structures of the application. Interface designers can model new objects that combine attributes from other objects, in order to better model the end user's view of the data. For example, in an application to allow users to dial the phone from the computer the interface designer might want to define new objects corresponding to countries and cities, even though this information is only implicitly represented in the data structures of the application (as prefixes to the number to be dialed).

4 EXPRESSIONS

MASTERMIND features an expression language to represent connections that tie the pieces of the model together. Examples of such expressions are assignment of parameters of model objects, invocation of application routines, predicates that test that certain conditions are true, arithmetic expressions, if-then-else and iteration expressions, and other programming language constructs.

A key feature of the expressions is that they are constraint like. When MASTERMIND evaluates an expression, it records the model elements on which the expression depends so that if these elements change later on, the expressions are automatically recomputed. For example, task preconditions are specified as an expression that tests that the value of certain task parameters satisfy some condition. Should the values of the parameters change, the precondition of the task is automatically brought up to date.

The expression language is suitably restricted so that MASTERMIND can analyze their behavior. For example, MASTERMIND can find out which tasks set the parameters that are used in the precondition of another task to determine which tasks need to be executed in order to make the precondition valid. For this reason, the expressions are represented internally in the model as objects with attributes rather than as textual scripts.

5 TASK MODEL

The task model describes the tasks that users can perform with a system. Task modeling in MASTERMIND centers around representing and elaborating user tasks by outlining the steps required to perform these tasks. Designers specify task hierarchies to tell MASTERMIND what users can do in an application, how the interface changes when users interact with the system, and what the underlying application does to provide users with needed or requested information to carry out intended tasks.

For each task, designers specify the goal of the task, the conditions in which the task can be performed, the effects of the task, information requirements of the task, and the breakdown into sub-tasks that specify how the task must be performed. The breakdown of a task is defined as a combination of user tasks, interface tasks, and application tasks. Designers can specify steps which are optional or steps which are only needed when certain conditions are true.

The lowest level of user tasks are interaction techniques, which correspond to primitives such as clicking on a button, or selecting an item from a menu. By putting

interaction techniques as part of a task breakdown, designers tell MASTERMIND what kind of inputs are expected from the user. By placing a system task designers tell MASTERMIND how it should update the interface. And by placing an application task, designers tell MASTERMIND that an application routine must be called at this point in the breakdown (when the user actually performs the task) to provide the task with information relevant to the interface or information required in the following steps.

Tasks are modeled in terms of two main objects, *Tasks* and *Task_Connections*. These objects and other auxiliary objects are described in the following sections.

5.1 Task

Tasks are modeled in terms of objects called *tasks*. The goal and effects attributes specify what the task does, the parameters specify the data on which the task operates, the preconditions specify when it is legal to execute the task, and the sub-tasks specify the steps for carrying out the task. MASTERMIND represents the sub-tasks with a task connection object that also specifies which of the sub-tasks must be executed and in what order. In addition, the task objects contain a set of flags that control dialogue sequencing.

Table 1 Attributes of Task.

Attribute Name	**Type**
name	*Symbol*
Prototype	*Task*
task_type	USER, PRESENTATION, APPLICATION, INTERACTION_TECHNIQUE, UNDETERMINED
goal	*Goal*
effects	*Expression, ...*
parameters	*name: Task_Parameter {...}, ...*
precondition	*Expression*
subtasks	*Task_Connection*
is_optional	*boolean*
is_resumable	*boolean*
is_interruptable	*boolean*
is_loop	*boolean*
is_reentrant	*boolean*

A task has a name, and a prototype task from which the newly defined task inherits information. For example, a task to print an Email message would be defined as a specialization of the generic Print task.

The task_type specifies the different categories of tasks. *User* tasks are tasks that the user performs, *Presentation* tasks are requests to present information to the user, *Application* tasks are tasks that the application performs without user involvement, *Interaction_Technique* tasks represent low-level tasks such as mouse clicks, and *Undetermined* tasks are a way for the developer to delay committing to a specific task type.

The effects are a specification of the actions to be performed when the task is executed. Typical effects are to invoke application routines, to present information, to change the status of other tasks, and to set task parameters. The effects of a task serve a dual purpose. They describe what the task does in a declarative way that can be analyzed by the various design and run-time tools. In addition, they are translated into executable code that makes the appropriate behavior happen at run-time.

The following is a list of the primitive expressions that can be used in the specification of the effects of tasks:

Method invocation

This expression is used to invoke a method on a specified object with a given set of arguments. The object and the arguments are specified by listing the task parameters that contain the values on which the method should operate.

Parameter setting

This expression is used to set the value of a parameter to the result of evaluating an arbitrary expression.

Task status modification

This expression is used to start, interrupt, abort and execute tasks.

Data presentation

This expression is used to present information to the user. The data is specified as a list of the task parameters that contain the information to be presented. Since expressions behave as constraints, when the task parameters change at run-time, the presentation is automatically updated.

The parameters are variables to store the data that the task operates on (details below).

The precondition specifies the conditions that must be true before the task can be executed. Preconditions are specified using expressions that test whether the values of task parameters satisfy some condition, or whether another task is in a given state.

Tasks provide a set of flags to control dialogue sequencing in a convenient way, without the use of preconditions. is_optional specifies whether the task is optional, and does not need to be performed; is_resumable specifies whether the task can be resumed after it is interrupted; is_interruptable specifies whether the task can be interrupted once it is started; is_loop specifies that the task can be performed multiple times, provided that the preconditions remain true, and is_reentrant specifies whether separate instances of this task can be spawned at run-time. In many applications users can spawn multiple instances of the same task, e.g., they can spawn a task to compose a message, and before finishing it, they can spawn a separate instance to compose a different message. In this case the task is said to be reentrant. If the task is not reentrant, only one instance of the task is used.

For *Interaction_Technique* tasks it is necessary to specify fields that depend on the particular interaction technique. For example, for the mouse click interaction technique it is necessary to specify which mouse button triggers the interaction technique, the area of the screen that can be clicked to invoke it, what happens if the user moves the mouse out of this area before releasing the mouse button, etc. These details are not discussed in this paper.

5.2 Goal

A goal is a specification of *what* a task does, in contrast to the effects that specify *how* a task accomplishes a goal. Goals can be represented either as text or as formal objects that MASTERMIND can analyze and operate with.

When a goal is specified as text, it just serves as documentation for what the task does, and can be shown to the user in help strings.

