The MASTERMIND User Interface Generation Project

Thomas Browne, David Dávila, Spencer Rugaber[1], Kurt Stirewalt
Graphics, Visualization, and Usability Center
Georgia Institute of Technology

I. Model-based User Interfaces

A. Motivation

Graphical user interfaces (GUIs) are difficult to construct and, consequently, suffer from high costs. Automatic generation of GUIs from declarative descriptions can reduce costs and enforce design principles. If, in addition, the declarative descriptions are broken into separate components, called models, modularity is improved, and advanced features such as design critics, context sensitive help, and dynamic reconfiguration are enabled.

MASTERMIND (MM) [Szekely96] is a research effort in the area of model-based user interfaces. It is a joint effort of the Graphics, Visualization, and Usability Center (GVU) of the Georgia Institute of Technology (GT) and the Information Sciences Institute (ISI) of the University of Southern California. This chapter describes the model-based user interface technology developed at GT and how it can be applied to designing a web browser.

MASTERMIND supports designers. Their job is to design user interfaces for software systems. They may or may not participate in the design of the software systems themselves. There may be more than one designer, and the designers may be broken into teams. Furthermore, the teams may be divided along functional lines, with, for example, one team responsible for screen layout, one for dialogue, and one for application software. Multiple designers require modularization and integration. Given a declarative specification mechanism, this implies strong support for consistency checking among the parts of a design.

designer: someone who specifies or constructs a user interfaces using MASTERMIND

Our customers produce user interfaces. MASTERMIND is designed to support the rapid production of high quality and powerful user interfaces. It can accomplish these goals because it is model-based.

model: a declarative specification of the structure and behavior of a software component

Models are declarative because they do not contain procedural code, such as C++ programs. Instead, models contain descriptive statements at a high level of abstraction. An added benefit is that the declarative style enables inferencing about an interface, both at design time and when the interface is eventually used by its end users.

end user: the ultimate users of the interfaces constructed by MASTERMIND

MASTERMIND declarations can be both abstract, thereby supporting rapid development, and detailed, providing precise control of the behavior of the interface being designed. That the models are declarative and modular enables inferencing at either UI generation time or at runtime. The inferencing mechanism support design critics, thereby improving quality, and powerful run-time features such as context-sensitive help, thereby extending the capability of the interface being designed. Hence, MASTERMIND addresses its customer's requirements of productivity, power, and quality.

B. State of the Art

Egbert Schlungbaum and Thomas Elwert provide an examination of other model-based approaches to user interface generation, the different models they use and the interface generation from the models [Schlungbaum96]. In their examination, they distinguish the approaches with three main distinctions:

---

[1] Point of contact: spencer@cc.gatech.edu; (404) 894-8450.

- the notation used to express the models
- the run-time environment they provide
- the explicit use of dialogue modeling

The notations that these systems use are often developed for the individual system. A number of groups have developed systems to use well-known software engineering notations. Sometimes these systems provide their own run-time support to control and execute the interface, while other systems only produce a description of an interface to then be used by other user interface management systems (UIMS).

UIDE [Foley95a] uses its own notation to declare a model of the application tasks to control the interface and an interface model to describe operations and constraints on application-independent tasks. These models are also used to generate run-time context-sensitive help automatically in the UIDE run-time environment. UIDE does not use explicit dialogue but rather relies on the use of preconditions and postconditions to sequence activity.

HUMANOID [Szekely93] uses a declarative language to express application semantics, presentation, input gestures and results, constraints on the ordering of commands and inputs, and side-effects of user actions. Presentation and gestures are defined using templates, while the dialogue constraints are derived from the application semantics. HUMANOID also provides a runtime system to control the designed interface.

MECANO [Puerta94] also provides it's own language (MIMIC) to define models of the type of users, the tasks and goals of the user, the domain that the user can affect by means of the interface, the presentation of the interface, and the dialogue between user and interface. These models are used by their own run-time system to generate and control the interface.

TRIDENT [Bodart95], GENIUS [Janssen93] and JANUS [Balzert95] all take a different approach in that they use software engineering notation for their models. Specifically, TRIDENT use Activity Chaining Graphs and entity-relationship diagrams to express user tasks and sequencing information. GENIUS uses an existing data model to define views that are used by Dialogue nets and for layout generation, while JANUS interfaces are generated by the application of knowledge base information to the results of object-oriented analysis. All three of these systems also generate a text description of the interface to be used by an existing UIMS.

TADEUS [Schlungbaum96] uses models of the task, problem domain and user to specify the requirements for a user interface. A dialogue model is then developed from those three domain models, describing both the static layout and dynamic behavior of the system. These four models are then used to automatically generate an interface along with some additional information supplied by the designer.

While these systems are almost evenly split on whether they develop their own notation, and their own run-time environment, only MECANO and TADEUS use an explicit dialogue model. MASTERMIND also includes an explicit dialogue model believing that such information will help generate better and easier interfaces.

II. The MASTERMIND Conceptual Architecture

A. MASTERMIND Models

MASTERMIND supports the automatic construction of user interfaces from declarative models. Three types of models are currently supported: the presentation model, the dialogue model, and the interaction model. In addition, a user interface state model (the context model) is now being designed. Furthermore, the interface between the user interface and the application must be specified by the designer. Because this application interface specification (the application wrapper) does not specify any semantics, it is not a full-blown model. Nevertheless, it serves an essential role in the overall design of the UI.

1. The Presentation Model

The presentation model (PM) describes the constructs that can appear on an end user's display and the dependencies among them. It is being developed by our colleagues at ISI. Although we will allude to its contents in the examples given below, further information should be obtained from their papers [Castells96].

2. The Dialogue Model

The dialogue model (DM) describes the various low-level input activities that may be performed by the end user during the course of using a MASTERMIND generated UI. This includes their relative orderings and the resulting effects they have on the presentation and the application.

3. The Interaction Model

The Interaction Model (IM) specifies the set of possible low-level interactions between the end user and the MM runtime environment. The set is determined by the underlying toolkit technology against which MM code is generated. Moreover, an interaction's primary value is in specifying communication from the end user to the other components of MM. We think of this communication as being *tight* in the sense that efficiency concerns are paramount. Specifically, we want to take advantage of underlying tool kit interactors to implement the interaction. And the interaction is atomic in the sense that no MM dialogue reasoning is used to effect the interaction.

interactor: one of a fixed set of toolkit software devices for communicating end user actions to MM.

4. The Application Wrapper

A fourth component that we have made use of is called the application wrapper (AW). It specifies the interface between MM and the application functionality. The wrapper is specified using IDL, the interface definition language for the CORBA distributed computing environment [OMG91]. Because names and types are specified, but semantics are not, AW does not qualify as a MM model. Nevertheless, it provides an essential piece of the declarative description from which a MM UI is generated.

