, , 1–22 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

The model-composition problem in user-interface generation

R. E. KURT STIREWALT stirewalt@cse.msu.edu
Dept. of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824

SPENCER RUGABER spencer@cc.gatech.edu
College of Computing, Georgia Institute of Technology, Atlanta, GA 30032

Abstract. Automated user-interface generation environments have been criticized for their failure to deliver rich and powerful interactive applications [22]. To specify more powerful systems, designers require multiple specialized modeling notations [15, 17]. The model-composition problem is concerned with automatically synthesizing powerful, correct, and efficient user interfaces from multiple models specified in different notations. Solutions to the model composition problem must balance the advantages of separating code generation into specialized code generators each able to take advantage of deep, model-specific knowledge against the correctness and efficiency obstacles that result from such separation. We present a correct and efficient solution that maximizes the advantage of separation by using run-time composition mechanisms.

Keywords: Model-based, user interface, code generation, multi-paradigm

1. Introduction

Building user interfaces (UIs) is time consuming and costly. In systems with graphical UIs (GUIs), nearly 50% of source code lines and development time can be attributed to the UI [14]. GUIs are usually built from a fixed set of modules composed in regular ways. Hence, GUI construction is a natural target for automation. Automated tools have been successful in supporting the presentation aspect of GUI functionality, but they provide only limited support for specifying behavior and the interaction of the UI with underlying application functionality. The *model-based* approach to interactive system development addresses this deficiency by decomposing UI design into the construction of separate models, each of which is declaratively specified [5]. Once specified, automated tools integrate the models and generate an efficient system from them. The *model-composition problem* is the need to efficiently implement and automatically integrate interactive software specified in separate, declarative models. This paper introduces the model-composition problem and presents a solution.

A *model* is a declarative specification of some single coherent aspect of a user interface, such as its appearance or how it interfaces to and interacts with the underlying application functionality. By focusing attention on a single aspect of a user interface, a model can be expressed in a highly-specialized notation. This property makes systems developed using the model-based approach easier to build and maintain than systems produced using other approaches [23].

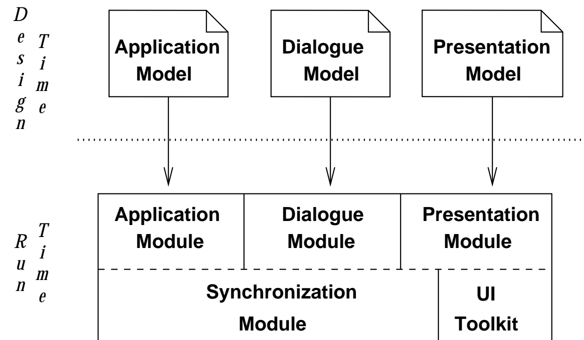

Figure 1. Model-based code generation

The MASTERMIND project [5, 15] is concerned with the automatic generation of user interfaces from three kinds of models: *Presentation models* represent the appearance of user interfaces in terms of their widgets and how the widgets behave; *Application models* represent which parts (functions and data) of applications are accessible from the user interface; and *Dialog models* represent end-user interactions, how they are ordered, and how they affect the presentation and the application. A dialog model acts as the glue between presentation and application models by expressing constraints on the sequencing of behavior in those models. Model-specific compilers generate *modules* of code from each model, and these resulting modules are composed into a complete user interface (**Figure 1**). A distinguishing characteristic of MASTERMIND is that the model-specific code generators work independently of one another.

Composing code generated from multiple models is difficult. A model, by design, represents a single aspect of a system and is neutral with respect to others [3]. Inevitably, however, functionality described in one model overlaps with or is dependent upon functionality described in another. A button, for example, is specified in a 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. Such effects are described in a dialog model. The effect of pressing a button can also cause an application method to be invoked. Such effects are described in an application model. 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 correctly unified, and the mechanism for this unification must be implemented efficiently.

The model-composition problem is concerned with automatically synthesizing powerful, correct, and efficient user-interfaces from separate presentation, dialog, and application models. We present a two-fold solution. First, we formalize the three models as concurrent agents, which synchronize on common events (**Section 3**). Second, we present a runtime architecture that supports the composition of modules generated from independent model compilers (**Section 4**). We present

the results of this approach on two examples and give evidence to show that it scales up (**Section 5**).

2. Background

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 (**Section 2.1**). Prior research on the architecture of user-interfaces suggests using communicating agents to structure user-interface code (**Section 2.2**). Formal models of communicating agents provide a technique called conjunction, which is useful for composing partial specifications of a system (**Section 2.5**). The contribution of this paper is an extension of conjunction as a specification-composition operator into a runtime-composition mechanism.

2.1. Model-based generation

The model-based approach to interactive system development expresses system analysis, design, and implementation in terms of an integrated collection of models. Unlike conventional software engineering, in which designers compose software documentation whose meaning and relevance can diverge from that of the delivered code, in the model-based approach, designers build models of critical system attributes and then analyze, refine, and synthesize these models into running systems. Model-based UI generation works on the premise that development and support environments may be built around declarative models of a system. Developers using this approach build interfaces by specifying models that describe the desired interface, rather than writing a program that exhibits the behavior [21].

One characteristic of model-based approaches is that, by restricting the focus of a model to a single aspect of a system, modeling notations can be specialized and highly declarative. The MASTERMIND Presentation Model [6], for example, combines concepts and terminology from graphic design with mechanisms for describing complex presentations using functional constraints. The MASTERMIND Dialog Model [19] uses state and event constructs to describe the user-computer conversation; the composition features include state hierarchy, concurrency, and communication. The MASTERMIND Application Model combines concepts and terminology from object-oriented design techniques [18] with mechanisms for composing complex behavior based on method invocation.

Figure 2 compares the MASTERMIND models in terms of their domains of discourse, communication mechanisms, runtime components, and how they are composed. Composition mechanisms in one model may not exist in another model. No single one of these intra-model mechanisms is sufficient for composing all three MASTERMIND models. The model-composition problem can be restated as the need to unify behavior in multiple models without violating the rules of intra-model composition and while generating efficient code. The model-composition problem is a declarative instance of the problem of constructing a software system where the ma-

Module	Process Implementation	Action Implementation	Intra-module Composition
Application	Abstract Classes	Method Invocation	Subclassing Aggregation
Presentation	Amulet Objects	Constraints, Commands	Instantiation, Aggregation
Dialog	State Machines	Synchronous Message passing	Orthogonal Composition

Figure 2. Multi-paradigm action implementations

jor components are expressed with programming languages from different families or paradigms. Zave has called this the multi-paradigm programming problem [24].