When a goal is represented formally as an expression, MASTERMIND can evaluate whether the goal is satisfied in any given context, and so can determine if the task needs to be executed. For example, if the goal of a task is to select an Email message, and a message is already selected, there is no need to force the user to execute the task. In this case, the task goal would be modeled using an expression that checks whether the task parameter that holds the current selections has a value of type Email message. In general, only tasks whose necessity is context dependent need to have their goal modeled formally.

5.3 Task_Parameter

Task parameters are variables defined in task objects for storing information needed for executing the task. Typical uses of parameters are to store data that needs to be passed to application routines, data that is needed to evaluate preconditions, and data that needs to be passed to other tasks.

The values of the parameters of root tasks (tasks without a parent) are set by MASTERMIND when the application starts. The values of parameters of subtasks can be defined either with an expression that computes the value based on parameters from other tasks, or can be explicitly set by the effects of tasks. Also, task parameters defined in a task are visible in the sub-tasks and all its descendants.

Table 2 Attributes of Task_Parameter.

Attribute Name	Type
name	*Symbol*
type	*type*
value	*Expression*
default	*Expression*
mode	PRODUCED, CONSUMED

The type of a parameter can be any standard C++ primitive types, or any type defined in the application model. The values of parameters are typically defined by expressions that compute a value in terms of the values of other parameters. Literal expressions are also allowed to support the specification of constants such as numbers and strings. Parameters can also have a default value, which is used when the expression for the value cannot compute a value. The mode specifies whether the task produces or consumes the value of the parameter.

5.4 Task_Connection

Task_Connections specify the sub-tasks of a task, and also specify which sub-tasks need to be executed and in what order. At any given moment MASTERMIND determines which tasks can or need to be executed next based on the information in the *Task_Connection*, and based on the information contained in the preconditions. For example, even though a task connection might specify that the sub-tasks can be done in *Unrestricted* order, the preconditions are taken into account too, and subtle, context-dependent sequencing restrictions get enforced.

Table 3 Attributes of Task_Connection.

Attribute Name	Type
tasks	*name: Task {...}, ...*
connection_type	SEQUENCE, PARALLEL, UNRESTRICTED, ONE_OF

The attribute tasks contains the list of sub-tasks of a task. Each sub-task can be named for easy reference from other sub-tasks. The connection_type specifies the order in which the subtasks of a task should be performed. *Sequence* specifies that the sub-tasks must be performed in sequence. *Parallel* specifies that the tasks can be executed in parallel, i.e., there is no need to wait for a task to complete before starting the next one. *Unrestricted* specifies that the sub-tasks can be executed in any order, except for ordering restrictions imposed by the preconditions, and *One_Of* specifies that only one of the sub-tasks needs to be executed.

6 PRESENTATION MODEL

The presentation model defines the visual appearance of the interface. Each display that an application can produce is defined by an object called a *Presentation*.

6.1 Presentation

This section gives an overview of the main attributes of a *Presentation*. The sections below describe the individual attributes in more detail. The appendix contains an example of the presentation definition for an Email application.

Table 4 Attributes of Presentation.

Attribute Name	Type
name	*Symbol*
Prototype	*Presentation*
parts	*name* : *Presentation* {...},...
parameters	*name* : *Presentation_Parameter* {...},...
guides	*name* : *Guide* {...},...
magnitudes	*name* : *Magnitude* {...},...
replication	*Replication* {...}, ...
grids	*name* : *Grid* {...}, ...
conditionals	*Conditional* {...}, ...

A presentation has a name, and a prototype presentation from which the newly defined presentation inherits information. For example, a dialogue-box would be defined as a specialization of the generic Dialogue-Box presentation.

Presentations are typically built up from smaller parts, which are also presentations. For example, the presentation definition for an Email program will define a part for command buttons, a part to show the headers of the messages in the user's mailbox, and a part to show the body of the selected message.

Presentations can have both standard parameters such as font and color, and application-specific parameters, such as the set of Email messages to be shown in a window. The guides are used to define layouts. For example, to specify that a collection of buttons should be placed in a row, a guide is defined, and the baselines of the buttons are snapped to the guide. The magnitudes are used to specify properties of the presentation such as the width and the height.

When a presentation is used as a part in another presentation, it can have a replication attribute to indicate that multiple copies of this part should be generated. For example, in an Email application, the presentation that displays the message headers is a replicated part, that gets replicated once for each message in the mailbox.

The grids support the definition of layouts, especially for defining the layout of replicated parts. Replicated parts can be attached to grids to align them with other parts, and to specify how the multiple replications should be laid out.

The conditionals support the specification of alternative presentations depending on characteristics of the data being displayed, or on characteristics of the display, such as the amount of space available. Conditionals can override any element of the specification where they appear (e.g., the position of grids, the prototype for a part), and can also add and remove parts.

6.2 Presentation_Parameters and Magnitudes

Presentation_Parameters are variables defined in presentation objects for storing values that control the appearance of the presentation. For example, a button presentation object

has parameters to specify the label of the button, the font for the label, whether the button is pressed or not, the color, etc.

MASTERMIND also supports application-specific parameters. Developers can add new parameters to presentation objects to keep track of whatever information is appropriate. For example, the presentation object for our Email application shown in appendix 1 defines a parameter called mailbox to store a pointer to the mailbox object that holds all the user's Email messages. In many cases the values of application-specific parameters are passed down to the parts of presentations, which extract from it information that is directly presented, or that is further passed down to other parts. As shown in appendix 1, the value of current folder is passed down to the headers, which extract from the current folder the individual messages. Then each message is further decomposed into date, sender and subject, and the pieces are passed as the values of label presentation.

As shown in the Table 5, presentation parameters are identical to task parameters, except that they don't have a mode attribute. Presentation parameters cannot produce values.

Table 5 Attributes of Presentation_Parameter.

Attribute Name	Type
name	*Symbol*
type	*type*
value	*Expression*
default	*Expression*

Magnitudes are special kinds of parameters used to store sizes of presentation components. In addition to the value, magnitudes support the specification of stretchability, and minimum and maximum values, which are useful in the specification of layouts.

Table 6 Attributes of Magnitudes.

Attribute Name	Type
is_a	*Presentation_Parameter*
name	*Symbol*
stretchability	*float*
min_value	*float*
max_value	*float*

6.3 Grids and Guides

Grids and guides are the basic facilities for defining layouts. Grids and guides have been used for many years by graphic designers to specify the layouts of pages in books, newsletters and advertisements because they provide a powerful framework for defining pleasing layouts. Grid theory has been adapted to electronic documents and interfaces and is part of many current user interface guidelines. For these reasons, MASTERMIND incorporates the notions of grids and guides into the presentation model. In addition, MASTERMIND supports a constraint language to allow the specification of layouts where the regularity that grids and guides enforce is not appropriate.