5. The Context Model

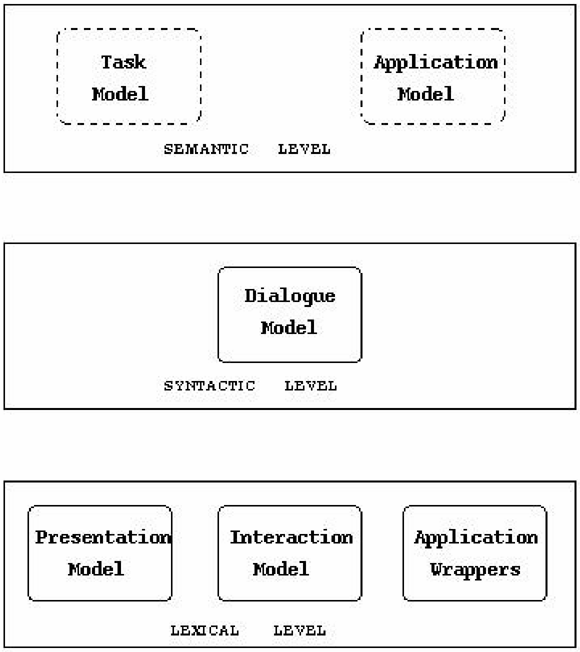
An important, but not yet developed, part of the MM conceptual architecture is the context model (CM). This model describes designer-specified state information of which the UI must be aware. For example, if it is possible for an end user to select a particular visible item and if it is important for the selected item to be visually highlighted, then it may be useful to define a context item corresponding to "currently selected item".

6. Other Models

The MM runtime architecture supports any number of cooperating models. Among those we think promising to eventually explore are a user model (e.g. novice, expert), a toolkit model (the underlying UI software for managing the screen and capturing end user actions), a display device model (enabling intelligent screen real estate management), a user task model (specifying how end user intentions are organized), and the evolution of the Application Wrapper to a first class model.

B. The Ontology of MASTERMIND Models

Ontology: a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents [Gruber].



The MASTERMIND model ontology is influenced by two ideas. The first is programming language design, which provides foundations for any modeling process. The second is object-oriented user interface toolkit technology, which provides abstractions and control mechanisms.

Interface design is analogous to programming language design [Foley95b]. Interfaces can be thought of as being composed of two languages: one in which the end-user communicates to the computer, and one in which the computer communicates to the end-user. The act of engineering interfaces can be therefore thought of as the simultaneous design of these two languages. Independent of this is the emergence of new abstractions in the object oriented user interface toolkit community [Myers90a, Myers90b; Myers92]. Two constructs, constraints and interactors, permit sufficiently declarative models to instantiate the design ontology and compile into executable code.

constraint: a declarative software device for enforcing dependencies among components of a UI.

The language analogy suggests applying the phases of programming language design to the simultaneous design of the input and output languages. There are three phases in this process:

1. Semantic Design

Semantics refers to the meaning or intentions of the end user. User task analysis [Diaper89] is applied to codify a system at this level. This is the understanding the end user will have in his or her mind when using the system. This conceptual model is refined into a detailed semantic model of the system by incorporating the functional requirements of the application.

2. Syntactic Design

A syntactic model is defined which denotes the detailed semantic model. Forms in the syntactic model represent procedures that, when executed, cause some semantically prescribed effect to occur. The primary difference between syntactic and semantic models is that syntactic models are specified procedurally (as an ordered series of steps); whereas semantic models are non-procedural (declarative). Dialogue techniques are syntactic; whereas the user tasks that these dialogue techniques are employed to accomplish are semantic.

3. Lexical Design

A lexical model, describing low-level tokens, is defined alongside the syntactic model. For the input language, these tokens include keystrokes, mouse clicks, or mouse motion. For the output language, these tokens include output characters, beeps, or graphical widgets. Sometimes lexical tokens are organized into *gestures*.

gesture: a stereotypical sequence of end-user input actions such as clicking on an object and then moving the mouse.

Experience in programming language design dictates that syntactic and lexical models correspond to compilable programs; whereas semantic models correspond to declarative specifications. For semantic models to be executed, they must be converted into a procedural form. Most semantic models have features that make this conversion practically impossible. When possible, the conversion is done on the fly, meaning that executable semantic models tend to be interpreted rather than compiled. As a general rule, code generators cannot be applied to semantic models because there are no compilation techniques that render efficient object code from them (if there were, the models would be syntactic).

MASTERMIND models address the syntactic and lexical levels. The language analogy, while conceptually elegant, is not feasible if the lexical level embodies keystroke level input events and atomic output events like beeps, or flashes. This is analogous to a compiler whose lexical analyzer feeds single characters to its parser. Compiler theory teaches us that single characters are inappropriate tokens because they inject too much complexity into the corresponding parsers. To see how this problem manifests itself in UI design, consider a drag-and-drop interface. A burst of activity begins with an end user clicking down on an icon. Immediately a shadow icon appears, and as the end-user moves the mouse, the shadow icon changes its position on the screen. The burst of activity ends when the end-user releases the mouse button signifying a drop over some other icon. If we consider each mouse move event and icon redraw to be tokens, then the syntactic description of drag and drop will be extremely complex. Compiler theory suggests we simplify this problem by making tokens more abstract than just mere input and output events. In compilers, tokens embody patterns of input characters. Lexical UI design should, therefore, identify patterns of input/output events and treat these patterns as tokens. User-computer interfaces are characterized by bursts of tight, high volume, feedback intensive, input/output event sequences, and these sequences can be described by patterns. To represent lexical patterns, we pull ideas from object oriented UI toolkits.

Object oriented toolkits like Garnet [Myers90b], and Amulet [Myers95] provide interactors which encapsulate these tight input/output protocols into implemented units that may be selected from a library and specialized to a particular use. In both toolkits, the number of interactors is fixed and relatively small. The MASTERMIND interaction model is an abstraction of interactors, and we consider it a lexical model. As another example, consider how window sizes might need to change dynamically to accommodate the insertion/deletion of graphical objects. Object oriented toolkits use constraints to declare that an object's attributes be dynamically recomputed when another object's attributes change. This mechanism abstracts away a great deal of sequencing that would otherwise have to be implemented in the two languages. The MASTERMIND presentation model allows designers to use constraints when specifying presentation layout, and we consider it a lexical model. The final lexical specification is the MASTERMIND application wrapper. The AW is currently nothing more than an application program interface for invoking behavior in a (possibly distributed) application. Treating it at the lexical level allows us to consider method invocations as tokens.

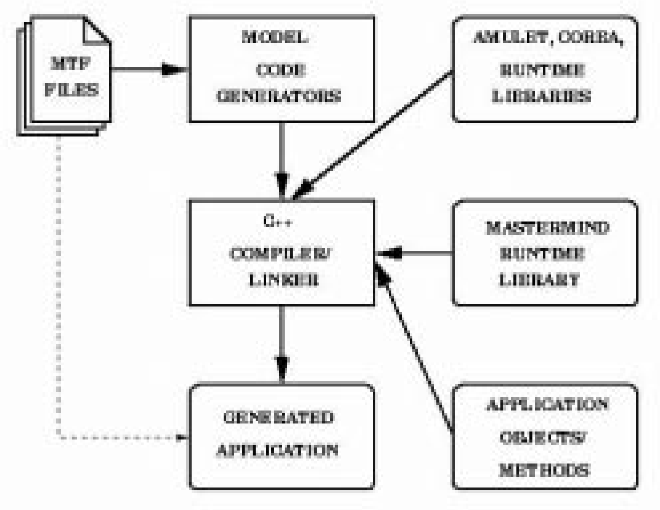
The syntactic component of design is captured by one MASTERMIND model, the dialogue model. Syntactic models abstract a meaningful ordering structure over tokens described in the lexical models. In programming languages, syntactic models are usually specified by context-free grammars whose terminal symbols correspond to tokens in a lexical model. Unfortunately, there are orderings mechanisms in dialogue models that are not expressible using context free grammars. The essence of grammars, however, are hierarchical ordering constraints over tokens. The MASTERMIND dialogue model declares hierarchical ordering constraints over interaction tokens.

