MASTERMIND Project Final Report

Spencer Rugaber
Graphics, Visualization, and Usability Center
Georgia Institute of Technology
November 25, 1998

I. Background and Motivation

The MASTERMIND research project is concerned with the automatic generation of user-interface software from declarative models. ("MASTERMIND" is an acronym that stands for "Models Allowing Shared Tools and Explicit Representations to Make Interfaces Natural to Develop".) The project is a collaboration of researchers in the Graphics, Visualization, and Usability Center at the Georgia Institute of Technology and in the Information Science Institute (ISI) at the University of Southern California. At Georgia Tech, project participants included Spencer Rugaber (Principal Investigator), Piyawadee Sukaviriya, Jim Foley, Kurt Stirewalt, Thomas Browne, Jeyakumar Muthukumarasamy, and David Davila. At ISI, participants included Pedro Szekely (Principal Investigator), Martin Frank, Robert Neches, Pablo Castells, Ewald Salcher, and Ping Luo. The project was funded by the Human Systems Interface program of the Defense Advanced Research Projects Agency, SUN Microsystems, and Digital Equipment Corporation.

Graphical user interfaces (GUIs) are difficult to construct and, consequently, suffer from high costs. The research question that MASTERMIND seeks to address is the extent to which automatic generation of GUIs from declarative descriptions can reduce costs and enforce design principles. Furthermore, if the declarative descriptions are broken into separate components, called models, modularity of the specification is improved, and additional, beneficial features such as design critics, context-sensitive help, and dynamic reconfiguration are enabled.

MASTERMIND supports user-interface designers. A *designer* is someone who specifies or constructs user interfaces using MASTERMIND. A designer's job is to design a user interface for an underlying software system. The designer may or may not participate in the design of the software system itself. 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.

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. A *model* is a declarative specification of the structure and behavior of a software component. Models are declarative because they do not contain procedural code (e.g., C++ code). 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.

II. Project Accomplishments

A. A Conceptual Architecture for the Automatic Generation of User Interfaces from Declarative 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. 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 first-class model. Nevertheless, it serves an essential role in the overall design of the user interface.

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. 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. The model also includes the relative orderings of the activities and the resulting effects they have on the presentation and the application. An *Interaction Model* (IM) specifies the set of possible low-level interactions between the end user and the MASTERMIND run-time environment. It is determined by the underlying toolkit technology upon which MASTERMIND code is generated. Moreover, an interaction's primary value is in specifying communication from the end user to the other components of MASTERMIND. We think of this communication as being tight in the sense that efficiency concerns are paramount. Specifically, we want to take advantage of underlying toolkit interactors to implement the interaction. And the interaction is atomic in the sense that no MASTERMIND dialogue reasoning is used to affect the interaction. The fourth component that we use is called the *Application Wrapper* (AW). It specifies the interface between MASTERMIND and application functionality. The Wrapper is specified using IDL, the notation used for describing programming-language independent interfaces to CORBA objects. The Application Wrapper specification includes method signatures and data item declarations for features that the application makes accessible to the user interface. Because names and types are specified but semantics are not, the AW does not qualify as a MASTERMIND model. Nevertheless, it provides an essential piece of the declarative description from which a user interface is generated.

B. A Language for Modeling User-Application Dialogue

The MASTERMIND Dialogue Notation (MDL) allows designers to specify the permissible ordering of the end-user interactions that carry out some task. Although the notation is textual, a design tool (Dukas, described below) supports the graphical entry and editing of models. MDL allows designers to organize end-user interactions using a declarative mechanism similar to production rules in context-free grammars. MDL supports five kinds of interaction orderings, each denoted by an operator: **alt** (alternative), **dis** (disabling), **seq** (sequential), **par** (parallel), **excl** (exclusive execution), and **int** (interrupt/resume). The notation also provides operators for expressing preemption. In particular, four operators are available: **opt** denoting optional, user-specified preemption and **cond** indicating optional, application-dependent preemption. The same two kinds of preemption are provided for modeling recurring situations: **loop** denoting user choice, and **while** indication application choice. An MDL code generator can then convert a dialogue model into C++ classes, which, when linked with the MASTERMIND run-time libraries, produces an executable capable of controlling the interaction of the end user with the user interface and the application.