Grids and guides represent a departure from the layout mechanisms found in many interface builders and tool-kits, which organize layouts using rows and columns (hboxes and vboxes). The main problem with rows and columns is that they do not provide a way

to align elements of a display that are far apart, leading to layouts that are not visually appealing. Grids and guides solve this problem.

6.3.1 Grid

A *grid* is a set of parallel lines that span an area of the display. Parts of a presentation can be snapped to grid lines to define the layout. Grid lines in a parent presentation are visible in the children of the presentation and their children and so on. This allows deeply nested parts to be aligned with less nested parts.

Table 7 Attributes of Grids.

Attribute Name	Type
name	*Symbol*
direction	HORIZONTAL, VERTICAL
start	*Expression<integer>*
end	*Expression<integer>*
distance	*Expression<integer>*
num_lines	*Expression<integer>*
is_stretchable	*Expression <boolean>*
exceptions	*list<index, distance>*

The direction attribute specifies the direction of the lines in the grid. Most displays will contain at least one vertical and one horizontal grid. The start attribute specifies the coordinate where the first line of the grid should be placed. This coordinate can be a constant, or an integer expression that depends on a guide or parameter of a presentation. Typically, the expression is just a reference to a guide, which causes the grid to start at the location of the guide. The end attribute specifies the coordinate where the grid ends. The distance and num_lines attributes specify the distance between the lines of the grid, and the the number of lines in the grid. The is_stretchable attribute specifies how the grid should be adjusted when the start or end of the grid change. If the grid is stretchable, when the start or end change, the distance between the grid lines is proportionally adjusted, and the number of lines remains fixed. If the grid is not stretchable, the number of lines is adjusted.

The start, end, distance, num_lines and is_stretchable attributes cannot all be specified. For example, if start, end, distance and is_stretchable are specified, MASTERMIND automatically derives the value for num_lines.

The exceptions allow the definition of irregular grids, where the distance between some grid lines is different from the default. This is useful for applications such as spread-sheets, where the user can manually adjust the size of columns.

6.3.2 Guide

A *guide* is a single line to which parts can be snapped. By default, all presentations have guides corresponding to their left, right, top and bottom. In addition, developers can define other guides as appropriate (e.g., a guide one third of the distance from the left). Layouts are defined by snapping guides to other guides or to grids.

Table 8 Attributes of Guides.

Attribute Name	Type
name	*Symbol*
direction	HORIZONTAL, VERTICAL
position	*Expression<integer>*
margin_1	*Expression<integer>*

margin_2	*Expression<integer>*

The direction attribute specifies the direction of the line. Its position can be a constant or an expression that depends on other guides and grids. Guides can have margins which are lines on both sides of the guide, parallel to it, and offset a certain amount. They make it easy to define the gaps between parts of a presentation. For example, when defining a row of buttons, consecutive buttons can be snapped to the margin_1 and margin_2 of the guides, rather than to the position, to define a gap between them.

Grids and guides are design time elements, i.e., they are used by developers to define the layout of the display, but they don't appear in the final interface that the end-user sees. However, it is possible to define tasks that manipulate the guide position, thus giving the end-user a way to adjust the layout of the display.

6.4 Replications

Most applications manipulate collections of information, for example, a collection of messages in a mailbox, a collection of notes in a music score, collections of nodes and lines in a graph editor. To define the presentations for these applications it is necessary to specify that some part of the presentation should be replicated as many times as there are elements in the collections to be displayed. The MASTERMIND replication construct is the mechanism to model presentations of collections. When combined with the conditional construct, replications can be used to specify the presentation of heterogeneous collections of information (e.g., Email and voice messages in a messaging application).

Table 9 Attributes of Replications.

Attribute Name	**Type**
name	*Symbol*
replication_data	*Expression<Set>*
is_on_demand	*boolean*
references	*Reference* {...}, ...
anchor	*Presentation* {...}
generic	*Presentation* {...}

The replication_data is an expression that computes the set of elements to be displayed. Typically this is an expression that invokes an application routine (e.g., a routine that extracts from a mailbox object the collection of messages it contains).

The is_on_demand attribute specifies whether all replicas of the part should be generated in advance, or only as needed. This attribute gives developers the flexibility to define presentations where all elements of a collection are displayed at once (e.g., in a scrollable window), or presentations where the display of the elements is constructed incrementally (e.g., a display composed of multiple pages, where pages are added only as needed).

The references specify how to lay out the multiple replicas of the part using grids or using other parts as a reference (see description of *Reference* object).

The anchor and generic provide an alternative to references for specifying the layout of replicated parts. This mechanism is used for layouts that cannot be defined appropriately in terms of grids or other parts. The basic idea is to define the layout by defining the position of the first element (anchor), and then defining the position of the nth element (generic) based in the position of the nth minus one element (generic).The anchor is a presentation skeleton that defines how the first replica should be laid out. Typically the skeleton presentation only contains definitions for two guides (e.g., left and top), in terms of guides and grids of the parents. The generic is a list of presentation skeletons that

defines how all the other elements are laid out. Typically two generic presentations are defined, corresponding to the nth (n) and nth minus one (n-1) replicas. Expressions for guides in the nth presentation define the layout with respect to the guides for the n-1 presentation. More than two generic presentations can be defined to define layouts that depend on the n-1, n-2, etc. replications.

Table 10 Attributes of References.

Attribute Name	Type
name	*Symbol*
reference	*Grid | Replication*
init_index	*integer*
end_condition	*Expression<boolean>*
init_procedure	*Expression<grid_index | part_index>*

References provide a convenient way for specifying the layout for replicated parts by specifying that the replications should fill a grid, or by specifying that the replications should be laid out according to some other replication which is already laid out (e.g., specifying that an icon should be placed to the right of every other element of a list).

It is possible to specify multiple references in a replication object. A common case is to use two references, containing a vertical and a horizontal grid. The effect is that elements are laid out in rows according to the vertical grid. When the columns in the vertical grid are exhausted, the next row in the horizontal grid is used. In general, the use of multiple references supports the specification of a large variety of layouts.

The reference attribute specifies the grid or replication on which the layout is based. When the layout is based on another replication, it means that each part will be laid out with respect to each part of the other replication (e.g., to the right of it).

The init_index specifies which grid line or replication element should be used to place the first element in this reference, which by default is the first one. The init_procedure is an advanced feature that provides more flexibility for specifying the init_index by allowing a procedure to be called, rather than always using the same init_index. The end_condition specifies when to stop using this reference for placing objects (e.g., when reaching the last column in a vertical grid).