In MASTERMIND, task organization and application functionality are semantic concepts. Interaction, presentation, and application invocation are lexical concepts, and dialogue is a syntactic concept. These models must precisely define the input and output languages and express their interleaving.

This ontology is depicted in the figure below:

C. UI Generation Process

MASTERMIND tools synthesize user interfaces from three distinct models of UI behavior. To go from models to executable code, MM employs model-specific code generators. Each code generator outputs C++ source files which are then compiled and linked with a number of runtime libraries. Since there are necessarily dependencies among models, model instances must refer to elements of other model instances. We use the MASTERMIND textual format (MTF), a frame-based notation, for expressing instances of models in the ontologies (presentation, dialogue, and interaction).


MTF FILES
MODEL CODE GENERATORS
AMULET, CORBA, RUNTIME LIBRARIES
C++ COMPILER/ LINKER
MASTERMIND RUNTIME LIBRARY
GENERATED APPLICATION
APPLICATION OBJECTS/ METHODS

The MASTERMIND architects have defined the ontology for each model, in a second notation called the MASTERMIND representation language (MRL), to be a set of classes with attributes. Each class has a fixed number of attributes, and each attribute has a fixed type (class). In an ontology, classes are related by subclass inheritance, meaning that if class A contained two attributes foo and bar, a class B subclassed from A would contain two attributes named foo and bar of the same type in A and any additional attributes declared specific to B. Subclass inheritance governs relative names and types of attributes--not their values.

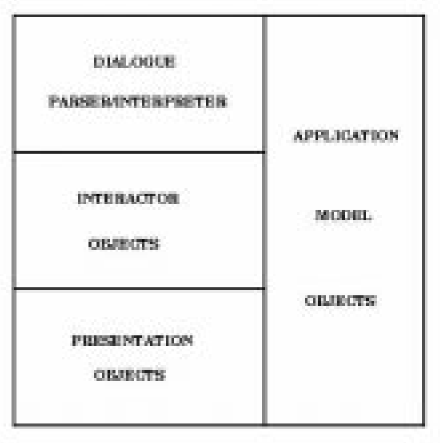
MASTERMIND architect: one who defines model ontologies

Designers who build models in the ontologies specify instances of these classes. These instances are called *objects,* and MTF provides a mechanism called prototype-instantiation that creates a new object (called an instance) from a prototype by copying all of the attributes of the prototype into the instance. When attributes of a prototype object change, the corresponding attributes of instances of the prototype object change accordingly. For this reason, we call prototype-instantiation another form of inheritance. Whereas subclass inheritance relates name and type of attributes; prototype inheritance relates values of attributes in different objects.

Occasionally, designers will want to subvert the prototype-instance relation. This is done by explicitly over-riding an inherited attribute with a new value or expression. Overriding is particularly useful for choosing to specialize a default behavior. As designers gain experience in defining and using models, we expect them to build new interfaces by instantiating objects from old interfaces and overriding a few attributes to fit these objects into the new context.

In addition to prototype-instance inheritance, MTF allows attribute values to be specified as *formulas,* computing one attribute value from the values of other attributes. Our code generators translate formulas into code that maintains the run-time consistency of the attribute values under dependent attribute modification. The presentation model code generator, for example, translates formulas into constraints which are kept consistent by a run-time object system. The interaction code generator, on the other hand, translates formulas into methods and adds calls to these methods at appropriate points in the code generated from other models.

The code generators create object code that fits into the run-time architecture.



The run-time architecture has four components: dialogue, interactors, presentation, and the functional core of the application. Connecting lines in this figure describe inter-component communications, some of which must be expressed by the designer. The reification of model inter-dependence varies according to the connected pair of models.

D. MASTERMIND Dialogue Notation

The MASTERMIND Dialogue Notation allows designers to specify the permissible ordering of the end user interactions that carry out some task. The notation described below is textual. Later a graphical design tool is described that provides an alternative notation. A dialogue model is an expression in this notation, and a dialogue parser is a machine generated from a dialogue model. This relationship between model and parser is similar to that between a context-free grammar and its associated parser. Elements of dialogue models are called symbols and may be either tokens or variables.

A dialogue entity that the designer distinguishes in the dialogue model is called a dialogue symbol. Dialogue symbols are given names and other attributes and are either tokens or variables. A dialogue token is a terminal symbol that represents either an application invocation or a user interaction. A dialogue variable is a non-terminal symbol that represents an ordered aggregate of other symbols. Variables are related to these other symbols by production rules.

1. Dialogue Tokens: Interaction/Application Invocation

End-user interactions and application invocations are the tokens accepted by the dialogue parser. Think of them as being *leaves* in a parse tree. Application invocation tokens are specialized with the information that MASTERMIND needs to invoke a computational service when this token is expected to be matched. Interaction tokens represent notification by an interaction technique specified in the interaction model.

interaction technique: a construct that "implements the hardware binding portion of a user-computer interface design" [Foley95b]

When an interaction token is expected, the associated interactor is enabled. When an interactor is enable, it may accept device input and notify a dialogue token. If, on the other hand, this token is no longer expected by the dialogue (perhaps because an alternative was selected) then the interactor associated with this input is disabled. When an interactor is disabled, it may not accept device input. When an interactor notifies a dialogue token, the dialogue parser considers this token to be accepted.

2. Dialogue Variables: Composite Ordering

The dialogue model allows designers to organize tokens using a declarative mechanism similar to production rules in context-free grammars. The most common form of ordering in this language is achieved by declaring an ordering (i.e., temporal) invariant over a sequence of subdialogue symbols. MASTERMIND supports five such invariants: Alternative, Sequential, Parallel, Exclusive, and Until.

We chose this particular set of orderings over others because they represent distinct points in a two dimensional design space of associative control regimens. The first dimension captures ordering policy; whereas the second captures the interpretation of user activity. MASTERMIND supports three ordering policies: sequential, alternative, and unconstrained. The sequential policy imposes a linear ordering, the alternative policy imposes an exclusive (choose one) ordering, and the unconstrained policy imposes a do all ordering but does not specify a relative ordering among the constituent sub-dialogues. Our first interpretation of the alternative ordering policy, for example, deduced the user's choice of subtask from the first recognized interaction with any subtask. That is, if the user is faced with choosing from among three subtasks T1, T2, and T3, and he or she performs an interaction within T2, we might deduce that T2 was the choice and disable tasks T1 and T3. The alt operator implements this behavior.

The second dimension in our design space has two values. Either the activity is synonymous with choice commitment, or the two are separate. This leads to the following design space (which we have populated with our chosen orderings):

	Sequential	**Alternative**	**Unordered**
Commit	seq	alt	par
Delay Commit	until	excl	par

Designers may specify tasks to be either optional or repeatable. In either case there are two possible agents who may choose whether to execute the optional task or repeat the repeatable task. These agents are the end-user and the environment. We see the agent of choice as being independent of the nature of the ordering and so define four possible combinations.

	User Choice	**Environment Choice**
Optional	opt	cond
Repeatable	star	while

The vertical dimension specifies optional behavior. The designer could allow an end-user to "optionally execute a task", that is, to skip the task or execute it only once. Alternatively, the designer could allow the end-user to skip the task, or execute the task once and then repeat the whole process. These alternatives correspond to the distinction between conditional statements and loops in programming languages. The horizontal dimension specifies the agent that enacts the optional behavior. The agent could be the end-user or it could be the machine. When the end-user is the agent, the optional tasks have behavior that is non-essential to the accomplishment of the goal. For example, a mail tool allows you to enter a subject for a message, but does not require it. When the machine is the agent, however, the method of achieving the goal changes in response to the condition of the external environment.