2.2. Multi-agent user-interface architectures

The MASTERMIND approach to model composition builds on prior work in multi-agent user-interface architectures, which provide design heuristics for structuring interactive systems. These architectures describe interactive systems as collections of communicating *agents*, which are independent computational units with identity and behavior. Two general frameworks—Model-View-Controller (MVC) [11] and Presentation-Abstraction-Control (PAC) [7]—define specific agent roles and provide guidance on how agents should be connected.

MVC prescribes how SmallTalk simulations can be composed by instantiating instances of three types of agents: *models* (not to be confused with the MASTERIMIND models) describing application state, *views* providing presentations of models, and *controllers* allowing users to affect simulation behavior. A view registers interest in one or more attributes of a model. When an attribute changes, all registered views are notified so that they can recompute their display if necessary.

The PAC framework more closely matches MASTERMIND than does MVC. In PAC, a *presentation* agent maintains the state of the display and accepts input from the user, an *abstraction* agent maintains a representation of the underlying application state, and a *controller* agent ensures that presentation and abstraction remain synchronized. The MASTERMIND Presentation, Application, and Dialog models are descriptions of the roles played by PAC's presentation, abstraction, and controller agents.

Since MASTERMIND models describe PAC agents, we chose to make MASTERMIND models compose in the same manner that PAC agents compose. Specifically, the presentation and application models define actions, which are ordered by temporal constraints in the dialog model. To make these ideas more formal, we built upon prior work on formal definitions of agent composition.

2.3. Formal models of agents

The PAC framework provides heuristic definitions of user-interface agent roles and connections. PAC agents are concurrent, and they compose by communicating control and data messages among themselves. To generate code from the models of these agents, we need to formalize the building blocks of agents and agent composition. We chose the terminology and definitions that have been adopted by the various process algebras, specifically LOTOS [4]. Process algebras formalize concurrency and communication, and they have proved particularly useful for describing UI software as a collection of agents [1, 2]. Other notations, such as StateCharts [8] and Petri nets [16], have also been explored for modeling UI agents, as these alternative notations also provide definitions of concurrency and communication. We chose LOTOS because composition in LOTOS resembles *conjunction* [25], which is a useful paradigm for composing partial specifications (**Section 2.5**).

We model the behavior of an agent using a LOTOS abstraction called a *process*, which is a computational entity whose internal structure can only be discovered by observing how it interacts with its environment. Processes perform internal (unobservable) computations and interact with other, concurrently executing, processes. The interaction between processes is synchronous: If one process tries to communicate with a process that is not ready to communicate, the former process blocks until the latter is ready. Thus, the act of communicating synchronizes concurrent processes.

A process represents the state of an agent as a procedure for performing future actions. An *action* is an atomic computational step taken by an individual process. Actions of a process can be observed through the events in which the actions participate. An *event* is an observable unit of multi-process communication. Multiple processes participate in an event by simultaneously performing actions over the same gate. A *gate* is a primitive synchronization device used to observe the occurrence of an action in a process. Each action is associated with a single gate. The gates of a composite agent are the union of the gates of its constituents. If two or more constituents name the same gate, then any actions over that gate proceed simultaneously. That is, the processes associated with the constituent agents synchronize actions that share the same gate name. Thus, gates also represent a class of possible inter-process synchronization events. During such an event, an action can *offer* one or more data values that can be *observed* by actions in other processes that are participating in the same event.

A complete agent is modeled by a process that represents the initial state of the agent. A multi-agent system is modeled by a collection of concurrent, communicating processes. When composing a system of multiple agents, the designer must decide how to coordinate actions in the various processes that model the agents. Processes are coordinated by synchronizing actions labeled with identically named gates.

2.4. Lotos

LOTOS is a rich language for specifying the partial ordering of actions within a process and the structure of multi-process interactions. Complex processes may be expressed by either combining sub-processes using an ordering operator (e.g., process P is the sequential composition of sub-processes P_1 and P_2) or by conjoining sub-processes so that they run independently but synchronize actions with gates. An event allows values to flow between participating actions. LOTOS also describes the semantics of value passing with respect to synchronization.

Actions in LOTOS have the following structure:

$$\begin{aligned} action &::= gate \ \ (input|output) \ * \\ input &::= \ '?' \ \ identifier \ \ ':' \ \ type \\ output &::= \ '!' \ \ expression \\ gate &::= identifier \end{aligned}$$

Each action names a gate and zero or more inputs and outputs. An input names a variable in which to record a value that is offered by an action in another process. An output is an expression for computing a value to offer to actions in other processes.

Actions concisely represent the occurrence of many possible events. Like actions, events are associated with a particular gate. Unlike actions, events have no concept of input or output; rather they represent unique values that flow between actions. Events have the following structure:

$$\begin{aligned} event &::= gate \ \ (value) \ * \\ value &::= \ '!' \ \ constant \end{aligned}$$

Note that the values are always constants because events are unique assignments of values during a synchronization.

In LOTOS, the gates over which two conjoined processes are required to synchronize must be specified between the vertical lines that symbolize the conjunction operator ($||$). For example, given the following LOTOS process definitions:

$$\begin{aligned} &\textbf{process } P \ [\ g_1, g_2, g_3\ ] \ldots \textbf{endproc} \\ &\textbf{process } Q \ [\ g_1, g_2, g_4\ ] \ldots \textbf{endproc} \\ &\textbf{process } R \ [\ g_1, g_2, g_3, g_4\ ] := P \ |[g_1, g_2]| \ Q \quad \textbf{endproc} \end{aligned}$$

Process R behaves like P on gate g_3 and Q on gate g_4, but R must behave like P and Q in synchrony on gates g_1 and g_2.

For processes with many gates, the LOTOS notation quickly becomes unreadable. In this paper, we abbreviate the conjunction operator using notational conventions similar to those used in CSP [9]. In our abbreviated notation, we write the conjunction of P and Q as $P \parallel Q$ with the understanding that P and Q must synchronize on gates that are common to the agents whose states P and Q respectively represent.