6.5 Conditionals

Conditionals allow the definition of presentations whose appearance depends on the characteristics of the data to be displayed (e.g., Email messages and voice messages in a messaging application), on the characteristics of the platform (e.g., color display vs. black and white), or on the characteristics of the interface at any given moment (e.g., given a row of buttons, if the window is narrowed, rather than truncating the row, some buttons become pull-down menus, so that all commands remain accessible without scrolling).

A template can have several conditionals, which is a case statement, consisting of several Case_Clauses. MASTERMIND evaluates each case statement independently: it evaluates the predicate of each clause in the sequence until one returns true, and then uses the specification associated with the successful clause.

Table 11 Attributes of Case_Clause.

Attribute Name	Type
predicate	*Expression<boolean>*
specification	*Presentation* {...}

The predicate is an expression that depends on presentation or task parameters, guides and grids of the containing presentation and its ancestors. When it is true, the features of the *Presentation* specified in the specification attribute are applied to the presentation being constructed. For example, the *Presentation* might specify values for only a few parameters or guides, and perhaps override the prototype presentation for a part.

When MASTERMIND evaluates conditionals, it records the dependencies of all the predicates it evaluates, so that if any of the parameters, guides or grids on which the predicates depend changes value, MASTERMIND will re-evaluate the appropriate predicates, re-apply the appropriate specifications, and update the display.

7 RELATED WORK

The tools for constructing user interfaces can be divided into three basic categories: interface builders, UIMSs and model-based tools. The following sections compare MASTERMIND to the tools in these categories.

7.1 Interface Builders

Interface builders [25, 26] are the most popular interface construction tools in the market. They provide an easy to use, WYSIWYG interface for constructing interfaces where designers can draw the screens of the interface. Interface builders use a very low level description of the interface consisting mainly of the location, properties and type of all the elements of the display. This is not enough information to support sophisticated analytic tools for design evaluation and critiquing, tools for retargetting the interface to a new platform or tools for automatically generating help.

In contrast, the model-based tools, and MASTERMIND in particular contain a rich model of the interface that supports the above-mentioned tools. Unlike other model-based tools [2, 10, 33, 34, 35, 36], MASTERMIND is designed to provide an easy to use interface for building the models, similar to the designer's interface that interface builders provide. The appendix shows a mockup of MASTERMIND's presentation editor: it is similar to an interface builder, but captures a much richer representation of the presentation. In addition, MASTERMIND can be used to specify all aspects of the interface, including the dynamic aspects that interface builders do not support.

7.2 UIMSs

UIMSs [30] are user interface construction tools used mainly to construct the dialogue component of the interface. They use specialized languages to describe the dialogue (transition networks, grammars or events). Some UIMSs also provide some facilities for specifying presentations, but they are very primitive. MASTERMIND's dialogue specification is more comprehensive. It captures not only the high level user tasks that end users are expected to perform with the system, but also their decomposition into low level tasks that correspond to dialogue specifications used in UIMSs.

Some UIMSs [1, 27] automatically generate interfaces from specifications that are similar to the MASTERMIND application model. MASTERMIND improves on these systems by providing much more comprehensive models of tasks and presentations, allowing designers much better control over the interfaces generated.

7.3 Model-Based Tools

In this section we compare MASTERMIND to other model-based tools based on the features and capabilities enabled by the task and presentation models.

7.3.1 Task Models

Task analysis is a very important part of the user interface design process leading to interface designs which better meet user requirements. In the past, a number of researchers have developed task representations which capture user task information in a processable form. Most of these representations capture the hierarchical nature of task decomposition in the same fashion as MASTERMIND.

The MASTERMIND task models are similar to GOMS [16, 20], TAKD [7] and UAN [12], but unlike GOMS, TAKD and UAN, the MASTERMIND models are also used to drive the interfaces at run-time, rather than just describing them. GOMS's task representation is useful for evaluating interfaces by predicting speed of use based on specified task decompositions. The representation emphasizes recording paths of user actions including both motor steps and mental steps in performing tasks. GOMS's representation is not intended to capture situations where multiple actions could be performed by the users in a concise fashion. The MASTERMIND task model was designed to be a superset of the GOMS representation, and a translator will be built that generates GOMS models from the MASTERMIND representation so that GOMS techniques can be used to evaluate interface designs represented in MASTERMIND. Task Analysis for Knowledge Description (TAKD) is a taxonomy developed to systematically capture objects and their functions. By applying TAKD, objects and functions in a domain can be precisely specified and classified. Unlike MASTERMIND, TAKD helps designers conceptualize tasks and objects. MASTERMIND only facilitates creating objects and task descriptions and leaves the responsibility for doing so to the designers. UAN uses task descriptions and hierarchical task decomposition as a way to specify links from the task model to the interface specification. This helps make sure all user requirements are fulfilled and all functions have accessing mechanisms for users. UAN's task representation is rather extensive including specification for time, parallelism, and task interruption.

Recently, several systems are using task models to drive the interfaces at run-time (ADEPT [17, 18] and TRIDENT [2, 35]). ADEPT automatically generates user interfaces from its task descriptions; however, its limitation lies in the environment which does not allow interface styles to be added or allow designers enough control over interface details. TRIDENT uses task descriptions to help in automatic generation of user interfaces. TRIDENT does a very good job in generating high quality interfaces because it makes use of a machine analyzable representation of interface guidelines. However, unlike MASTERMIND, TRIDENT only addresses the generation of form-based interfaces.

Like UIDE, MASTERMIND makes uses of pre and post-conditions (called preconditions and effects in MASTERMIND) to model the dialogue component of the interface. MASTERMIND improves on UIDE in that it uses an efficient constraint system to evaluate and maintain the preconditions, thus allowing MASTERMIND to scale up to large applications.

7.3.2 Presentation Models

MASTERMIND's presentation model is similar to HUMANOID'S [33, 34], and to ITS's [36]. MASTERMIND's main contribution is that its presentation model is designed to support graphical specification of presentations similar to that of interface builders. In addition, MASTERMIND supports notions of visual graphic design (grid design) [38] not supported in these systems.

MASTERMIND's presentation model is similar to Garnet [24] in that both make heavy use of constraints to define layouts. MASTERMIND provides less control over interface details than Garnet does, but provides higher level components that automatically update the display when the data being presented changes, and so are easier to use.

8 CURRENT STATUS AND FUTURE WORK

The specification of the MASTERMIND modeling language as described in this paper is complete. The modeling language is specified in a CORBA callable frame-based system. In addition, we developed a grammar to for the textual specification of the models, which is used to store models in files. We will use the textual specification to bootstrap the interactive modeling tools, which will become the standard way for specifying models. We do not expect developers to use the textual modeling language.