3. Data Parameters

Since dialogue sequences interactions and application invocations, dialogue productions are a natural place for the transfer of data from the user into the application methods. Furthermore, the hierarchical nature of dialogue models gives rise to a convenient lexical scoping for temporary data variables. For the sake of consistency in model naming, we call these local variables data parameters. The designer may introduce zero or more data parameters with any dialogue symbol. Data parameters may be initialized, and sub-dialogue symbols may see and modify the parameters of their ancestors in the hierarchy. Data parameters may be modified either by an interaction technique binding (discussed later) or by an effect, which is a declarative specification of the state of the dialogue after the activity associated with a dialogue symbol completes. Likewise, data parameters may be used by interaction techniques, supplied as parameters to application methods, or occur in dialogue symbol enabling preconditions. We don't use preconditions or effects in the web browser example, and so do not expand upon these topics here.

E. Dialogue Validation

Dialogue models specify the legal orderings of user and system interactions. We implement dialogue models by generating a component called a dialogue parser. The parser maintains a running representation of the current dialogue state, dynamically computing the set of interactions legal in this state, and enabling only the interactors associated with this set. Generating a parser automatically from a hierarchical dialogue specification is complicated by the fact that the language associated with hierarchical dialogue specifications is not context free. We solved this problem by identifying a set of hierarchical temporal operators and building canonical state machines that, when connected according to a dialogue hierarchy, enforce the intent of each operator. The state machines are complex, and we employed a technique called symbolic model checking [McMillan92] to validate them against a set of desired behaviors. This degree of confidence is a direct benefit of the model-based approach. The technology allowed us to confidently create a dialogue parser generator, and we expect it will also be useful for driving dialogue debugging tools.

F. Presentation Model

The MM presentation model [Szekely96] was developed at ISI. Though thorough coverage is not provided in this chapter, the following summary is intended to provide sufficient background to understand the web browser case study described below.

Presentation models are declared as a hierarchical collection of method-less objects whose attributes are related by formulas. These model objects are translated into run-time objects with special attributes called *slots*. Run time slots store values that may be set in one of two different ways. An external agent may physically place a value into a slot, or a slot value may change automatically in response to changes in other slot values. The mechanism that enforces the latter behavior is a constraint. Presentation models have a declarative mechanism called an *expression* that establishes inter-attribute dependencies. These expressions are translated into run time constraints. In addition to establishing dependencies among presentation model object attributes, expressions may also establish dependencies upon application object attributes. Expressions, therefore, are the mechanism for presentation/application binding. They represent an adequate communication mechanism for the presentation model and the application wrapper because the flow of information is always one-way from the application wrapper to the presentation model.

G. Binding in MASTERMIND Models

The example in the last section described communication between the application wrapper and the presentation model. This straightforward communication mechanism is not sufficient for the general case of inter-model communication. The communication between the presentation and interaction models, for example, involves a two way communication that is an example of inter-model *binding*.

binding: the process by which a designer specifies the occurrence of a specific inter-model dependency.

Binding involves relating elements and behaviors of one model to elements and behaviors of another. The binding problem occurs in any multi-model system, and it occurs between any pair of mutually effecting models. Binding is generally difficult because models might have different mechanisms for composing model elements, and a binding must respect each mechanism in order to be used correctly. This unfortunate property can lead to inelegant and esoterically technical binding strategies, and it precludes any hope of a universal binding notation. The MASTERMIND approach fits a custom binding strategy to each configuration of inter-dependent MASTERMIND models. Currently there are only four such configurations: Presentation/Interaction, Presentation/Application, Dialogue/Application, and Dialogue/Interaction.

Presentation/interaction cooperation involves a high volume of tightly synchronized input and output activities. Interactors field user input events, decide what purpose they represent and set presentation slots as appropriate. Interactor objects are the run time images of design time abstractions called interaction techniques. The manifestation of a binding between interaction techniques and presentation models is an attachment of interactor objects to run-time presentation object slots. Details of these bindings are declared in the interaction model. Designers, when specializing interaction techniques to a particular application, will give the name of objects/attributes that instantiate the technique. The interaction model translator then uses this information to build an interactor object and attaches it to the appropriate run-time presentation objects.

The binding of presentation and application is one way only--information flows from application wrapper objects into the presentation. Method invocation is declared in the dialogue model and represents a binding of dialogue and application models. This binding is accomplished at design time by associating a token with the methods that need to be called. As per the language analogy, as parse state (dialogue state) changes at run time, these tokens may become expected. When this occurs, the methods are invoked, and the symbols are considered matched. The binding of data flowing into and returning from a method call is accomplished using a *data parameter*.

The dialogue model distinguishes two classes of tokens. One represents application invocation and the other user interaction. Associating dialogue terminals with user interactions is a binding of the dialogue model to the interaction model. As with presentation/interaction bindings, the specification of dialogue/interaction binding is a specification of dialogue terminal obligations.

H. The MASTERMIND Interaction Model.

Presentation and dialogue models do not directly model end-user input. MASTERMIND defines input in a separate interaction model. The interaction model ontology appeals to the influence of input on model binding and the mutually exclusive nature of interactions.

Interaction techniques implement the hardware binding portion of interface design. MM interactions abstract this definition to the binding of input event sequences with presentation, dialogue, and application model behavior. The ternary nature of these bindings makes them difficult to delegate to either the dialogue or presentation model. The X windows Xt toolkit, for example, employs widgets that invoke application callbacks when user events occur. For interactions like drag-and-drop, this approach necessarily scatters the implementation of the binding among all possible widgets that might participate. This is not only difficult to implement but also difficult for designers to reason about.

The interaction model ontology is organized into classes whose names mimic those of familiar interaction techniques. Attributes of these classes capture information needed to bring the design time binding to run-time fruition. We later describe three such classes.

Interaction techniques represent natural dialogue tokens because they are logical end-user computer exchanges, and, because of mutual demands upon shared resources, they tend not to be interleaveable. A shared resource in this context is an input device like a mouse, the keyboard focus, or voice input. For example, consider the dragging and dropping one presentation object over another. MASTERMIND considers this one logical interaction. It cannot be interleaved with, say, clicking a button because during one or the other interaction the mouse is being used and cannot be relinquished temporarily to complete the other. Interactions encapsulate access to shared resources into atomic tokens that can be ordered according to the dialogue model. These interactions are complicated by the tight web of resource inter-dependencies that govern an interaction. A specific interaction might involve presentation feedback to the end-user, a commitment to a specific dialogue token, and communication with an application wrapper. The designer must understand how the pieces of these three components interact in order to specify the behavior he or she has in mind.

Since interaction techniques naturally capture logical end-user interactions, and since they also localize complex multi-model dependencies, MASTERMIND allows designers to choose an interaction technique and then supply information that completes the binding of the three models according to the interaction protocol. For the direct manipulation example, the designer would choose a DragAndDrop interaction technique, supply the names of presentations to which it should apply, and supply the names of the dialogue tokens to notify when the interaction completes. Some interaction techniques may require a large number of designer supplied parameters to complete the binding. This, unfortunately, gives the interaction model a synthetic feel, but we envision a tool that will allow designers to graphically attach interaction techniques to task and presentation models so that many of these parameters may be inferred rather than explicitly supplied.