Suppose the behavior of an agent can be described by a LOTOS process B. If the agent can perform an action by synchronizing on event e (denoted $B(e)$), then its

behavior from that point on is defined by another process $B' = B(e)$. The systems under study are deterministic, which means that $B(e)$ is always unique. Moreover, when a system is defined by conjoining sub-processes, the compositional structure is preserved throughout the lifetime of the system. That is, if $B = B_1 \parallel B_2 \parallel \cdots \parallel B_n$ then $B(e) = B'_1 \parallel \ldots \parallel B'_n$ where:

$$B'_i = \begin{cases} B_i(e) & \text{if } e \text{ occurs over a gate of agent } i \\ B_i & \text{otherwise} \end{cases}$$

Any event that can be observed of a process P can also be observed of any conjunction of P with other processes. This fact will be important when we define the *Obs* observer function (**Section 3.4**).

2.5. Conjunction as composition

Alexander uses conjunction to compose separately defined application and presentation agents [2]. Abowd uses agent-based separation to illuminate usability properties of interactive systems [1]. Both of these approaches rely on the use of conjunction to compose agents that are defined separately but interact. In fact, conjunction is a general operator for composing partial specifications of a system [25]. The idea is that each partial specification imposes constraints upon variables (or, in the case of agents, events) that are mentioned in other partial specifications. When these specifications are conjoined, the common variables must satisfy all constraints.

We define the behavior of a system generated from MASTERMIND models to be any behavior that is consistent with the conjunction of constraints imposed by the dialog, presentation, and application models. We then extend conjunction from a specification tool into a mechanism for composing runtime modules.

2.6. Summary

Three issues must be addressed to solve the model-composition problem: The solution must generate user-interfaces with rich dynamic behavior, the correctness of module composition must be demonstrated, and the generated modules must cooperate efficiently. In MASTERMIND, the rich expressive power is achieved through special-purpose modeling notations [15, 5]. The remainder of this paper addresses the generation of correct implementations with maximal efficiency while preserving the expressive power of MASTERMIND models.

3. Model-composition theory

Recall from **Figure 1** that each class of model has a code generator that synthesizes runtime modules for models in that class. The modules are generated without detailed knowledge of the other models. At run time, however, modules must

cooperate as prescribed by the conjunction of the models that generated them. This section describes the relationship between model composition and the mechanism by which the associated modules cooperate at runtime.

3.1. Notation

The subject of this paper is the automatic generation and composition of runtime modules from design-time models. A *module* is a unit of code generated from a single model. We use a third class of construct—the LOTOS process—to define the correctness of model and module composition. In formal arguments, we need to refer to all three types of constructs; thus we distinguish the constructs by using different fonts. We also need special functions that map models and modules into comparable domains.

We represent the classes of MASTERMIND models using German letters. The symbols $\mathfrak{P}$, $\mathfrak{D}$, and $\mathfrak{A}$ represent respectively the classes of MASTERMIND presentation, dialog, and application models. We use the italic font to represent LOTOS processes and the semantic models of these processes. The set $Process$ represents the set of LOTOS processes. Specific processes are written in capital italic letters (e.g., P, D, and A, respectively). The set $TraceSets$ defines the set of event traces over the alphabet of gates and the space of values that can be offered and observed by LOTOS actions. The function $Tr : Process \rightarrow TraceSets$ maps a LOTOS process to the set of all event traces that can be observed of that process.

We represent runtime entities using the Sans serif font. The set $\mathsf{Component}$ represents the class of all runtime components. A *component* is a block of code that provides gates for observing the actions of the component. By defining components as runtime code that provides gates for observing behavior, we can define the function $Obs : \mathsf{Component} \rightarrow TraceSets$ that maps a component to the set of event traces that can be observed through the gates that the component provides.

There are two categories of component in the MASTERMIND architecture: the generated modules and the synchronous composition of these modules. Instances of the generated modules are written Pres, Dialog, and Appl, respectively. We also think of the modules in synchronous composition as a component, which is attained by connecting the generated modules using some synchronization infra-structure (defined in **Section 4**). This composite component is written $\mathsf{Synch}[\mathsf{Pres}, \mathsf{Dialog}, \mathsf{Appl}]$. The name Synch suggests that the component is the synchronization of the three generated modules; the brackets suggest that the generated modules fit into the larger system and that Synch by itself is not a component.

3.2. Inter-model composition

Model-based code generators construct runtime modules from design-time models. The code generation strategy is model-specific, reflecting the specialization of models to a particular aspect of a system. At run time, however, modules must cooperate, and the cooperative behavior must not violate any correctness constraints imposed by the models. There is an inherent distinction between behavior that

is limited to the confines of a given model and behavior that affects or is affected by other models. Inter-model composition is concerned with managing this latter inter-model behavior.

Some behavior is highly model specific and neither influences nor is affected by behavior specified in other models. As **Figure 2** illustrates, in a MASTERMIND presentation model, graphical objects are implemented using primitives from the Amulet toolkit [13], and attribute relations are implemented as declarative formulas that, at runtime, eagerly propagate attribute changes to dependent attributes. As long as changes in these attributes do not trigger behavior in dialog or application models, these aspects can be ignored when considering model composition.

In an application model, object specifications are compiled into abstract classes under the assumption that the designer will later extend these into subclasses and provide implementations for the abstract methods. As long as the details of these extensions do not trigger behavior in dialog or presentation models, this application behavior may also be ignored when defining model composition.

Within a module, entities compose according to a model-specific policy. In a presentation model, for example, objects compose by part-whole aggregation, and attributes compose by formula evaluation over dependent attributes. In an application model, objects compose using a combination of subclassing, aggregation, and polymorphism. When considering how models compose, some details of *intra-model* composition can be abstracted away, but not all of them. Models impose temporal sequencing constraints on the occurrence of inter-model actions, and models contribute to the values computed by the entire system. These constraints and contributions must be captured in some form and used to reason about model composition.

We map this *inter-model* behavior into a semantic domain that is common across all of the models. This domain is described by the LOTOS notation, which specifies temporal constraints on actions and data values. We assume that LOTOS processes can be derived from the text of a model specification (**Section 3.4**). Designers may, for example, need to designate actions of interest to other models. LOTOS processes do not capture all of the behavior of models in composition, but they do express the essential inter-model constraining behavior.