C. The MASTERMIND Interaction Model

Presentation and Dialogue Models do not directly model end-user input. MASTERMIND defines input in a separate Interaction Model. Interaction techniques implement the hardware-binding portion of user interface design. Such interaction techniques are normally provided by an existing toolkit such as Amulet, Java AWT, or X-MOTIF. MASTERMIND interactions abstract this interface 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 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. Interaction techniques represent natural dialogue elements because they are logical end-user/computer exchanges, and, due to mutual demands upon shared resources, because 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 dragging and dropping one presentation object over another. MASTERMIND considers this one logical interaction. It cannot be interleaved with, for example, clicking a button because during one of these interactions the mouse is being used and cannot be relinquished temporarily to complete the other. Interactions encapsulate access to shared resources into atomic interactions that can be ordered according to the Dialogue Model. These interactions are complicated by the tight web of resource interdependencies that govern an interaction. A specific interaction might involve feedback to the end user, a commitment to a specific dialogue action, 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.

Because interaction techniques naturally capture logical end-user interactions, and because 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.

D. A User Interface Code Generation Process

MASTERMIND tools synthesize user interfaces from three distinct models of UI behavior. To go from models to executable code, MASTERMIND employs model-specific code generators. Each code generator outputs C++ source files which are then compiled and linked with a number of run-time libraries. Because there are necessarily dependencies among models, model instances must refer to elements of other model instances. By focusing attention on a single aspect of a user interface, a model can be expressed in a highly-specialized notation. Model-specific compilers generate modules of code from each model, and these resulting modules are composed. A distinguishing characteristic of MASTERMIND is that the model-specific code generators work independently of one another.

Composing the code generated from multiple models is difficult. A model, by design, represents some aspects of a system and is neutral with respect to others. Inevitably, however, functionality described in one model overlaps with or is dependent upon functionality described in another. A button, for example, is specified in the presentation model, but the behavior of the button influences behavior in other models, such as when pressing the button causes other widgets to be enabled or disabled. The effect of pressing a button might also cause some application method to be invoked. When code generated from multiple models must cooperate, these redundancies and dependencies can be difficult to resolve. Resolving them automatically means that behavior in different models must be unified and the mechanism for this unification must be implemented efficiently. Model-based approaches to user-interface generation use models that are specified in diverse and often incompatible notations. This characteristic complicates model composition because the composition mechanisms in one model may not exist in another.

One characteristic of model-based approaches is that, by restricting the focus of a model to a single attribute of the system, modeling notations can be specialized and highly declarative. The MASTERMIND Presentation Model, for example, combines concepts and terminology from graphic design with mechanisms for composing complex presentations through functional constraints. Dialogue models use state and event constructs to describe the user-computer conversation. Example notations include StateCharts and Petri nets, which use a variety of composition mechanisms that include state hierarchy, concurrency, and communication. The MASTERMIND Application Model combines concepts and terminology from object-oriented design techniques with mechanisms for composing complex behavior based on method invocation. These examples illustrate that composition mechanisms in one model may not exist in another model.

Our approach to model composition is based on prior work in multi-agent user-interface architectures, which provide design heuristics for structuring an interactive system. These architectures describe interactive systems as collections of independent communicating agents, which are computational units with identity and behavior. The MASTERMIND Presentation, Application, and Dialogue models are descriptions of the roles to be played by presentation, computational, and controller agents. Specifically, the presentation and application models define actions, which are ordered by temporal constraints in the dialogue model.

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. Generated applications are cast onto the run-time architecture. Dialogue control is separated from presentation and application functionality. Dialogue, presentation and interactor agents may directly observe the application state, but only the dialogue component may effect a change in the application. The MASTERMIND architecture rejects a rigid boundary between dialogue and presentation in favor of a more fluid boundary, constituted by interactor objects. Interactor objects encapsulate tight protocols of dialogue state, application state, user input, and presentation feedback.

E. Application of Model Checking

We applied automated model checking to detect flaws in the MDL library design. The architecture of the library allows a programmer to declaratively create user-interfaces with complex temporal behavior (e.g., optionality, interruption and termination). The key to making the library declarative is to provide a set of reusable objects that impose hierarchical orderings over widgets (e.g., buttons, sliders, and text-entry boxes). The base set of widgets--that is, the widgets that are combined using the orderings--come from the Amulet user-interface library. The MASTERMIND library extends the Amulet widgets by providing a set of ordering constructors that aggregate and impose complex temporal orderings upon the underlying Amulet widgets. This design decision simplified the composition of widgets, but it resulted in complex communication protocols in the constructors.

We used SMV (the specification language of the symbolic model checker from Carnegie-Mellon University) as the input language to describe the behavior of each constructor and then ran the model checker to discover flaws in these designs. Using the model checker, we found several design flaws, which were subsequently fixed. Since then, there have been no errors discovered in the library.

F. The MASTERMIND Design Method

Building a user interface in MASTERMIND means constructing models. Our current understanding forms the basis for a model-based, user-interface design method. 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 rather from design artifacts. These are the Presentation Model, the Dialogue Model, and the Application Wrapper. These components are 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.

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 among 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 Model. Because 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.