This chapter uses two interaction techniques--Selection and FormBasedEntry--which we present here. Common to all techniques is the notion that they are enabled and disabled according to the state of dialogue objects. We define the super-class of all interaction techniques to capture this behavior:

InteractionTechnique
enable when [dialogue-symbol] is [dialogue-state].
disable when [dialogue-symbol] is [dialogue-state].

The enable:when:is: statement identifies a dialogue-symbol whose transition into dialogue-state enables this interaction technique. The disable:when:is: statement performs a similar function for disabling the technique. Though interaction techniques always activate dialogue tokens, the techniques themselves may be enabled and disabled by dialogue variables.

1. Selection Interaction Techniques

Myers [Myers96] distinguishes between two kinds of selection. In one the user can mouse over the possible selections and observe interim feedback from the interface. This feedback identifies the selection that will be chosen should action be taken. In the other form, no such feedback is given. These two forms have different interaction protocols and different binding obligations. We therefore represent them in two separate interaction techniques: OneShotSelection and InterimFeedbackSelection.

The binding obligation of selection techniques typically must associate presentation entities with dialogue entities and possibly transfer some application specific data in the process. We have observed that this association has two natural cardinalities. Either many presentations may be associated with a single dialogue token as in a drawing tool that allows the user to choose one of many presentations to delete, or presentations can be in one-to-one correspondence with dialogue tokens, as in an email tool whose buttons denote different functions to perform. To capture this association, the interaction model provides designers with a template called a PresentationDialogueAssociation (PDAssoc). Selection interaction techniques are parameterized by this association. The two cardinalities have different features, and so we define two subclasses of PDAssoc: OneToOnePDAssoc and ManyToOnePDAssoc. The template for each appears below:

ManyToOnePDAssoc : PDAssoc
choose [part-name] from [pres-group].
within [dialogue-token] assign [dialogue-parameter] to [pres-datum].

This template assumes that many-to-one associations occur when the end-user must select one from among many parts of a presentation group. The choose:from: statement identifies a group (pres-group) from which the end-user will choose a part and gives this chosen part a name (part-name) which can be used in the within:assign:to: statement that follows. The latter statement identifies a dialogue token to be activated when the selection is made and allows the designer to bind some data from the presentation object (pres-datum) to a data parameter (dialogue-parameter) of the dialogue token. The pres-datum object is usually defined by means of a formula over the designated part-name. The code generator checks to make sure that the types of these entities are compatible.

OneToOnePDAssoc : PDAssoc
assign [presentation-object] to [dialogue-token].
within [dialogue-token] assign [dialogue-parameter] to [pres-datum].

The OneToOne association is similar to the ManyToOne except that the choose:from: statement is replaced by one or more assign:to: statements. The assign:to: statement simply identifies dialogue tokens to notify when certain presentation objects are selected.

With this in place, the two selection techniques are simple to use.

OneShotSelectionInteraction : InteractionTechnique
bind [PDAssoc].

InterimFeedbackSelectionInteraction : InteractionTechnique
bind [PDAssoc].

Since each inherits from InteractionTechnique, they inherit the enable/disable syntax.

2. Form Entry Interaction Technique

Text entry interactions are abstracted into the more general interaction technique FormBasedEntry which controls text entry with and without the support of application completion queries. The relevant attributes of this technique are listed below:

FormBasedEntry : InteractionTechnique
notify [dialogue_token] upon [input_events]
attach to [presentation]

enable input when [dialogue_symbol] is [dialogue_state]
disable input when [dialogue_symbol] is [dialogue_state]

Unlike the Selection technique, text entry has two logical interpretations for being enabled. Enabling could mean it is legal for users to enter text into the field, or it might mean it is legal for users to communicate a commitment of edited text to the system. In text entry, the user often communicates the commitment by hitting the return key. We allow designers to apply this distinction by providing two separate enable/disable specializations--one for input and one for dialogue activation. Of course, the enabling for text entry and the enabling for do it entry are often synonymous.

I. MASTERMIND Design Method

Building a UI in MASTERMIND means constructing models. From the examples that we have tried, we are beginning to understand how best to go about this. Our current understanding forms the basis for a model-based user interface design method. There are five components to an application system built using MASTERMIND. These components are distinguishable by what they are designed from and their corresponding position in the MASTERMIND design method.

Components designed from artifacts (AW, DM, PM)
A component that binds two other MASTERMIND components (IM)
A component for shared data communication between all components (CM)

In MASTERMIND, there are three components which are relatively independent of each other, because they are designed not from other MASTERMIND models and bindings but from design artifacts. These components should be the first ones built in MASTERMIND, because of their close relationship to the design method, and because they can be designed independently of each other with little modification required for integration. This provides a design team with the ability to work on several components at the same time, with specialists devoted to each component.

These components are the presentation model (which is developed through prototyping or storyboarding), the application wrapper (developed from the application code), and the dialogue model (designed based on a formalized task analysis and specification). While some coordination in developing these components is required, such as common naming for application methods listed in the AW and invoked by the dialogue model, it is at a very high level of design. A small amount of time spent during design to agree on the application and interface components and to application method and parameter names should be enough to allow the completion of these models independently so that little if any modification will be required.

Once the presentation, application and dialogue components have all been designed, the next step is to build the components that are responsible for the binding and communication between them. The first step is the binding of the presentation to the behavior of the system (its dialogue). That is the purpose of the interaction component. Since there is already an implied binding between the dialogue model and the application wrapper (through the use of a common naming convention for application methods), binding the dialogue model to the presentation unites all of the components into a single system.

The final component that needs to be constructed is the Context Model. It is only after all of the other components are designed and integrated, that a designer can be certain of which additional information needs to be shared between components, what application data is reflected in the presentation, or affects what action the user is capable of performing next. By waiting until all of the other components are designed to build this model of communication, the amount of modification to any of the earlier components is minimized and if the necessary communication is identified ahead of time, eliminated.

III. Case Study

A. Context: Web Browsers

While the relationship between MASTERMIND components in general can be examined in the abstract, it is more difficult to explain the actual method for designing the individual components in such a manner. For this reason, the development of individual components will be examined in the context of designing the user interface for a web browser.

First, a simple browser will be designed, capable of web navigation, saving pages to files, and printing a hardcopy of the page. This will not only demonstrate the process for designing MASTERMIND components, but will also show how the application wrapper, dialogue model and presentation model can be developed independently and how that provides flexibility for prototyping. In addition, it will demonstrate how MASTERMIND allows multiple interactions to accomplish the same task, and how different interactions can use the same application code to accomplish different goals.

Once the simple browser has been developed, a history mechanism will be added to the system to maintain what web pages the user has visited, and to offer operations such as traversing forwards and backwards through the list, or jumping to any point in the list. This will demonstrate how a MASTERMIND application can easily be modified and evolved over time.

B. Dialogue Model Example

For this example, assume that the following design specification for the browser has been given. That is, the browser is to allow the end-user to do the following:

- visit a new page by giving the URL
- go to a specified home page
- request a reload of the current page
- print a copy of the current page to any of a number of printers
- save a copy of the page to a filename of their choice
- stop any one of the previous actions at any time they are in progress

- quit the web browser at any time

The first step in the design method is to identify all of the operations that the end-user can perform according to the specification--not how the user performs them or specifies them, but the actual functions they can perform. In this specification there are seven--open a new page, go to a home page, reload the current page, print a page, save a page, stop an action, and quit the application. Notice that there is no differentiation between opening a page by name, or by following a link--that is a distinction of how the URL is provided, not whether one is specified, and so it is not a distinction made in the dialogue model.