3.3. Example

We now present an example of inter-model behavior expressed as a LOTOS process. The dialog model being considered is for a Print/Save widget similar to those found in the user interfaces of drawing tools, web browsers, and word processors (See **Figure 3**). These widgets allow the user to format a document for printing either to a physical printer or to a file on disk; we call the former task *printing* and the latter task *saving*. Options specific to printing, such as print orientation (e.g., portrait vs. landscape), and to saving, such as the name of the file into which to save, are typically enabled and disabled depending upon the user's choice of task. These ordering dependencies are reflected in the dialog model for this widget shown by the LOTOS process in **Figure 4**.

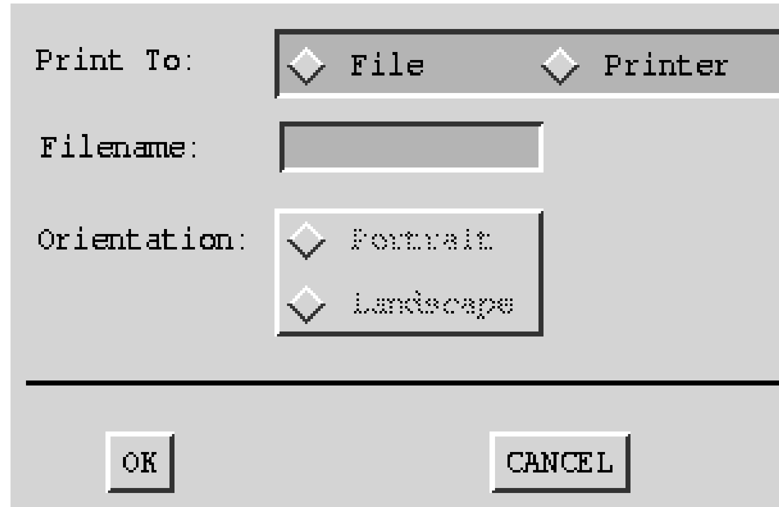

Figure 3. Screen shot of the Print/Save dialog box

The process $PrintSave$ can synchronize on any of the gates that follow it in square brackets. In this example, the gates *print*, *save*, *go*, *cancel*, *layout*, and *kbd* (line 1 in the figure) define points for synchronizing with the presentation; whereas the gates *lpr* and *write* define points for synchronizing with the underlying application. The process parameters *lpdhost* and $filename$ (line 2) store the name of the default printer and the user-selected filename, respectively. The parameter *doc* represents the document to be printed or saved, and the parameter *port* represents the print orientation (portrait if true, landscape if false).

The widget in **Figure 3** is specified by a separate presentation model (not shown). This model defines a pair of radio buttons labeled **File** and **Printer** and two buttons labeled **OK**, and **CANCEL**. When these buttons are pressed, they offer the events *save*, *print*, *go*, and *cancel* respectively. The presentation model also contains a pair of radio buttons that specify paper orientation. These buttons display graphics of a page in either portrait or landscape mode and, when selected, offer the event *port* with a value of *true* if the choice is for portrait orientation and $false$ for landscape orientation. Finally, there is a text entry box in which the user can type in a file name. As the user edits this name, the text box responds by offering the contents of the string typed so far as part of the *kbd* event. Note that the actual keys being pressed are not returned, as editing functionality is best handled in a text widget and is not considered inter-model behavior. A separate application model (not shown) defines procedures for issuing a print request and saving a file to disk. These procedures are responsive to the events *lpr* and *write* respectively. Actions that synchronize on these events offer a number of values including printer name (*lpdhost*) and filename ($filename$).

```
 1. process PrintSave[ print, save, go, cancel, layout, kbd, lpr, write ]
 2.      ( lpdhost, filename  : string, doc : doctype, port : bool )  : exit :=
 3.      P[ go, lpr, write, layout, kbd ]  [>  ( cancel; exit )
 4. where
 5.      process P[ go, lpr, write, layout, kbd ]  : exit :=
 6.           Layout[ go, lpr, layout ]  [>  ( save; F[ go, lpr, write, layout, kbd ] )
 7.      endproc
 8.      process F[ go, lpr, write, layout, kbd ]  : exit :=
 9.           Edit[ go, write, kbd ]  [>  ( print; P[ go, lpr, write, layout, kbd ] )
10.      endproc
11.      process Layout[ go, lpr, layout ]  : exit :=
12.           ( layout ? port; Layout[ go, lpr, layout ] )
13.        [] ( go; lpr ! lpdhost ! port ! doc; exit )
14.      endproc
15.      process Edit[ go, write, kbd ]  : exit :=
16.           ( kbd ? filename; Edit[ go,write,kbd ] )
17.        [] ( go; write ! doc ! file; exit ) )
18.      endproc
19. endproc
```

Figure 4. Print/Save dialog process.

The temporal structure of dialog, presentation, and application model composition is given in the behavior specification (line 3). The behavior of *PrintSave* is the behavior of the process *P* (defined on lines 5 through 7) with the caveat that it may be disabled (terminated) at any time by the observation of the *cancel* event. Disabling is shown with the [> operator. Process *P* represents which interactions and application invocations must happen in order to send a document to a printer. Most of this functionality is actually expressed in the sub-process *Layout* (defined on lines 11 through 14). *P* behaves like *Layout* in the normal case, but it can be disabled if the *save* event is observed. Recall that the *save* event is offered whenever the user presses the **Save to File** button in the presentation model. The process *F* (defined on lines 8 through 10) likewise behaves like the process *Edit* (defined on lines 15 through 18) in the normal case, but is disabled if the event *print* is observed. Note that *F* and *P* can disable each other, which means that the user can switch back and forth between printing and saving as many times as he or she likes before hitting the **Go** button.

$$\begin{array}{ccc} \mathfrak{D} & \xrightarrow{\mathcal{A}_D} & Process \\ {\scriptstyle \mathcal{C}_D}\downarrow & & \downarrow{\scriptstyle Tr} \\ \mathsf{Component} & \xrightarrow{Obs} & TraceSets \end{array}$$

Figure 5. Dialog compiler correctness.

3.4. Models, modules, and processes

Processes like those shown in **Figure 4** are useful for understanding the relationship between models and modules. This relationship is complex, and so we describe it first for a single model and then for the three models in composition. We now formalize correctness conditions for the MASTERMIND dialog model. A similar formalization exists for the other MASTERMIND models.

