Automating UI Generation by Model Composition

Kurt Stirewalt
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824
stire@cps.msu.edu

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
spencer@cc.gatech.edu

Abstract

Automated user-interface generation environments have been criticized for their failure to deliver rich and powerful interactive applications[24]. To specify more powerful systems, designers need multiple specialized modeling notations[17, 19]. The *model composition problem* is concerned with automatically deriving powerful, correct, and efficient user interfaces from multiple models specified in different notations. Solutions balance the advantages of separating code generation into specialized code generators with deep, model-specific knowledge against the correctness and efficiency obstacles that result from such separation. We present a solution that maximizes the advantage of separating code generation. In our approach, highly specialized, model-specific code generators synthesize run-time modules from individual models. We address the correctness and efficiency obstacles by formalizing composition mechanisms that code generators may assume and that are guaranteed by a run-time infra-structure. The mechanisms operate to support run-time module composition as *conjunction* in the sense defined by Zave and Jackson[28].

Keywords: Model-based user-interface generation, conjunction as composition, LOTOS.

1 Introduction

Building user interfaces (UIs) is time consuming and costly. Myers and Rosson[16] found that in systems with graphical UIs (GUIs), nearly 50% of source code lines and development time could be attributed to the UI. GUIs are usually built from a fixed set of modules composed in regular ways. Hence, GUI construction is a natural target for automation, and many commercial and research tools exist for that purpose[26]. While these tools have been successful in supporting the presentation aspect of GUI functionality, they provide only limited support for specifying behavior and the interaction of the UI with the 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 from 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, how it interfaces to the underlying application functionality, or how the user interacts with it. By focusing attention on a single aspect of an interactive system, a model can be expressed in a highly-specialized notation. This property makes systems developed using the model-based approach easier to develop and maintain than systems produced using other approaches.

The MASTERMIND project[5, 17] is concerned with the automatic generation of user interfaces from three kinds of models:

- *Presentation models* specifying the appearance of user interfaces in terms of their widgets and how the widgets behave.
- *Application models* specifying which parts (functions and data) of applications are accessible from the user interface.
- *Dialogue models* specifying end-user interactions, how they are ordered, and how they affect the presentation and the application.

Source code is generated from each model by a model-specific compiler, and the resulting three modules are composed (Figure 1). 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[3]. Inevitably, however, functionality specified by one model overlaps with or is dependent upon functionality specified by 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. Such effects are described in the dialogue model. The effect of pressing a button might also cause some application method to be invoked. Such effects are described in the 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 unified and the mechanism for this unification must be implemented efficiently.

The model composition problem is concerned with automatically deriving powerful, correct, and efficient user-interfaces from separate presentation, dialogue, and application models. We propose a two-fold solution to the model composition problem. First, we handle model integration by formalizing the three models as concurrent agents that synchronize on common actions (Section 3). The dialogue model expresses the temporal sequencing of these actions, and so it acts as the glue between the presentation and application models. Second, we make the synthesis efficient by implementing a run-time dialogue engine that actively synchronizes actions in the code generated from presentation and application models (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

We now present the issues that comprise the model composition problem and the influences on our solution. A recurring characteristic of all model-based approaches to UI generation is that models are specified using diverse and often incompatible notations. This characteristic complicates the formal definition of model composition (Section 2.1). Researchers in the Human Computer Interaction (HCI) community[2, 1] address this obstacle by representing models as agents defined in a process algebra (Section 2.2). Researchers in the Software Engineering community[27, 28] address similar problems with multi-paradigm specifications and describe a technique called *composition by conjunction* that solves the problem. These approaches deal with specifications only. Our solution to the model composition problem uses the agent-based representation of models, suggested by Alexander[2] and Abowd[1], to extend conjunction into a run-time mechanism for composing generated modules. The solution makes some assumptions about the class of applications being modeled. These assumptions are described in Section 2.3.