The prototyping environment is being designed, and implementation started in the Spring of 1995. Some of the interactive tools have been partially designed too, and their implementation will commence once the prototyping environment is ready. We expect to have an initial version of the system complete by the end of the Fall of 1995.

This paper reports work in progress, and we expect to need to modify the modeling language as we gain more experience using MASTERMIND to build interfaces. The following are a set of issues that we have not yet resolved satisfactorily, and which will have an impact on the modeling language, the prototyping environment and the generated interfaces.

- Expressiveness. No matter how hard we try to make a declarative modeling language expressive, there will be interface designs that cannot be modeled. Our approach to deal with this problem is to allow interface developers to use "foreign" components, such as OLE custom controls. These foreign components will be minimally modeled, by specifying the parameters and methods that can be called on them. MASTERMIND will not be able to reason with them fully, but at least will allow them to be included in the designs.
- Extensibility. The MASTERMIND modeling language is extensible in the sense that new attributes and objects can be added to the modeling language. The problem is that for the extensions to have any effect, the tools must be updated to take into account the new information. We envision two kinds of extensions: extensions to the MASTERMIND core, which will be done by the MASTERMIND developers, and tool-specific extensions that can be done by anyone who wants to incorporate a new tool into the MASTERMIND suite of tools. As the system evolves, some of the tool-specific extensions will be migrated into the core.
- Semantics. Currently there is no formal semantics for the MASTERMIND modeling language. The semantics of the frame-based system used for modeling are straight forward (inheritance, and part/whole hierarchies), but a lot of the semantics of the model is implicit in they way that the tools make use of the attributes being modeled. We expect that we or others will construct tools to check models for consistency, and various notions of quality, but we do not expect to develop a formal semantics for our model.
- Relationship to other task modeling schemes. The literature on task modeling is very rich, and many schemes for modeling tasks have been proposed. We have tried to arrive at a compromise that satisfies many conflicting goals: easy to specify, directly executable and expressive. We have given more weight to the first goals, and expect to enhance the modeling language to incorporate features of the more expressive notations.

9 CONCLUSIONS

MASTERMIND is a model-based interface development environment designed to address the shortcomings of existing model-based tools. The modeling language is designed to be as expressive as possible without making it hard for developers to model interfaces:

- The application modeling language is an extension of the CORBA IDL language. MASTERMIND allows applications to be specified in IDL, so they can use CORBA to support application embedding and network distribution (like OLE and OpenDoc). The application model complements the IDL with information needed to drive the user interface.
- In addition to the task, sub-task decomposition features of other task modeling systems, in MASTERMIND it is possible to specify detailed ordering constraints for tasks, optional and repeatable tasks, preconditions and effects. In addition the MASTERMIND specifications are described in a formal language that supports generation of the interface as well as analysis.
- The presentation modeling language features grids and guides to allow the specification of pleasing layouts, it features a constraint system to support the specification of complex layouts, and to support screen resize and update, and it features iteration and conditional constructs to support the specification of displays of dynamic information. Together, these features support the specification of the main windows of applications, not just the menus and dialogue boxes.

In addition to its expressivity, the MASTERMIND modeling language is implemented as separate process so that it can be used by an open set of tools, and by several designers working simultaneously.

10 ACKNOWLEDGMENTS

We wish to thank David Kieras for spending several days with us explaining how the GOMS models work, and helping us design a modeling language that is as expressive as GOMS, executable, as needed by the prototyping environment, and analyzable, as needed by the tools. We also want to thank the reviewers for their extensive and insightful comments.

11 APPENDIX

The appendix describes parts of the model for an Email application. The example is expressed in the syntax of the textual representation of MASTERMIND models.

Figure 2 shows the presentation model for an Email application. The parts of the template are shown as boxes labeled with the name of the part. The presentation editor allows parts, guides, grids, iterations and conditionals to be added, deleted and manipulated in a graphical way. Layout is defined by dragging parts until they snap to guides or grids. The region bounded by hguide 1 and hguide 2 contains a replication of a part to show the message headers (see Header_Template below). Each replication consists of three labels to show the date, sender and subject of a message.

Figure 3 shows the interface that MASTERMIND would generate using the model specified in the previous figure. The parts of all the presentation objects have been instantiated and bound to data from the application.

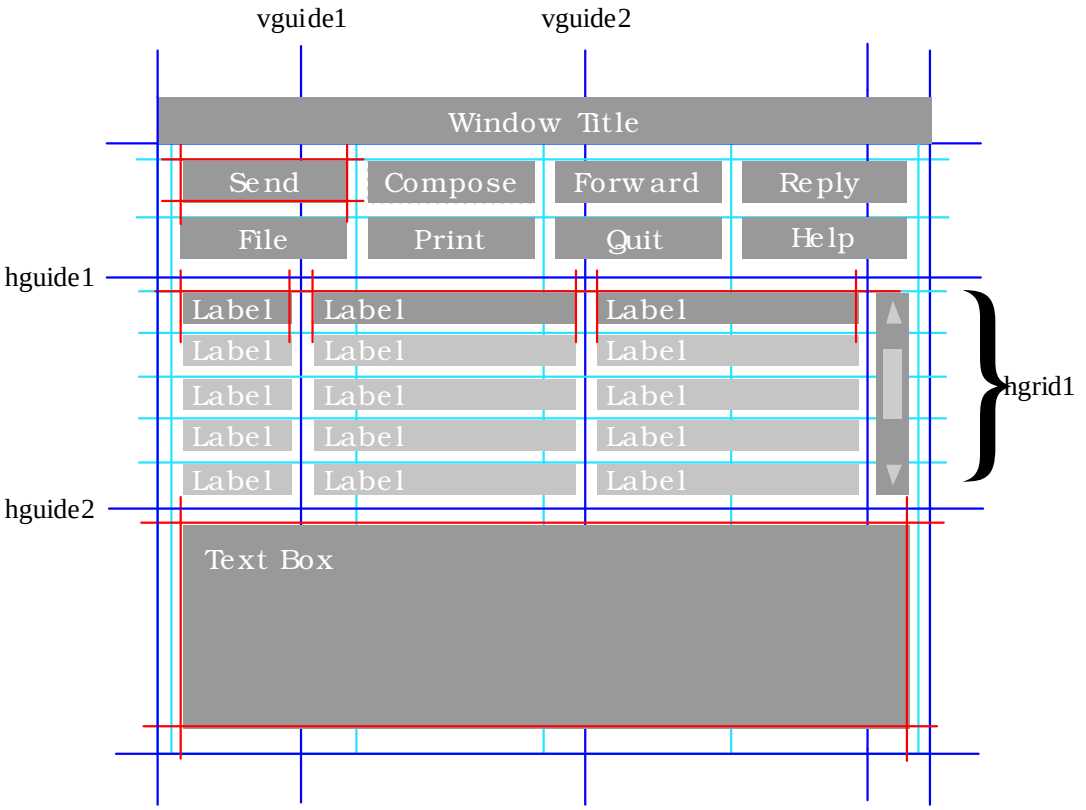

Figure 2 A mockup of the presentation modeling tool showing the model for the main display of an Email application.