Figure 5 shows the relationship between dialog models (members of the set $\mathfrak{D}$), runtime modules generated by dialog models (members of the set Dialog), and the inter-model behavior of dialog models (members of the set *Process*). The relationships between these sets are defined as functions that map members of one set into members of another. The function $\mathcal{C}_D : \mathfrak{D} \rightarrow \mathsf{Dialog}$ maps dialog models to runtime modules. Think of $\mathcal{C}_D$ as an abstract description of the dialog-model compiler. The function $\mathcal{A}_D : \mathfrak{D} \rightarrow Process$ maps dialog models into LOTOS processes describing their inter-model behavior. Think of $\mathcal{A}_D$ as an abstract interpretation of the dialog model expressing its semantics in LOTOS.

These sets and functions are related by the commutative diagram of **Figure 5**. Externally observable model behavior is mapped into a LOTOS process by $\mathcal{A}_D$, and the set of traces of a module's externally observable events is recorded by *Obs*. We say that a dialog model $d \in \mathfrak{D}$ is consistent with the module $\mathcal{C}_D(d)$ if every trace $\sigma \in Obs(\mathcal{C}_D(d))$ is in the set $Tr(\mathcal{A}_D(d))$ and if there are no sequences $\varsigma \in Tr(\mathcal{A}_D(d))$ such that $\varsigma \notin Obs(\mathcal{C}_D(d))$. That is, the inter-model behavioral interpretation of d agrees exactly with the observable behavior of the runtime module generated from d. Commutativity of the diagram requires this property for any dialog model in the set $\mathfrak{D}$.

3.5. Model-based synthesis

The correctness relationship between models and modules (**Figure 5**) can be extended to specify the correctness of module composition. We now have functions $\mathcal{A}_P$, $\mathcal{A}_D$, and $\mathcal{A}_A$ that map models into LOTOS processes. These processes should compose by conjunction. We also have a runtime component Synch that combines modules Pres, Dialog, and Appl into a single component whose actions are observable by the *Obs* function. **Figure 6** shows the constraints on the behavior of these entities. Let $p \in \mathfrak{P}$, $d \in \mathfrak{D}$, and $a \in \mathfrak{A}$. Then the code generated from these

$$\forall p \in \mathfrak{P} : \ \forall d \in \mathfrak{D} : \ \forall a \in \mathfrak{A}:$$
$$Obs(\mathsf{Synch}[\mathcal{C}_P(p), \mathcal{C}_D(d), \mathcal{C}_A(a)])$$
$$= Tr(\mathcal{A}_P(p) \parallel \mathcal{A}_D(d) \parallel \mathcal{A}_A(a))$$

Figure 6. Module-composition correctness.

models is correct if and only if, for any observable behavior σ, σ is a legal trace in the conjunction of the models. This equation defines the conditions necessary for correct module composition without assuming any model-specific interpretation of these actions. It serves, therefore, as a specification of design requirements. In the next section, we present an implementation that satisfies these requirements.

4. Module-composition runtime architecture

We now turn to the designs of the run-time synchronization module and model-specific compilers of **Figure 1**. The essential design problem is how to make the generated modules compose while retaining the independence of the model-specific compilers. The conditions of **Figure 6** impose constraints on these designs. Fortunately, these constraints do not require model-specific knowledge (e.g., graphical concepts in the presentation model or data layout in the application model). Thus, module-composition logic can be separated from the model-specific functionality within a module. This separation is the key to making model-based synthesis independent without sacrificing the correctness of module integration. The MASTERMIND runtime library contains efficient primitive classes that enable independent module synthesis and correct composition by conjunction. This library provides a great deal of generality and flexibility for code generation. In this paper, we describe only those aspects of the library that are relevant for supporting independent synthesis. First, we introduce the mechanism for composing generated modules (**Section 4.1**). We then describe how this mechanism implements conjunction without sacrificing the independence of model synthesis (**Section 4.2**) and demonstrate its operation through an example (**Section 4.3**).

4.1. Design structures to support conjunction

To facilitate the independence of model synthesis, we designed a mechanism that enables a module to compose with other modules without directly referencing them. As **Figure 1** suggests, generated modules compose through the aid of a special synchronization component, called Synch. We designed the Synch interface to simplify the generation of modules. This section describes the interface and the process of model compilation and integration.

Figure 7 illustrates the interface between the generated modules and the Synch component. Modules contain `Action` objects that *link* (explicitly refer to) `Gate`

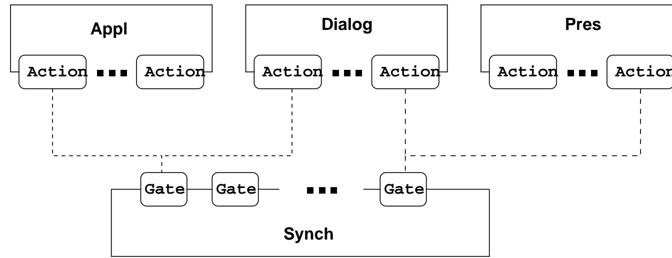

Figure 7. Structural depiction of composition according to Synch[Pres, Dialog, Appl].

objects in the Synch component. As the names suggest, an `Action` object reifies a LOTOS action, and a `Gate` object reifies a LOTOS gate. At runtime, `Actions` implement a unit of observable behavior in a module, and `Gates` implement the synchronization of `Actions` by conjunction. The mathematical connection between LOTOS actions and gates is reified using explicit links between `Action` and `Gate` objects. These links constitute the mechanism for composing generated modules with the Synch component: A module "plugs in" to the architecture by linking its `Action` objects to appropriate `Gate` objects in the Synch component. The dashed lines in **Figure 7** illustrate some (of many possible) links.

This architecture enables model synthesis to be treated separately from module integration, similar to the way compilation is treated separately from linking in traditional programming. This separation allows a module to be synthesized from a single model, independent of the synthesis of the other models. During synthesis, model-based compilers independently generate modules. Any behavior that must be observed by other modules must be packaged into an instance of the class `Action`. When emitting the code that creates this instance, the compiler also writes out the name of the associated gate to an auxiliary file. Consequently the output of a model compiler is a module and an auxiliary file listing the names of dependent gates. During module integration, a module integrator reads in these auxiliary files, creates the Synch component, and combines it with the generated modules to produce an executable image.