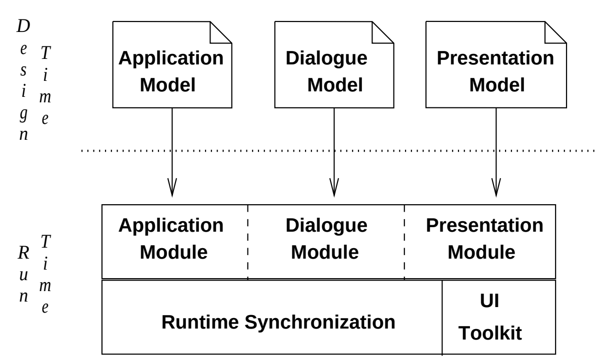

Figure 1: Model-based code generation

2.1 Model-based UI Generation

The model-based approach to interactive system development bases system analysis, design, and implementation on a common repository of models. Unlike conventional software engineering, in which designers construct artifacts 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 paradigm build interfaces by specifying models that describe the desired interface, rather than writing a program that exhibits the behavior[23].

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[7], for example, combines concepts and terminology from graphic design with mechanisms for composing complex presentations through functional constraints. Dialogue models often use state and event constructs to describe the user-computer conversation. Example notations include StateCharts[10] and Petri nets[18]. These schemes have a variety of composition mechanisms that include state hierarchy, orthogonality (concurrency), and alternation. The MASTERMIND Application Model combines concepts and terminology from object-oriented design techniques[20] with mechanisms for composing complex behavior based on method invocation. These examples illustrate how modeling notations provide intra-model composition mechanisms that vary among the different models. It is not clear that any one of these mechanisms are sufficient for inter-model composition, the subject of this paper.

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. We accomplish inter-model composition by introducing sequencing constraints over actions in the presentation and application models. The dialogue model embodies these constraints.

2.2 Composition Techniques

Our approach to model composition blends prior research on formal techniques for interactive system decomposition with a technique that has been used to support multi-paradigm specification.

2.2.1 Agent decomposition

Complex dynamic systems are often best understood as a network of cooperating *agents*, which are system entities with identity and behavior. A number of researchers, including Abowd[1] and Alexander[2], suggest that decomposing interactive systems into a collection of specialized agents illuminates usability and correctness properties. Often properties of a whole system can be described as properties of a single agent.

Alexander[2] decomposes dialogue specifications into presentation and application agents that represent the two parties of the user-computer conversation. Each of these agents are defined formally as processes in the CSP[13] notation, and they cooperate as prescribed by the synchronous parallel composition operator (||) of CSP. This decomposition, while nice conceptually, often distributes global behavior into complex protocols of local behavior. We improve upon this result by introducing a third agent, called dialogue, that explicitly represents global sequencing and synchronization constraints. Dialogue existed only implicitly in Alexander's framework. With an explicit dialogue agent, presentation and application may be specified more independently of each other.

There are other uses of the agents in this domain. Abowd[1], for example, decomposes interactive systems into four cooperating agents in order to test for usability properties. In addition, a number of run-time architecture frameworks are based on the decomposition of user-interfaces into a network of communicating agents (Section 2.3).

2.2.2 Agents and Processes

Processes are formal abstractions used to describe concurrent, communicating agents. A *process* is an entity whose internal structure can only be discovered by observing the actions in which it participates. Processes are often denoted by formal notations called *process algebras*. The CSP notation mentioned above is one example of a process algebra. In this paper, we use the LOTOS[4] notation, which uses temporal operators to specify permissible orderings and dependences over actions.

A process defined in LOTOS performs actions and interacts with other, concurrently executing, processes. Actions are built up out of atomic units called *events*. The set of events in which a process P may participate is called the *alphabet* of P (denoted $\alpha(P)$). If an event e is in the alphabet of two processes, then these processes can participate in actions that synchronize on e. When processes synchronize on an event, they simultaneously participate in actions over that event. During synchronization, an action can *offer* one or more values that can be *observed* by other actions participating in the same event.

Complex processes may be built by either combining sub-processes through an ordering operator (e.g., process P is the sequential composition of subprocesses P_1 and P_2) or by conjoining sub-processes so that they run independently but synchronize on common events. The conjoining operator specifies that P_1 and P_2 act independently and concurrently with the exception that actions with commonly named events synchronize. In LOTOS this is called *parallel composition* and is denoted $P_1 \parallel P_2$.