G. Publications

1. Book Chapter

The MASTERMIND project contributed a refereed chapter to the book *Formal Methods in Human Computer Interaction* published last year. Specific citation information is given by the following.

Thomas Browne, David Davila, Spencer Rugaber, and Kurt Stirewalt.
"Using Declarative Descriptions to Model User Interfaces with MASTERMIND."

In *Formal Methods in Human Computer Interaction,* Fabio Paterno and Philippe Palanque, editors, Springer-Verlag, 1997.

2. Thesis

Kurt Stirewalt, a Graduate Research Assistant on the MASTERMIND project, wrote a PhD dissertation describing research that he performed on the project. The dissertation is titled *Automatic Generation of Interactive Systems for Declarative Models.* Dr. Stirewalt is now an Assistant Professor at Michigan State University. The abstract for his dissertation is the following.

> This dissertation applies formal methods to the automatic generation of interactive systems from multiple declarative models. We are interested in two kinds of models: user-task and presentation. The MM Dialogue Language (MDL) is presented. MDL is a notation for describing interactive system behavior in terms of user tasks. MDL task models are expressed independently of other models, like presentation, but are later composed with the behavior of these other models. Such separation is important for preserving the integrity of models over the lifetime of a system. The technical challenge in this approach is to generate code that combines the functionality of task and presentation models without violating that integrity. To meet this challenge we implement MDL task models as run-time dialogue constraints that synchronize with presentation components. The constraint engine is implemented as a virtual machine that simulates the execution of tasks and resolves the dependencies that arise as a result of task and presentation model composition. To simplify the generation process, a toolkit of reusable run-time components is provided. Each component in this toolkit implements an MDL operator, and components aggregate into trees whose structure corresponds one-to-one with the abstract syntax tree of a corresponding MDL model. Thus implementations can be generated through a simple syntactic transformation of the MDL source code. The design correctness of these components is validated by a novel application of symbolic model checking. The run-time attributes of systems generated using this approach are measured, and we conclude that this strategy of model composition is feasible for use with real interactive systems.

3. Bibliography of Papers

• P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. "Declarative interface models for user interface construction tools: the MASTERMIND approach." In: Bass L., Unger C. (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, Chapman & Hall, 1996, pp. 120-150.

• R. Neches, J. Foley, P. Szekely, P. Sukaviriya, P. Luo, S. Kovacevic, and S. Hudson. "Knowledgeable Development Environments Using Shared Design Models." *Proceedings IWIUI'93,* Orlando Florida, January 1993, ACM Press, pp. 63-70.

• P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. "Declarative Interface Models for User Interface Construction Tools: The MASTERMIND Approach." *Proceedings EHCI'95.*

• K. Stirewalt and S. Rugaber. "Automating UI generation by model composition." *Proceedings of Automated Software Engineering (ASE'98) Thirteenth International Conference.* 1998.

H. Tools

The MASTERMIND project has built a series of prototype user-interface generation tools. A designer specifies the features of an interface, including its dialogue, presentation, and application interface, and a program is automatically generated that implements that interface. The MASTERMIND environment is intended to be both a user interface design platform and a delivery platform where run-time services are provided based on the models. In the design environment, we have built a graphical design tool where models can be described to the system using a visual representation, bottom-up drag-and-drop from palette to the design space. We have also built the MASTERMIND run-time environment which can automatically generate interactive help from the model. The environment was designed to support distributed applications using CORBA technology.

One of the tools developed is the Dukas Graphical Task Modeling tool. The Dukas tool is designed to assist an application designer in creating a task model of their system and using that model to assist in the coding of an application and its interface. Dukas allows designers to develop a graphical tree describing system behavior. This tree is defined in terms of low-level tasks (leaves) and behavioral operators between nodes. These operators are based on the MASTERMIND Dialogue Language notation. Dukas stores this representation in a grammatically simple text format, which can be used by the designer. Used by itself, Dukas can provide a designer with a tool to specify, study and refine a model of the system behavior before implementation begins. The graphical representation also presents an easy tool for collaboration and discussion of system behavior in a development team environment. Using other MASTERMIND system components, Dukas representation supports the automatic generation of C++ code to execute the task model and application. This provides Dukas with the ability to not only work as a tool for assisting analysis and design process, but allows for rapid prototyping of designs and easy maintainability of a complex application system.

K. AASERT

The MASTERMIND project served as the base grant under which AASERT funds were received. Those funds supported the training of Thomas Browne, a graduate student in the GVU Center. Thomas was responsible for enhancing the Dukas dialogue modeling tool.

III. Further Information

The MASTERMIND project home page can be found at URL http://www.cc.gatech.edu/gvu/user_interfaces/Mastermind/.