The next step is to determine how these operations are related to one another (see Dialogue Model for Simple Browser figure below). Since Stop can affect Open, Home, Reload, Print or Save, it suggests that those five actions should be related. Since we want the browser to only be able to perform one of these actions at a given time, and that once one of them is started, the other actions are no longer available, we use an ALT ordering to organize these five operations together.

Stop should be available whenever one of these five operations is being performed, and executing a stop action should abort the current activity. This fits the idea of an EXCL ordering. Stop will be available until one of the other actions completes, but if Stop is activated, it will abort the current action. These pairings complete the basic navigation available to the browser.

The Navigate grouping though only allows one action to be completed. Since the browser allows any number of actions until the user decides to quit, Navigate should be in an infinite WHILE loop. This still allows Stop to abort any current action, but will then allow the user to choose a new action. Quit is then related to the ability to Navigate by another EXCL ordering. Since Navigate is an infinite loop, the only way the dialogue can terminate is through the completion of the Quit action. This provides us with exactly the desired behavior for the browser.

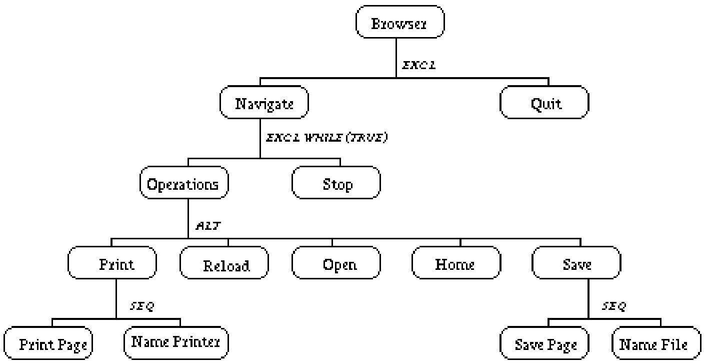
Finally, the implementation and behavior of each of these actions needs to be defined. It would be easy to design Quit, Stop, Home and Reload all to be single application methods which get invoked--they need no other information from the system other than what the application would already have, therefore they are leaf nodes in the model.

Open can also be defined as a single application method which takes a parameter indicating the URL of the page. This parameter needs to be determined in some manner (by user interaction), but no matter how the URL is determined, the behavior is the same. For this reason, Open is also just a single node in the model.

There are two steps to either saving or printing the page. The first step is to express the intention to Save or Print the web page. The second step is to then determine the parameters required for the application. When saving a page, the parameters required for the application are the filename desired, and for printing a page, the name of the printer. Only after that information is obtained can the application method be invoked to accomplish this action. Save and Print therefore have a structure of having two children in a SEQ ordering. The first child is used to capture the intent to perform the action, while the second child obtains the parameter information from the user, and invokes the application method.

Of course Save and Print could be implemented as single actions depending on the style of the interface. For instance, if a list of printers was always displayed to the user, then choosing the printer might be a single action like Open. Providing such lists, though, clutters up the screen. It is more common that these features be provided only when the user expresses an interest in them, and that is the behavior assumed in this model.

Dialogue Model for Simple Browser



C. Presentation Model Example

Since presentation has been the focus of the work done by ISI, it is treated here only in the details that are necessary for the understanding of MASTERMIND and the other components of the system. For the presentation of a Web Browser, only a few presentation primitives are required--window, button, text field, and label. These are common forms of presentation, and so other details regarding them are ignored in this discussion.

A text field is used to represent the current URL of the browser, as well as being used to let the user enter a new URL. It only needs to be one line and be editable by the user. Ignoring such considerations as the actual layout, it can be described by the following:

```
URL_field : Text_Field
visual_parameters
    editable = TRUE
layout_parameters
    width = 80
    lines = 1
    label = "URL:"
```

All of the remaining functions of the browser are to then be available either in the form of a button or by a choice from a menu. Menus are not yet one of the primitives supported. Menus are just a window which contains a collection of buttons. Recall that there is no behavior or interaction attached to buttons, since they are purely a display notation.

The menu can be defined as follows:

```
Browser_Menu : Window
parts
    Home_item : Button
    Print_item : Button
    Reload_item : Button
    Save_item : Button
    Quit_item : Button
```

The browser itself is simple a window with a number of parts in it: the menu, the buttons, the URL text field and a sub-window for the display of the page. So the entire presentation for the browser (other than layout and some other detailed parameters) is given by the following:

```
Browser : Window
parts
    URL_field : Text_Field
```

File_Menu : Browser_Menu
Display : Window
Home_button : Button
Print_button : Button
Reload_button : Button
Save_button : Button
Stop_button : Button
Quit_button : Button

The only other objects required is the 'pop-up' windows for saving pages or selecting a printer. These are simply defined as windows which either contain a Menu (for selecting which printer to use), or a Text Field (to name the file you wish to save).

D. Interaction Techniques for Web Browser

The presentation object Browser has three parts which react to input through interaction techniques. The File_Menu is controlled by a selection interaction technique that displays interim feedback, the various buttons are controlled by a selection interaction technique without interim feedback, and the URL_field is a text field that is controlled by a FormBasedEntry technique.

The menu of choices binds menu items to dialogue symbols in a one-to-one correspondence. These correspondences are expressed as instances of OneToOnePDAssoc classes.

MenuChoices : OneToOnePDAssoc
assign Browser.parts|Browser_Menu|.parts|printItem| to Dialogue.PrintPage.
assign Browser.parts|Browser_Menu|.parts|reloadItem| to Dialogue.Reload.
assign Browser.parts|Browser_Menu|.parts|homeItem| to Dialogue.Home.
assign Browser.parts|Browser_Menu|.parts|saveItem| to Dialogue.SavePage.
assign Browser.parts|Browser_Menu|.parts|exitItem| to Dialogue.Quit.

In each line of the association, a button in the browser menu is connected to a token in the dialogue model. At run time, if any of these dialogue tokens are not enabled, the labels in the corresponding buttons will be greyed out. The interaction technique that employs this binding is called FileMenu synonymously with the Netscape browser File menu that contains similar options.

FileMenu : InterimFeedbackSelectionInteraction
enable when Dialogue.Browser is enabled.
disable when Dialogue.Browser is disabled.
bind MenuChoices.

This interaction technique is enabled whenever the Browser dialogue symbol is enabled (which is the lifetime of the session), and disabled only when this symbol is disabled. Note that by capturing all of the presentation/dialogue binding in MenuChoices, only one entity need be updated to add functionality to the menu.

The binding of operation buttons to dialogue tokens is also a one-to-one correspondence.

ButtonChoices : OneToOnePDAssoc
assign Browser.parts|printButton| to Dialogue.PrintPage.
assign Browser.parts|reloadButton| to Dialogue.Reload.
assign Browser.parts|homeButton| to Dialogue.Home.
assign Browser.parts|saveButton| to Dialogue.SavePage.

OpButtons : OneShotSelectionInteraction

```
  enable when Dialogue.Browser is enabled.
  disable when Dialogue.Browser is disabled.
  bind ButtonChoices.
```

Note that the dialogue tokens activated by ButtonChoices overlap with those of MenuChoices. MASTERMIND allows designers to express bindings without concern for overlapping functionality in the knowledge that the code generators will work out the details.