2.2.3 Agent Composition

In both CSP and LOTOS, parallel composition is defined so that more than two actions may synchronize on the same event at a given time. This is useful because, when multiple processes synchronize on the same action, each process can be thought of as adding a *constraint* to the occurrence of that action[4]. This style of expressing behavior is often called constraint-oriented specification. Constraint-oriented specification has been used to rigorously define composition in multi-paradigm specifications[27].

A multi-paradigm specification is one in which a system is described by multiple partial specifications written in different notations. Zave and Jackson[28] represent partial specifications as constraints in first-order predicate logic and note how logical conjunction specifies the simultaneous solution of these constraints. Specifications written in this style use mathematical free variables to represent values to be filled in by other specifications. When the specifications are conjoined, the free variables must satisfy constraints in all specifications. Conjunction is the mechanism for composing multi-paradigm specifications.

We adopt conjunction as the mechanism for multi-model composition. Designers treat different models as constraints on an overall solution, and system behavior is defined as any behavior that is consistent with the conjunction of constraints imposed by the dialogue, presentation, and application models respectively. Our results extend conjunction as a tool for composing models into a mechanism for composing run-time modules generated from these models.

2.3 GUI Architecture Frameworks

MASTERMIND generates software modules from declarative models. Modules must inter-operate in the context of an underlying run-time environment. Currently, there is no single run-time environment architecture that fits all user-interface problems[3]. There are, however, two frameworks that seem to capture most cases: SmallTalk's Model-View-Controller (MVC)[14] and Coutaz's Presentation-Abstraction-Control (PAC)[8]. Both of these cast a system into a collection of cooperating agents. An agent is a complete information processing unit with attributes from each model. Agents communicate with other agents by observing, acting on, and issuing events rather than making sequential procedure calls and waiting for their return.

In MVC, the *model*[1] is an agent responsible for maintaining application state, the *view* is an agent that reflects the state of the model on the display, and the *controller* is an agent that accepts input from the user and initiates changes in the model. Users in the MVC framework interact with the controller and observe changes in the view. The MVC framework is useful for multi-viewed user-interfaces because view agents can be added to the framework without affecting the other agents.

In PAC, the *presentation*[2] combines input and output handling (the model and view agents of MVC) into a single agent, the *abstraction* is similar to the model in MVC, and the *controller* is an agent that ensures that presentation and abstraction remain synchronized. The PAC framework faithfully represents the modularity decisions of object-oriented user-interface toolkits like Artkit[12] and subArctic[9]. These toolkits package input and output together into reusable widgets called *interactors*.

Interactive systems tend to be structured using either the MVC or PAC frameworks, with the choice depending on the nature of the application. The composition technique presented in this paper assumes systems are built using the PAC framework. Those systems that are difficult to to organize into a PAC model may be difficult to specify and generate in MASTERMIND.

2.4 Summary

There are three issues that must be addressed in solving the model composition problem. First, the approach must generate user-interfaces with rich dynamic behavior. Second, the correctness of module composition must be demonstrated. Third, the generated modules must cooperate to form an efficient system. The first issue is a function of the expressive power of the modeling notations. In MASTERMIND, special-purpose modeling notations were chosen to overcome this deficiency[17, 5]. This paper is concerned with generating correct implementations with maximal efficiency while preserving the expressive power of MASTERMIND models.

3 Design Requirements

Recall from Figure 1 that each class of model has a code generator that synthesizes run-time 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. In this section, we present a detailed specification of the relationship between model composition and how the associated modules cooperate at run-time.

3.1 Notation

The subject of this paper is the automatic generation and composition of run-time 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 composition correctness. In formal correctness arguments, we often refer to all three types of constructs and distinguish them by using different fonts. MASTERMIND models are written in the Sans Serif font (e.g., Presentation, Dialogue, and Application). LOTOS processes are written in capital italic letters (e.g., P, D, and A, respectively). Run-time modules are written in German letters (e.g., $\mathfrak{P}$, $\mathfrak{D}$, and $\mathfrak{A}$, respectively).

We now briefly define some process algebra terminology. Suppose the behavior of an agent can be described by the LOTOS process B. If the agent can observe or offer values by synchronizing on event e, its behavior from that point is be described by another process B'. Notationally, the synchronization of B on e is written $B(e)$, and we assume there is always a unique B' such that $B(e) = B'$. This assumption follows the fact that the systems under study are deterministic.