Below is the textual specification that defines the model that builds the interface shown in Figure 3. Words formatted like Mail_Interface represent the names of objects being defined. Words formatted like *Parameter* represent the type of an object or a value of an enumeration, and expressions in square brackets (e.g., [(hguide1 + bottom) / 2]) represent *Expressions*.

The presentation object for the main window is called Mail_Interface. The parameter mailbox is bound to the mailbox parameter defined in the root task for the Email interface.

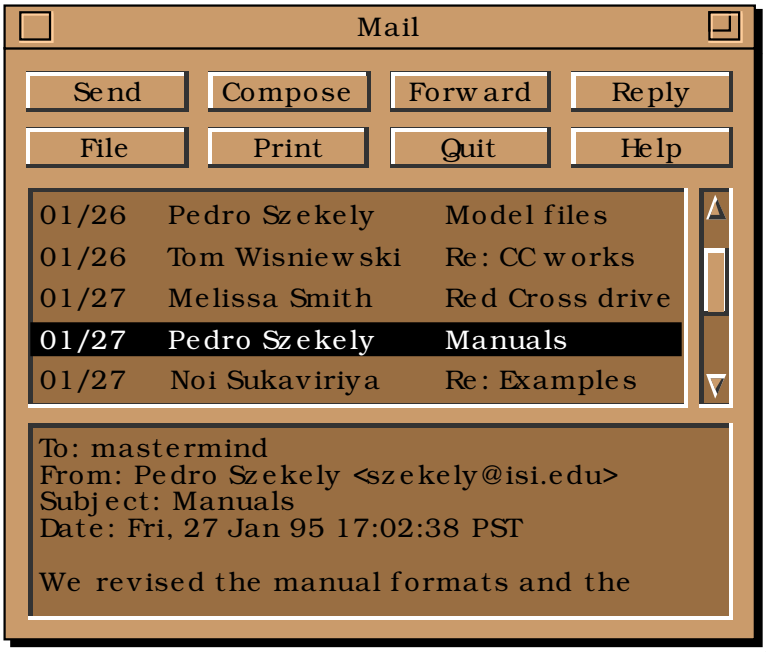

Figure 3. Screen shot of the interface that MASTERMIND would generate from the presentation model specified in Figure 2.

This task parameter is initialized when the application starts. The definitions of guides hguide2, vguide1 and vguide2 use expressions that depend on parameters of the window itself (bottom, left, etc.) so that the space is proportionally assigned when the window is resized. The definition of the gird hgrid1 depends on the font used for the window, so that the grid is adjusted according to the font being used. We only show the definition of one part of the window that displays the message headers (the other parts are defined similarly). The header part is replicated according to the contents of the mailbox, which are computed calling the contents method on the value of the mailbox parameter. The reference for the replication is hgrid1, so that the headers are displayed in a column.

```
Mail_Interface : Window {
  parameters =   mailbox : Parameter {
                                value = Email_Task.mailbox },
                 font : Parameter {value = Chicago12;};
  guides =       left : Guide {      // To leave a small space at the left of the window.
                   direction = VERTICAL; right_margin = 5;},
                 right : Guide {
                   direction = VERTICAL; left_margin = 5;},
                 hguide1 : Guide {   // Top for headers.
                   direction = HORIZONTAL; position = 200;},
                 hguide2 : Guide {   // Bottom for headers.
                   direction = HORIZONTAL; position = [(hguide1 + bottom) / 2];},
                 vguide1 : Guide {   // Position for message sender field.
                   direction = VERTICAL; position = [2/3 * left + 1/3 * right];
                   right_margin = 2; left_margin = 2;},
                 vguide2 : Guide {   // Position for subject field
                   direction = VERTICAL; position = [1/3 * left + 2/3 * right];
                   left_margin = 2; right_margin = 2;};
  grids =        hgrid1 : Grid {
                   direction = HORIZONTAL; start = [hguide1]; end = [hguide2];
                   stretchable = FALSE; distance = [font.height () + 2];};
  parts =        header : Header_Presentation {
                   replication =    {
                                      is_on_demand = FALSE;
                                      replication_data = [mailbox.contents ()];
                                      references =    grid_ref {
                                                        reference = [hgrid1];};
                                    };
                   guides =         top : Guide {
                                      direction = HORIZONTAL; position = [grid_ref];};
                 };
};
```

Header_Pres specifies how to display each individual header. The value of the message parameter each data element of the replication defined above. The presentation has three parts, for the date, sender and subject attributes of the message. The parts are aligned to the top guide defined in the replication in of the Mail_Interface presentation object. Note the use of consume_reports in the definition of the sender part. It declares that when the

message produces the sender_changed report, this part will be informed so that the display can be appropriately updated.

```
Header_Pres : Presentation {
  parameters =  message : Parameter
                  value = [Mail_Interface.header.replication.replication_data];};
  parts =       date : Label {
                  parameters =  text : Parameter {
                                  value = [message.date ()];};
                  guides =      top : Guide {
                                  direction = HORIZONTAL; position = [Header_Pres.top];},
                                left : Guide {
                                  direction = VERTICAL;
                                  position = [Mail_Interface.left.right_margin];},
                                right : Guide {
                                  direction = VERTICAL;
                                  position = [Mail_Interface.vguide1.left_margin];};
                },
                sender : Label {
                  parameters =  text : Parameter {
                                  value = [message.sender ()];
                                  consume_reports = sender_changed;};
                  guides =      top : Guide {
                                  direction = HORIZONTAL; position = [Header_Pres.top];},
                                left : Guide {
                                  direction = VERTICAL;
                                  position =  [Mail_Interface.vguide1.right_margin];},
                                right : Guide {
                                  direction = VERTICAL;
                                  position = [Mail_Interface.vguide2.left_margin];};
                },
                subject : Label {
                  parameters =  text : Parameter {
                                  value = [message.subject ()];};
                                  consume_reports = subject_changed;};
                  guides =      top : Guide {
                                  direction = HORIZONTAL; position = [Header_Pres.top];},
                                left : Guide {
                                  direction = HORIZONTAL;
                                  position =  [Mail_Interface.vguide2.right_margin];},
                                right : Guide {
                                  direction = HORIZONTAL;
                                  position = [Mail_Interface.right.left_margin];};
                };
};
```