Going back to our running example, consider the compilation of the presentation model for the Print/Save dialog box (**Figure 3**). As the model is processed, the compiler emits `Action` objects that interface directly with UI toolkit widgets. After compilation, the Pres module will contain an `Action` for each widget in the dialog box. For example, there will be a distinct `Action` object paired with the **OK** and **CANCEL** buttons, each of the radio buttons, and **Filename** text-entry widget. To integrate the Pres module with the other modules, each of these `Actions` must link to `Gate` objects in the Synch component.

Note that when the `Actions` are being emitted, the corresponding `Gate` object will not yet exist, as the `Gate` is created by the module integrator. Thus, the link between an `Action` and its corresponding `Gate` cannot be established at compile time. Instead, an `Action` object is instantiated with the name of the gate over

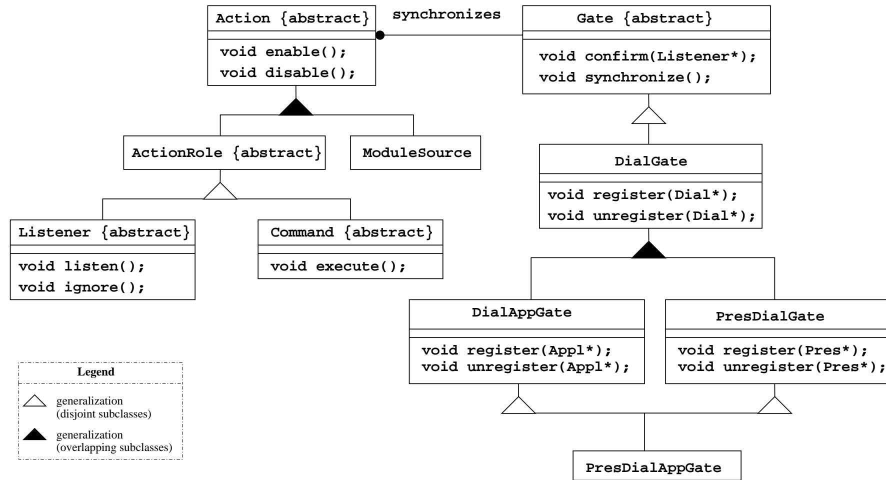

Figure 8. Detailed design of action and gate classes.

which it must synchronize. At runtime, the `Action` uses this name to locate the corresponding `Gate`. Because the module integrator creates a `Gate` for each named gate, the `Action` object can assume that the gate will exist at runtime. This design greatly simplifies model compilation: The presentation-model compiler need not concern itself with locating an object in another component. Rather, the compiler simply creates a module using `Action` objects and writes out the names of gates to an auxiliary file.

4.2. Behavior of the design structures

The synthesis of one MASTERMIND model can proceed independently of the synthesis of other models because the generated modules only refer to each other indirectly, through `Gate` objects. The `Gate` objects are responsible for determining when a synchronization should occur and dispatching control the associated `Action` objects in an appropriate order once the synchronization constraints are satisfied. Consequently, `Action` objects need not be concerned with these issues. Rather, `Actions` are concerned with implementing model-specific functionality. This separation is crucial to supporting the independence of model synthesis.

Figure 8 describes the design of classes `Action` and `Gate`. Class `Gate` is designed to internalize information about the modules whose actions are required to synchronize at the gate. Henceforth, we shall refer to this information as the *synchronization constraint* of a `Gate`. The rules of conjunction (**Figure 6**) establish a small number of possible variations of this constraint. At runtime, a `Gate` determine whether or not to synchronize by checking whether or not this constraint is satisfied. To make this determination, a `Gate` must infer the location (module) of each `Action` that wishes to synchronize over the `Gate`. We call this process of inference *tabulation.*

Tabulation occurs when an Action announces its readiness to synchronize. Such announcements are made by an Action *registering* itself with its Gate; an Action registers itself by passing itself to an invocation of the register operation on its Gate. When a Gate determines that its constraint is satisfied, it invokes the synchronize operation, which dispatches control to the registered Actions so that they may execute.

For a Gate to tabulate the modules that request activity, the Gate must be able to infer the module of every Action that registers. This means that an Action must know the module in which it exists. Class Action has a subclass, called ModuleSource, which further specializes into three subclasses, Pres, Dial,and Appl (not shown in the figure). The concrete class of every Action must inherit from one of these three subclasses. We implemented tabulation by specializing the register operation so that it dispatches based on these subclasses. The subclasses of Gate contain module variations of the register function. These subclasses embody each of the three possible synchronization constraints that arise in MASTERMIND. The constraint associated with class PresDialGate requires Pres and Dial actions to be present at the Gate. Similarly, the constraint associated with class DialApplGate requires Dial and Appl actions to be present at the Gate, and the constraint associated with class PresDialogApplGate requires all actions from all three modules to be present at the Gate. These are the only three types of synchronization constraints required of MASTERMIND-generated user interfaces.

The next issue concerns dispatching control to registered actions once a Gate's synchronization constraint is satisfied. MASTERMIND supports two different action-control mechanisms (generalized by ActionRole). One mechanism is a generic interface for executing a model-specific operation (class Command). The other mechanism is a generic interface for reactively observing an asynchronous event, such as a user interaction with a graphical widget (class Listener). What happens when a Gate's synchronization constraint is satisfied depends upon the control mechanisms used by the registered Actions. For example, if two Commands are waiting at a Gate, and they satisfy the synchronization constraint for the Gate, then the execute method for both Commands are invoked. If, instead, one of these actions is a Listener and the other is a Command, then the Command is not invoked until the Listener receives an event. Because Listeners are reactive, they need to be able to announce the reception of an event to the Gate. This is accomplished by invoking the confirm operation on the Gate.