We assume that the number of generated run-time modules is known and does not vary during the execution of a generated system. In MASTERMIND, there are exactly three generated modules, and so the behavior of the system at any time can be described as $B \equiv B_1 \parallel B_2 \parallel B_3$ where B_i describes the future behavior of module i. The behavior of B after synchronizing on an event e (denoted $B(e)$) is another process $B' = B'_1 \parallel B'_2 \parallel B'_3$ where:

$$B'_i = \begin{cases} B_i(e) & \text{if } e \in \alpha(B_i) \\ B_i & \text{otherwise} \end{cases}$$

[1]Not to be confused with our use of the word model.

[2]Not to be confused with MASTERMIND Presentation.

Any event that can be observed of a module can be observed of all the modules in composition. This observation will be important when we define the Ω observer function in Section 3.4.

We occasionally need to represent the architectural embedding of Figure 1 equationally. In this case, we refer to the entire run-time system, $\mathfrak{U}$, as a module composed from $\mathfrak{P}$, $\mathfrak{D}$, and $\mathfrak{A}$, using the notation $\mathfrak{U}[\mathfrak{P}, \mathfrak{D}, \mathfrak{A}]$.

3.2 Inter-model Composition

Model-based code generators construct a run-time module from a design-time model. The code generation strategy is model-specific, reflecting the specialization of models to a particular aspect of the system. At run time, however, modules must cooperate, and the cooperative behavior must not violate any of the 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. In the presentation model, for example, objects are implemented using graphical primitives in the Amulet toolkit[15], and attribute relations are implemented as declarative formulas that, at run-time, eagerly propagate attribute changes to dependent attributes. As long as changes in these attributes do not trigger behavior in the dialogue or application models, these aspects can be ignored when considering model composition. In the 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 designer extensions do not trigger behavior in dialogue or presentation models, this behavior may also be ignored when defining model composition.

Within a module, entities compose according to a model-specific policy. In the presentation model, for example, objects compose by part-whole aggregation, and attributes compose by formula evaluation over dependent attributes. In the 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 over the entire system. These constraints and contributions must be captured in some form and used to reason about model composition.

We chose to 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 constraining behavior.

```
process PrintSave[ print, save, go, cancel, layout, kbd
                   lpr, write]
    (lpdhost, filename : string,
     doc               : doctype,
     port              : bool) : exit :=
   P[go, lpr, write, layout, kbd] [> (cancel ; exit)
where
   process P[go, lpr, write, layout, kbd] : exit :=
       Layout[go, lpr, layout]
     [> (save ; F[go, lpr, write, layout, kbd])
   endproc
   process F[go, lpr, write, layout, kbd] : exit :=
       Edit[go, write, kbd]
     [> (print ; P[go, lpr, write, layout, kbd])
   endproc
   process Layout[go, lpr, layout] : exit :=
       (layout ? port ; Layout[go, lpr, layout])
     □ (go ; lpr !lpdhost!port!doc; exit)
   endproc
   process Edit[go, write, kbd] : exit :=
       (kbd ? filename ; Edit[go, write, kbd])
     □ (go ; write ! doc ! file ; exit))
   endproc
endproc
```

Figure 2: LOTOS abstraction of print/save dialogue.

3.3 An Example

We now demonstrate an example of inter-model behavior expressed as a LOTOS process. The dialogue 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. Such widgets allow the user to print a document or save it to a file on disk. Options specific to printing, such as print orientation (e.g., portrait vs. landscape), and to saving, such as the file to save in, are typically enabled and disabled depending upon the user's choice of task. These ordering dependencies are reflected in the dialogue model for this widget. The inter-model behavior of this dialogue model can be described by the LOTOS process in Figure 2.

The process *PrintSave* can synchronize on any of the events that follow in square brackets. In this example, the events *print*, *save*, *go*, *cancel*, *layout*, and *kbd* (line 1 in the figure) define points for synchronizing with the presentation; whereas the events *lpr* and *write* (line 2) define points for synchronizing with the underlying application. The parameters *lpdhost* and *filename* (line 3) store the name of the default printer and the user-selected filename respectively. The parameter *doc* (line 4) represents the document to be printed or saved, and the parameter *port* (line 5) represents the print orientation (portrait if true, landscape if false).