Finally, the interaction technique attaching input to the text field URL_field is given below:

```
URLInput : FormBasedEntry
  enable input when Dialogue.Operations is enabled.
  disable input when Dialogue.Operations is active.

  enable activation when Dialogue.Open is enabled.
  enable activation when Dialogue.Open is disabled.

  attach to Browser.parts|URL_field|.
```

In this technique, input is enabled at any time that the symbol Operations is enabled but inactive. Users may only commit selections, however, when the dialogue symbol Open is enabled.

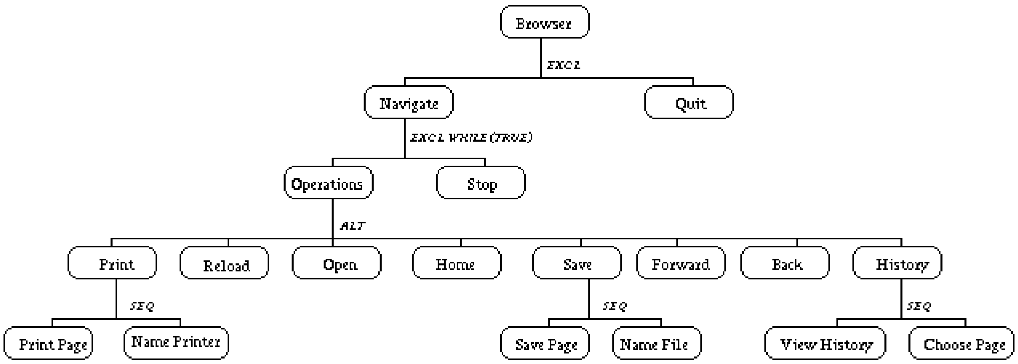
E. Dialogue Model Evolution - Web Browser History Mechanism

Once the browser has been created, adding new functionality to the system is a very simple job in MASTERMIND . In this case, a history system is added to the browser. The first job is to decide what functionality the history mechanism must provide. The end-user is allowed to traverse the history list in a linear fashion (Back and Forward functions), or to jump directly to any web site already in the history list. Each of these is a navigation operation of the same type as Home or Open and therefore will simply be additional choices available to the user at the same time--they will be children of the Alt ordering of Operations.

The next step is to decide how they will be implemented. Back and Forward are very simple behaviors along the history list which need no information other than the status of the history list. For this reason, it is decided that they will be developed as application methods which can just be invoked by the dialogue model, making them single nodes. In addition to the application method they invoke, the model also needs to know the conditions under which they may be invoked--Back should only work if there is a previous page to go back to, and Forward should only work if the current page is not the last page in the history list. This requires information to be added to the context model described below.

It looks as though going to any place in the history list can be implemented by yet another interaction binding to the Open process. This will not work however. Where following a link or typing in a URL directly already provide the URL parameter that is required by the application method, the history mechanism can not provide such a value unless the history list is constantly displayed to the user. Since that is undesirable due to the amount of screen space the list takes up, using the history mechanism is similar in structure to Print--there is a step to view the history list, and then another step to choose a page from that list. The history task needs two children defined, one for each of these steps. It can then invoke the same application method as Open, with its own parameter

Those are the only changes required to the dialogue model for the addition of the history mechanism. The only changes this requires to other models are the creation of two new application routines (Back and Forward) and their specification in the AW, a couple of new CM variables (to maintain information on the current history list), and appropriate presentation and interaction techniques for the new dialogue symbols to represent. The evolved dialogue model is shown below:



Dialogue Model for Browser with History Mechanism

F. *Presentation Model Evolution - Web Browser History Mechanism*

The presentation features for the history mechanism are merely additional instances of the objects already defined. Three new instances of Buttons are declared for Back, Forward and History. In addition, each of these three items is included in the initial menu of the system. This can be handled easily by the inclusion of Back, Forward and History in the application operation_list.

The only additional presentation that is needed is the presentation of the history list. This is defined as a simple window which contains a Menu, and is initially set to be non-visible, and only becomes visible when the user attempts to choose from the list. This structure is of the form:

```
History_Window : Window
data_parameters
    history_list : sequence < URL >
parts
    history : Button
    replicate [history] for history_list
```

This new window is the only additional coding necessary for the presentation model, other than the addition of the new operations in the application operation list. The replication system of the presentation automatically handles the new menu item and buttons for the new operations.

G. *Interaction Model Evolution - Web Browser History Mechanism*

The new input behavior is easily added to the existing interaction techniques. The behavior of the new buttons and menu option need only be added to the presentation dialogue associations. The ButtonChoices association should be extended with three new assignments:

```
assign Browser.parts|forwardButton| to Dialogue.Forward.
assign Browser.parts|backButton| to Dialogue.Back.
assign Browser.parts|historyButton| to Dialogue.History.
```

Likewise, the MenuChoices association should be extended with three new maplets:

```
assign Browser.parts|Browser_Menu|.parts|forwardItem| to Dialogue.Forward.
assign Browser.parts|Browser_Menu|.parts|backItem| to Dialogue.Back.
assign Browser.parts|Browser_Menu|.parts|historyItem| to Dialogue.History.
```

The history selection pane represents a more complicated interaction and showcases a typical ManyToOnePDAssoc. We first define:

```
GoToURLAssociation : ManyToOnePDAssoc
  choose hist_url from History_Window.parts|history_menu|.
  within Dialogue.History assign hist_url.data_parameters|url|
       to data_parameters|the_url|.
```

With these changes and a regeneration of the code, the extension is complete.

IV. Status

A. Code Generators

Each of the MASTERMIND models has a corresponding code generator. The implementation of the code generators requires three steps. Each code generator takes as input a textual MTF files and produces one or more C++ source files. Since MTF is a syntactic format as opposed to a compilable language, the code generators are distinguished by the model-specific domain knowledge they apply in synthesizing code from models. The maturity of the code generators is a function of the depth or precision of the model ontologies. The presentation code generator is the most mature as evidenced by the depth of the presentation ontology. Next is the dialogue code generator which generates state machines implementing the ordering constraints. The newest and least mature technology is the interaction model code generator. This generator currently resembles a macro processor more than a compiler, but as the interaction ontology becomes more well-defined we expect this to change.

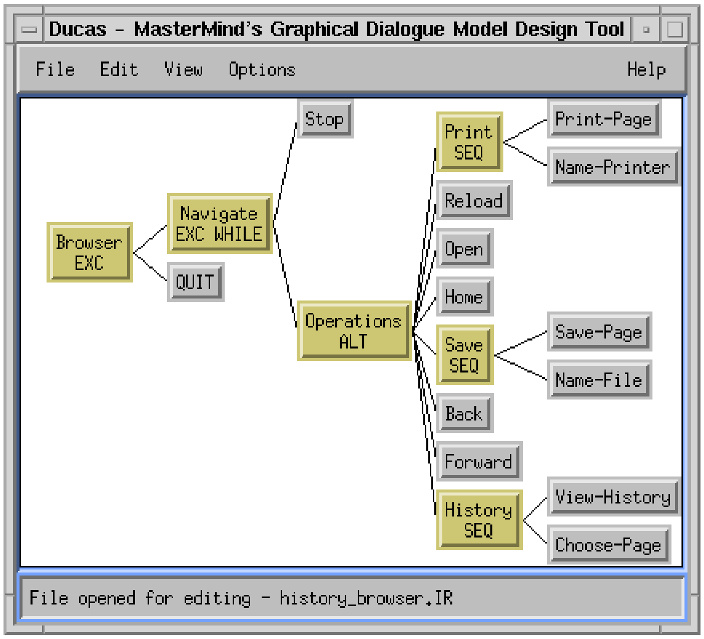
We have tried several validations of our modeling and code generation approach. One of the most interesting occurred prior to when we were able to integrate our technology with that of ISI's presentation engine. In this exercise we used HTML as our presentation notation. That is, we wanted to generate forms-based web applications where screen updates consist of automatically generated HTML pages. While bandwidth limitations make this approach is prohibitively slow for practical situations, the simplicity of the HTML presentation language was such that we could readily drive the page construction process from our dialogue model.