A module requests the performance of an Action by invoking the operation enable. Enabling causes an Action to register itself with its Gate. Our design abstracts the logic for requesting the performance of an Action into the enable and disable methods, which correctly cooperate with the corresponding Gate irrespective of the particular synchronization constraints. Thus, the logic can be completely encapsulated in the abstract class Action, which a model-compiler writer need never modify. Moreover, model-compiler writers can package model-specific functionality using one of two quite different control policies, Command and Listener. One consequence of this design is that the module integrator must determine the type of Gate to emit. This is a simple task, however, given the information written to the

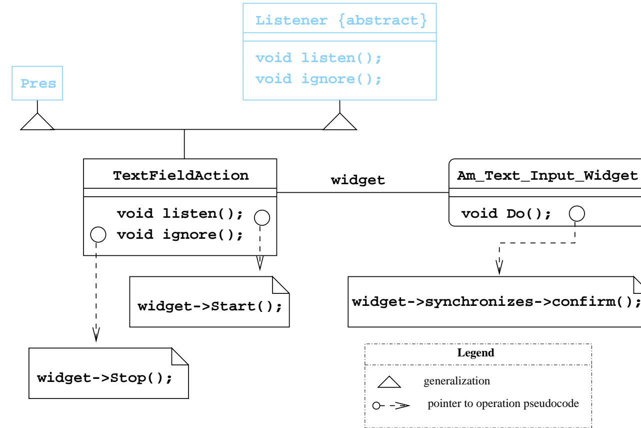

Figure 9. Example of use.

auxiliary files by the model compilers. For example, the gate *cancel* that is used in the Print/Save dialog is used in both the presentation model, where it observes the pressing of the **CANCEL** button, and in the dialog model, where it observes the completion of the dialog. Because modules compose by conjunction, the `Gate` associated with *cancel* always synchronizes an action from the Pres module with an action from the Dialog module. To implement this behavior, the module integrator emits an instance of `PresDialogGate`, which is returned when the associated `Actions` link to the named gate.

4.3. Example

We now demonstrate how these features work in the context of the Print/Save dialog. Recall from **Figure 3** the text entry field that allows a user to enter file name in which to save a document. In the dialog model (**Figure 4**), the entry of the file name is modeled as an atomic action over the gate *kbd*. To connect this dialog action to the text entry widget that ultimately witnesses the action, we need a presentation `Action` that knows how to attach to the text entry widget, and we need a `Gate` object to represent the *kbd* gate.

Figure 9 illustrates how a reusable action that listens for text entry can be created from the primitives introduced in **Section 4.2**. The presentation-model compiler emits instances of this class to implement text-entry boxes. In the figure, we rendered the primitive classes in grey to distinguish them from new objects and classes that the model-compiler writer creates. The new class is called `TextFieldAction`. It inherits from class `Pres` because its instances will be emitted into the Pres module. It inherits from class `Listener` because it is concerned with monitoring and controlling the text-input widget. The class is associated with an `Am_Text_Input_Widget` object by an association called `widget`. This object is prede-

fined in the Amulet toolkit [13], which the current version of MASTERMIND uses for presentation support. The `TextFieldAction` controls the Amulet object by invoking the `Start` and `Stop` operations on the object, which instruct the widget to enable and disable keyboard input. The invocations of these methods form the implementation of `listen` and `ignore` respectively. We also need a way for the widget to signal the `Action` object with the event. This is accomplished by overriding the `Do` method of the widget to go find the `Gate` associated with the `Action` and invoke the `confirm` operation on this `Gate` to signify the occurrence of the event. The `Do` method can be thought of as a callback function that Amulet invokes to deliver an keyboard event (in this case, the event is a keyboard return).

The example serves to illustrate the sequence of behaviors that are enacted by the MASTERMIND library primitives. Suppose an object of class `TextFieldAction` is registered at the `Gate` associated with *kbd*. If the synchronization constraint for this `Gate` is satisfied, the `Gate` invokes the `listen` method of the `TextFieldAction`. This invocation in turn causes the `Start` method of the `Am_Text_Input_Widget` to be invoked, which enables user input at the widget. If the synchronization context changes so that the constraint is no longer satisfied—either because the Pres module disables the `TextFieldAction` or because another module disables an `Action` that is waiting at the `Gate`, then the `Gate` invokes the `ignore` operation. This causes the `TextFieldAction` to invoke the `Stop` method of the `Am_Text_Input_Widget`, which disables text input. If, on the other hand, the user enters a string and hits the return key, the `Do` method of the widget is invoked. This causes the invocation of the `confirm` method on the `Gate`, and the `Gate` proceeds to `execute` any `Commands` that are waiting.

4.4. Summary

Our design enables independent code generation because the `Actions` in a generated module are insulated from `Actions` in other modules by the gate objects. We compose modules by creating `Gate` objects that embody the synchronization requirements of the models and by linking `Actions` to their `Gates`. The independence that is afforded by this approach allows model-based code generators to apply deep model-specific knowledge to the synthesis of code.

5. Results and status

We evaluated our solution to the model-composition problem with respect to power, correctness, and efficiency. Multi-paradigm actions have proved easy to specialize to accommodate features from disparate implementation toolkits and architectures. For example, we have specialized `Actions` to represent actions in: the Amulet object system [13], the C++ object system, and a special-purpose state-machine language. **Figure 2** summarizes the different applications and results.

5.1. Power

We were able to express user interfaces using our modeling notations in several case studies. We tested the quality of user interfaces on two specific examples: the Print/Save widget described in **Section 3.3** and an airspace-and-runway executive that supports an air-traffic controller (ATC) [19]. The former demonstrates the ability to generate common, highly reusable code for standard graphical user interfaces. The latter demonstrates the ability to support a complex application using a direct-manipulation interface.

The ATC example testifies to the power of our approach. When a flight number is keyed into a text-entry box, an airplane graphic, annotated with the flight number, appears in the airspace. As more planes come into the airspace, the controller keys their flight number into the text-entry box. When the controller decides to change the position of a plane, she does so by dragging the airplane icon to a new location on the screen. As soon as she presses and holds the mouse button, a feedback object shaped like an airplane appears and follows the mouse to the new location. When the mouse is released, the plane icon moves to the newly selected location.

The presentation model of the ATC example is quite rich. It specifies gridding so that airplane graphics are always uniformly placed within the lanes, and it specifies feedback objects that present controllers information about the planes during operation. In a real deployment, the locations of flights change in response to asynchronous application signals from special hardware monitors. In such a deployment, these signals would be connected to `Listener` actions and would fit into the framework without change. For more details on this case study and the Print/Save dialog, see Stirewalt's dissertation [19].

5.2. Correctness

In addition to being able to generate and manage powerful user interfaces, the composition of our modules is correct. Two aspects of our approach require justification on these grounds. First is the design of runtime action synchronization. Second is the synthesis of runtime dialog components (members of the set $\mathfrak{D}$) from dialog models.