The following is the task model for the task to forward a message. The message parameter specifies the message to be forwarded. Its value is specified as the effect of the Select_Message task, which is not shown here. In order to forward a message, the user has to complete the tasks specified in the subtasks attribute. These subtasks must be done in sequence. The precondition of the task specifies that the Forward_Message task cannot be invoked if a message is not selected.

```
Forward_Message : Task {
   goal = "To forward a received message to a different recipient.";
   task_type = User;
   parameters =   message : Parameter {
                    type = Message; mode = CONSUMED;},
                  recipient : Parameter {
                    type = String;};
   subtasks = :Task_Connection {
                  connection_type = SEQUENCE;
                  tasks = Invoke_Forward, Specify_Recipient, Fill_in_Message, Send;};
   preconditions = [selected_msg != NULL];
   is_reentrant = TRUE;
   is_interruptable = TRUE;
};
```

The Invoke_Forward task is a leaf task bound to the interaction technique bound to the Forward_Button presentation object (not shown here). In the interactive environment the developer would only have to define the goal of the task, because the task would have been automatically created when the button part was added to the Email_Window presentation. This task has no explicit effects. It serves to block the following tasks until the user clicks on the forward button.

```
Invoke_Forward : Task {
   goal = "Indicating that a message is to be forwarded.";
   task_type = Interaction_Technique;
   task_extension = :Technique_extension {
                        interactor =  :Am_Choice_Command {
                                        object = Forward_button;
                                        ...
                                      };
                      };
};
```

After the user clicks on the forward button, he has to specify to whom the message should be forwarded. The effect of the task is to set the recipient parameter string that the user types in (the contents of the text edit interaction technique).

```
Specify_Recipient : Task {
   goal = "Indicate who will receive the forwarded message.";
   task_type = Interaction_Technique;
   task_extension = :Technique_extension {
                        interactor = :Am_Text_Edit_Interactor {
                                        object = Forward_Address_Field .....
```

```
                                        };
                        };
  effects = [recipient <- Forward_Address_Field.contents];
  };
```

Once the recipient has been specified, the system will display the message being composed using the Display_Message task, and the user has to specify the body of the message in the Modify_Text task. These tasks are not described here.

```
Fill_in_message : Task {
  goal:  "To add to the forwarded message.";
  task_type = USER;
  subtasks = :Task_Connection {
                    connection_type = SEQUENCE;
                    tasks = Display_Message, Modify_Text;
                };
  };
```

Once the message is filled in, the user can send it by completing the Send task. This task involves two steps. The first one, Invoke_Send is an Interaction_Technique task where the user clicks on a button to request that the new message be sent. The second one, Call_Send is an Application task that invokes the application routine that sends the message.

```
Send : Task {
  goal = "Finishing off the message to be forwarded.";
  task_type = USER;
  subtasks = :Task_Connection {
                    connection_type = SEQUENCE; tasks = Invoke_Send, Call_Send;
                };
  };
```

12 REFERENCES

1. M. Beshers and S. Feiner. Scope: Automated Generation of Graphical Interface. In Proceedings of ACM SIGGRAPH 1989 Symposium on User Interface Software and Technology (UIST '89). pp.76-85.

2. F. Bodart, A. Hennebert, I. Provot, J. Leheureux, J. Vanderdonckt. A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype. In the Proceedings of the Eurographics Workshop on Design, Specification, and Verification of Interactive Systems. Bocca di Magra, Italy, June 8-10, 1994.

3. F. Bodart and J. Vanderdonckt. On the Problem of Selecting Interaction Objects, in Proceedings of HCI'94 "People and Computers IX" (Glasgow, 23-26 August 1994), G. Cockton, S.W. Draper, G.R.S. Weir (Eds.), Cambridge University Press, Cambridge, 1994, pp. 163-178.

4. R. Braudes, A Framework for Conceptual Consistency Verification, D.Sc. Dissertation, Dept. of EE&CS, The George Washington University, Washington, DC 20052, 1990.

5. M. D. Byrne, P. Sukaviriya, S. D. Wood, J. D. Foley and D. Kieras. Automating Interface Evaluation. In Proceedings of Human Factors in Computing Systems, CHI'94. Boston, April 1994.

6. D.J.M.J. de Baar, J.D. Foley, and K.E. Coupling Application Design and User Interface Design. In Proceedings of Human Factors in Computing Systems, CHI'92. Monterey, California, May 1992, pp. 259–266.

7. D. Diaper. Analysing Focused Interview Data with TASK Analysis for Knowledge Description (TAKD). In the Proceedings of IFIP INTERACT' 90: Human-Computer Interaction.

8. S.K. Feiner. APEX: An Experiment in the Automated Creation of Pictorial Explanations. IEEE Transactions on Computer Graphics and Applications, November 1985.

9. S.K. Feiner. and K. R. McKeown. Generating Coordinated Multimedia Explanations, Proceedings of the 6th IEEE Conference on Artificial Intelligence Applications, pp. 290-303, 1990.

10. J. Foley, W. Kim, S. Kovacevic and K. Murray, UIDE - An Intelligent User Interface Design Environment, in J. Sullivan and S. Tyler (eds.) Architectures forIntelligent User Interfaces: Elements and Prototypes, Addison-Wesley, Reading MA, 1991, pp.339-384.

11. D.F. Gieskens and J.D. Foley. Controlling User Interface Objects through Pre- and Postconditions. In Proceedings of Human Factors in Computing Systems, CHI'92. Monterey, California, May 1992, pp. 189–194.

12. R. Hartson, K. Mayo. A Framework for Precise, Reusable Task Abstractions. In the Proceedings of the Eurographics Workshop on Design, Specification, and Verification of Interactive Systems. Bocca di Magra, Italy, June 8-10, 1994.

13. P. J. Hayes. and P. Szekely. Graceful interaction through the COUSIN user interface, International Journal of Man-Machine Studies, vol. 19, pp. 285-305.

14. P. J. Hayes, P. Szekely and, R Lerner. Design Alternatives for User Interface Management Systems Based on Experience with COUSIN, in Proceedings of CHI'85 (San Francisco, 14-18 April 1985), Addison-Wesley, Reading, 1985, pp. 169-175.

15. P. J. Hayes. Executable Interface Definitions Using Form-Based Interface Abstractions, in Advances in Human-Computer Interaction, vol. 1, Hartson, R. (Ed.), Ablex Publishing Corp., Norwood, Chapter 6, pp. 161-189.