B. Dukas[2]

Dukas is MASTERMIND's graphical dialogue-modeling tool. With this tool, dialogue modelers can create hierarchical trees, identifying both types of leaf nodes, and also relationships between dialogue symbols (e.g., ordering, parent-child). All properties of dialogue models important for their connection to the other MASTERMIND models are also editable from within Dukas. The following screenshot shows the basic layout of Dukas, and the dialogue model for a browser's history mechanism.

---

[2] Paul Dukas is the composer of the *Sorcerer's Apprentice*.



File opened for editing - history_browser.IR

Dukas currently allows basic dialogue model editing, but there are several additions planned for future implementation. Some of these additions are enhancements to the look and feel of the tool. For example, in the future, the ordering of a group of nodes will be indicated by their shape on the display. A tool palette will be added to allow for easier editing of dialogue models. Currently the different editing commands are accessible with various combinations of mouse and keyboard input. Also, indication of which node is currently selected, and the ability to select a group of nodes will be added to Dukas. Because dialogue models can become larger than the display, making it difficult to see the entire model at once, the ability to expand and contract the tree will be included. A separate overview window which shows a smaller view of the entire tree is another enhancement planned for future versions of Dukas.

In addition to enhancements to the look of Dukas, several features are planned which will extend the modeling capabilities of this tool as well. For example, the creation of dialogue models often requires the use of prototypical sub-trees of tasks. In future implementations, Dukas will allow a designer to re-use sub-trees that are part of a library (e.g., the dialogue representing a file dialogue box). The designer will also be able to create and save his own sub-trees into a library. One of the important features enabled by the MASTERMIND approach is the use of on-line "advisors" that will be able to make suggestions to the dialogue modeler by detecting various design patterns (i.e., groups of nodes and orderings) during the design process. In the future, Dukas will be able to make such intelligent suggestions to the designer.

C. Contributions and Future Directions

MASTERMIND is a model based user interface generation system. Its specific contributions include user task oriented design method, a dialogue specification notation from which provably correct implementations are automatically generated and an ontology for binding presentation, application, and dialogue components. We are developing code generators for our models and integrating them with the presentation model code generator developed at ISI. We are also enhancing our dialogue modeling tool, Dukas.

Our immediate concern is to complete works on the MM code generators and to explore the other models described above, particularly the context model. Once this is done, we will be able to claim that we can generate complete user interfaces. At this point we can begin exploring some of the possibilities raised by having models available. For example, having models available at design time enables the use of design critics which can suggest improvements to a design or enforce style rules. Runtime models enable truly powerful context sensitive help mechanisms that can have some hope of understanding what it is that the end user was trying to accomplish with a user interface. Finally, we would like to explore interesting application areas such as the generation of internet web applications. In particular, we would like to replace HTML with Java as a presentation engine in the internet application demonstration we described above.

V. Bibliography

[Balzert96] H. Balzert, F. Hofmann, V. Kruschinski, C. Niemann: The Janus Application Development Environment Generating More than the User Interface. In: J. Vanderdonckt (ed.): Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 183-205.

[Bodart95] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, B. Sacre, J. Vanderdonckt: Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide. In P. Palanque, R. Bastide (eds.): Design, Specification and Verification of Interactive Systems. Wien: Springer, 1995, 262-278.

[Diaper89] Dan Diaper (ed.). Task Analysis for Human-Computer Interaction. Ellis-Horwood, 1989.

[Foley95a] J. Foley, P. Sukaviriya: History, Results, and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. In: F. Paterno (Hg.): Interactive Systems: Design, Specification and Verification. Berlin: Springer, 1995, 3-14.

[Foley95b] Computer Graphics: Principles and Practice. James D. Foley, Andvers van Dam. Addison-Wesley, 1995. Reeding: Mass.

[Gruber] Tom Gruber. *What is an Ontology?* http://www-ksl.stanford.edu/kst/what-is-an-ontology.html.

[Janssen93] C. Janssen, A. Weisbecker, J. Ziegler: Generating User Interfaces from Data Models and Dialogue Net Specifications. In: S. Ashlund, et.al. (eds.): Bridges between Worlds. Proceedings InterCHI'93 (Amsterdam, April 1993). New York: ACM Press, 1993, 418-423.

[McMillan92] K. L. McMillan. Symbolic Model Checking : An Approach to the State Explosion Problem. Ph.D. Thesis, Carnegie-Mellon University. 1992. Tech report # CMU-CS-92-131.

[Myers90a] Brad A. Myers. A New Model for Handling Input. In ACM Transactions on Information Systems, volume 8 number 3 ,1990, 289-320.

[Myers90b] Brad A. Myers, et. al. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. IEEE Computer volume 23 number 11 (November, 1990), 71-85.

[Myers92] Brad A. Myers, Dario Giuse, Brad Vander Zamiten. Declarative Programming in a Prototype Instance System : Object Oriented Programming Without Writing Methods. In Proceedings of the ACM Conference on Object Oriented Programming: Systems, Languages, and Applications OOPSLA'92. (1992).

[Myers95] B. Myers, R McDaniel, A Mickish, and A Klimovitski. The Design for the Amulet User Interface Toolkit. Human Computing Interaction Consortium. 1995.

[Myers96] Brad A. Myers, Alan Ferrency, Rich McDaniel, Robert C. Miller, Patrick Doane, Andy Mickish, and Alex Klimovitski. The Amulet V2.0 Reference Manual. Carnegie-Mellon School of Computer Science Tech Report # CMU-CS-95-166-R1. February, 1996.

[OMG91] Common Object Request Broker Architecture and Specification: OMG Document Number 91.12.1

[Puerta94] A. Puerta, H. Eriksson, J. Gennari, M. Musen: Beyond Data Models for Automated User Interface Generation. In: G. Cockton, S. Draper, G. Weir (eds.): People and Computers IX. Proceedings British HCI'94 (Glasgow UK, August 1994). Cambridge: Cambridge University Press, 1994, 353-366.

[Puerta96] A. Puerta: The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In: J. Vanderdonckt (ed.): Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 19-36.

[Schlungbaum96] E. Schlungbaum, T. Elwert: Automatic User Interface Generation from Declarative Models. In: J. Vanderdonckt (ed.): Computer-Aided Design of User Interfaces. Namur: Presses Universitaires de Namur, 1996, 3-18.

[Shneiderman87] Ben Shneiderman. Designing the User Interface: Strategies for Effective Human-Computer Interaction.

[Szekely93] P. Szekely, P. Luo, R. Neches: Beyond Interface Builders: Model-Based Interface Tools. In: S. Ashlund, et.al. (eds.): Bridges between Worlds. Proceedings InterCHI'93 (Amsterdam, April 1993). New York: ACM Press, 1993, 383-390.

[Szekely96] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, E. Salcher: Declarative interface models for user interface construction tools: the MASTERMIND approach. In: L. Bass, C. Unger (Eds.): Engineering for Human-Computer Interaction. Proceedings of the IFIP TC2/WG2.7 working conference on engineering for human-computer interaction (Yellowstone Park, August 1995). London: Chapman & Hall, 1996, 120-150.