This paper addresses the theoretical issues involved in the design of runtime action synchronization. The `Gate` and `Action` classes are traceable refinements of the corresponding concepts in LOTOS. In practice, we found this design to be quite robust. One aspect of synchronization correctness, which we do not address in this paper, is how to show that a model-specific specialization of `Action` does not violate the delicate callback protocol that underlies the system. For example, say that an `Appl`, which when modeled in LOTOS observes a value x and offers a value y, is to be implemented using a method invocation. The method should bind x to its parameter and bind its return value to y. Since value offerings are evaluated in sequence, how can we be sure that the ordering of evaluation does not interfere with the invocation of the method or vice versa? Currently, we check this by inspection, but we are investigating ways of packaging this problem so that a model checker

(e.g., SPIN [10], SMV [12]) can detect such anomalies. Stirewalt used the SMV model checker to validate the inter-operation of `Action` and `Gate` objects [19].

As we mentioned earlier, the MASTERMIND Dialog model notation is a syntactic sugaring for a subset of LOTOS. This language is described in greater detail in [20]. We implemented a prototype dialog model code generator whose correctness was validated as described in Stirewalt [19]. This code generator compiles dialog models without reference to other models.

5.3. Efficiency

We measured efficiency empirically by applying our code generator on the ATC example. We generated dialog modules and connected these with hand-coded presentation and application modules. On the examples we tried, we observed no time delays between interactions. We quantified these results by instrumenting the source code to measure the use of computation resources and wall-clock time. The maximum time taken during any interaction was 0.04 seconds. This compares well to the *de facto* HCI benchmark of response time, 0.1 seconds. We believe that more heavyweight, middle-ware solutions, such as implementing synchronization through object-request brokers, are not competitive with these results.

5.4. Future work

We are currently completing a more robust dialog code generator. This new code generator incorporates state-space reduction technology and will improve interaction time, which in the prototype is a function of the depth of a dialog expression, with interaction that executes in constant time.

6. Conclusions

How to generate code for a specialized modeling notation is a well understood problem. Integrating code generated from multiple models is not. Integration is much more complicated than merely linking compiled object modules. For models to be declarative, they must assume that entities named in one model have behavior that is elaborated in another model. Designers want to treat presentation, temporal ordering, and effect separately because each aspect in isolation can be expressed in a highly specialized language that would be less clear if it were required to express the other aspects as well. For interactive systems, composition by conjunction is essential to separating complex specifications into manageable pieces.

Unfortunately, programming languages like C++ and Java do not provide a conjunction operator. Such an operator is difficult to implement correctly and efficiently, and, in fact, we did not try to implement it. Rather, by casting model composition into a formal framework that includes conjunction, we are able to express a correct solution and then refine the correct solution into an efficient de-

sign. This is a key difference between our approach and middle-ware solutions that implement object composition by general event registry and callback.

Our results contribute to the body of automated software engineering research in two ways. First, our framework is a practical solution that helps to automate the engineering of interactive systems. Second, our use of formal methods to identify design constraints and the subsequent refinement of these constraints into an object-oriented design may serve as a model for other researchers trying to deal with model composition in the context of code generation. The formality of the approach allows us to minimize design constraints and is the key to arriving at a powerful, correct, and efficient solution.

References

1. G. D. Abowd. *Formal Aspects of Human-Computer Interaction.* PhD thesis, University of Oxford, 1991.
2. H. Alexander. Structuring dialogues using CSP. In M. Harrison and H. Thimbleby, editors, *Formal Methods in Human-Computer Interaction.* Cambridge University Press, 1990.
3. L. Bass and J. Coutaz. *Developing Software for the User Interface.* SEI Series in Software Engineering. Addison-Wesley, 1991.
4. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Network ISDN Systems*, 14(1), 1987.
5. T. P. Browne et al. Using declarative descriptions to model user interfaces with MASTERMIND. In F. Paternò and P. Palanque, editors, *Formal Methods in Human Computer Interaction.* Springer-Verlag, 1997.
6. P. Castells, P. Szekely, and E. Salcher. Declarative models of presentation. In *IUI'97: International Conference on Intelligent User Interfaces*, pages 137–144, 1997.
7. J. Coutaz. PAC, an object-oriented model for dialog design. In *Human Computer Interaction - INTERACT'87*, pages 431–436, 1987.
8. D. Harel. On visual formalisms. *Communications of the ACM*, 31(5), 1988.
9. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice/Hall International, 1985.
10. G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
11. G. E. Krasner and S. T. Pope. A cookbook for using the model view controller user interface paradigm in smalltalk. *Journal of Object Oriented Programming*, 1(3), 1988.
12. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992. CMU-CS-92-131.
13. B. A. Myers et al. The Amulet environment: New models for effective user-interface software development. *IEEE Transactions on Software Engineering*, 23(6), 1997.
14. B. A. Myers and M. B. Rosson. Survey on user interface programming. In *SIGCHI'92: Human Factors in Computing Systems*, May 1992.
15. R. Neches et al. Knowledgeable development environments using shared design models. In *Intelligent Interfaces Workshop*, pages 63–70, 1993.
16. P. Palanque, R. Bastide, and V. Sengès. Validating interactive system design through the verification of formal task and system models. In *Working Conference on Engineering for Human Computer Interaction*, 1995.
17. A. Puerta. The Mecano project: Comprehensive and integrated support for model-based user interface development. In *Computer-Aided Design of User Interfaces*, 1996.
18. J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.
19. R. E. K. Stirewalt. *Automatic Generation of Interactive Systems from Declarative Models.* PhD thesis, Georgia Institute of Technology, 1997.
20. R. E. K. Stirewalt and S. Rugaber. Design and implementation of mdl: The mastermind dialogue language. In preparation.

21. P. Szekely et al. Declarative models for user-interface construction tools: the MASTERMIND approach. In Bass and Unger, editors, *Engineering for Human-Computer Interaction*. Chapman & Hall, 1996.
22. Pedro Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *Bridges Between Worlds: Human Factors in Computing Systems: INTERCHI'93*, 1993.
23. S. Wilson et al. Beyond hacking: A model based approach to user interface design. In J. L. Alty, D. Diaper, and S. Guest, editors, *People and Computers VIII, Proceedings of the HCI '93 Conference*, 1993.
24. P. Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 6(5), September 1989.
25. P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):371–411, 1993.