16. B.E. John and A.H. Vera. A GOMS Analysis of a Graphic, Machine-Paced, Highly Interactive Task. In Proceedings of Human Factors in Computing Systems, CHI'94. Monterey, California, May 1992, pp. 251-258.

17. P. Johnson, S. Wilson, P. Markopoulos, J. Pycock. ADEPT - Advanced Design Environment for Prototyping with Task Models. In the Proceedings of INTERCHI '93.

18. P. Johnson, S. Wilson and H. Johnson. Scenarios, Task Analysis And The Adept Design Environment. In J. Carroll (ed) Scenario based Design. Addison Wesley. (In Press).

19. P. Johnson, H. Johnson, and S. Wilson. Rapid Prototyping of User Interfaces Driven by Task Models, to appear in Scenario-Based Design, John M. Carroll (Ed.), John Wiley & Sons, 1995, pp. 209-246.

20. D. E. Kieras and P. G. Polson. An Approach to the Formal Analysis of User Complexity. International Journal of Man Machine Studies, 22, 365-394.

21. W. Kim and J. Foley, DON: User Interface Presentation Design Assistant, In Proceedings UIST'90. October 1990, pp. 10-20.

22. J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. ACM Transactions on Graphics, pp. 110-141, April 1986.

23. R. Moriyon, P. Szekely and R. Neches. Automatic Generation of Help from Interface Design Models. In Proceedings of Human Factors in Computing Systems, CHI'94. Boston, April 1994.

24. B. Myers, et. al. The Garnet Reference Manuals. Technical Report CMU-CS-90-117-R2, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. May 1992.

25. Neuron Data, Inc. 1991. Open Interface Toolkit. 146 University Ave. Palo Alto, CA 94301.

26. NeXTStep and the NeXT Interface Builder. NeXT, Inc. 900 Chesapeake Drive, Redwood City, CA 94063. 1991.

27. D. Olsen. MIKE: The Menu Interaction Kontrol Environment. ACM Transactions on Graphics, vol 17, no 3, pp. 43-50, 1986.

28. A. Puerta. The Study of Models of Intelligent Interfaces. In Proceedings of the ACM International Workshop on Intelligent User Interfaces. Jan, 1993. pp. 71-78.

29. A.R.Puerta, H. Eriksson, J.H. Gennari and M.A. Musen. Toward ontology-based frameworks for knowledge-acquisition tools. In Proceedings of the Eigth Knowledge-Acquisition Workshop for Knowledge-Based Systems. Banff, Alberta, Canada, February 1994.

30. G. Singh and M. Green. A High-level User Interface Management System. In Proceedings SIGCHI'89. April 1989, pp. 133-138.

31. P. Sukaviriya. Dynamic Construction of Animated Help from Application Context, Proceedings of ACM SIGGRAPH 1988 Symposium on User Interface Software and Technology (UIST '88), 1988, ACM, New York, NY, pp. 190-202.

32. P. Sukaviriya and J. Foley. Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help. In Proceedings of UIST '90. October 1990, pp. 142-146.

33. P. Szekely, P, Luo, and R. Neches. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In Proceedings SIGCHI'92. May 1992, pp. 507-515.

34. P. Szekely, P. Luo, and R. Neches. Beyond Interface Builders: Model-Based Interface Tools. In Proceedings of INTERCHI'93 April, 1993, pp. 383-390.

35. J. M. Vanderdonckt , F. Bodart. Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection. In INTERCHI'93 Proceedings, Amsterdam, Netherlands. April, 1993, pp. 424-429.

36. C. Wiecha, W. Bennett, S. Boies, J. Gould and S. Greene. ITS: A Tool For Rapidly Developing Interactive Applications. ACM Transactions on Information Systems 8(3), July 1990. pp. 204-236.

37. S. Wilson, P. Johnson, C. Kelly, J. Cunningham and P. Markopoulos. Beyond hacking: a model-based approach to user interface design, in Proceedings of the HCI'93 "People and Computer VIII", Cambridge, University Press, 1993, pp. 215-231.

38. R. Williams. The Non-Designer Design Book. Peachpit Press Inc., Berkeley, California, 1994.

13 BIOGRAPHIES

13.1 Pedro Szekely

Pedro Szekely is a research assistant professor at ISI concerned with the development of principled, general-purpose user interface management systems. He received his Ph.D. in Computer Science from Carnegie Mellon University in 1987 for research on user interface management systems, focusing on defining clear standards for the requirements of communication between application programs and a user interface management system. He was one of the designers and implementors of COUSIN, one of the first model-based user interface management systems. He also developed the initial version of the constraint-based graphics system for the Garnet project. At the ISI Dr. Szekely developed HUMANOID, a model-based user interface design environment, and is now principal investigator for the MASTERMIND project, an ARPA funded project in collaboration with Georgia Tech. MASTERMIND will produce a next generation model-based interface development environment by combining the best features of HUMANOID and Georgia Tech's UIDE system.

13.2 Piyawadee "Noi" Sukaviriya

Piyawadee Sukaviriya , is a Research Scientist II (equivalent of Research Assistant Professor) in the College of Computing at Georgia Institute of Technology. She earned her doctoral degree from the George Washington University, where her dissertation work was on automatic generation of context-sensitive animated help. Her interests include model-based user interface technology, automatic generation of intelligent help for on-line applications, multimedia help, interactive help, high-level specifications of user interfaces, the user interface design process, adaptive interfaces, usability testing, and international user interfaces.

13.3 Pablo Castells

Pablo Castells is a visiting scientist at the Information Sciences Institute. He received his Ph.D. degree in Computer Science in 1994 from the Universidad Autonoma of Madrid, Spain, where his dissertation was on the use of metaknowledge and high-level heuristics as a way to provide guidance and control for automatic problem solving in mathematics. In the past years Dr. Castells was involved in several projects funded by the Spanish government in the area of knowledge-based systems. His research at ISI is currently focused on providing knowledge-based support for user interface design, in the context of a model-based framework.

13.4 Jeyakumar “JK”Muthukumarasamy

Jayakumar Muthukumarasamy, is a member of the technical staff at Silicon Graphics. He has an M.S. in Computer Science from the Georgia Institute of Technology. His interests include programming, user interfaces, and distributed systems.

13.5 Ewald Salcher

Ewald Salcher is a doctoral candidate at the Institute for Computer Graphics at Graz University of Technology. He expects to complete his degree in July, 1996. Mr. Salcher is interested in tools for raising the level of abstraction for programming user interfaces, and in semi-automatically generating user interfaces from models of the semantics of an application.