A separate presentation model defines buttons labeled **Send to Printer**, **Save to File**, **Print**, and **Cancel** which, when pressed, offer the events *print*, *save*, *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 *kbd* event parameterized by the contents of the string typed so far. Note that the actual keys being pressed are not returned, as editing functionality is best handled in a text widget and is not what we would consider inter-model behavior. A separate application model 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*).

The temporal structure of dialogue, presentation, and application model composition is given in the behavior specification (line 6). The behavior of *PrintSave* is the behavior of the process P (defined on lines 8 through 11) with the caveat that it may be disabled (terminated) at any time by the observation of the *cancel* event. Process P represents what interactions and application invocations must happen in order to send a document to the printer. Most of this functionality is actually expressed in the subprocess *Layout* (defined on lines 16-19). 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 alternates from the **Send to Printer** button to the **Save to File** button in the presentation model. The process F (defined on lines 12-15) likewise behaves like the process *Edit* (defined on lines 20-23) in the normal case, but is disabled if the event *print* is observed. Note that F and P are mutually disabling, which means that the user can switch back and forth between printing and saving as many times as he or she likes until hitting the **Go** button.

$$\begin{array}{ccc} \mathsf{Dialogue} & \xrightarrow{\mathcal{A}_D} & \mathit{Process} \\ {\scriptstyle \mathcal{C}_D}\downarrow & & \downarrow{\scriptstyle tr} \\ \mathfrak{D} & \xrightarrow{\Omega} & \mathit{TraceSets} \end{array}$$

Figure 3: Dialogue compiler correctness.

3.4 Models, Modules, and Processes

Processes like those in Figure 2 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. In this section, we formalize correctness conditions for the MASTERMIND dialogue model, but a similar formalization exists for all three of the MASTERMIND models.

Figure 3 shows the relationship between dialogue models (members of the set Dialogue), run-time modules generated by dialogue models (members of the set $\mathfrak{D}$), and the inter-model behavior of dialogue models (members of the set *Process*). The relationships between these sets are defined as functions that map members of one set into another. The function $\mathcal{C}_D : \mathsf{Dialogue} \rightarrow \mathfrak{D}$ maps dialogue models to run-time implementation modules. Think of $\mathcal{C}_D$ as an abstract description of the dialogue model compiler. The function $\mathcal{A}_D : \mathsf{Dialogue} \rightarrow \mathit{Process}$ maps dialogue models into LOTOS processes describing their inter-model behavior. Think of $\mathcal{A}_D$ as an abstract interpretation of the dialogue model. The function $\Omega : \mathfrak{D} \rightarrow \mathit{TraceSets}$ maps run-time implementation modules (of any kind) into event traces of their observable behavior. Think of Ω as an observer of run-time behavior. Finally the function $tr : \mathit{Process} \rightarrow \mathit{TraceSets}$ maps a LOTOS process to the set of all possible action traces that can be observed of that process.

These sets and functions are related by the commutative diagram of Figure 3. 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 actions is recorded by Ω. We say that a dialogue model $d \in \mathsf{Dialogue}$ is consistent with the module $\mathcal{C}_D(d)$ if every trace $\sigma \in \Omega(\mathcal{C}_D(d))$ is in the set $tr(\mathcal{A}_D(d))$ and if there are no sequences $\varsigma \in tr(\mathcal{A}_D(d))$ such that $\varsigma \notin \Omega(\mathcal{C}_D(d))$. That is, the inter-model behavioral interpretation of d agrees exactly with the observable behavior of the run-time module generated from d. Commutativity of the diagram requires this property for any dialogue model expressible in the set Dialogue.

3.5 Model-based UI Synthesis

The correctness relationship between models and modules (Figure 3) can be extended to specify the

$$
\begin{array}{l}
\forall\, p \in \mathsf{Presentation} \\
\quad \forall\, d \in \mathsf{Dialogue} \\
\quad\quad \forall\, a \in \mathsf{Application} \\
\quad\quad\quad \Omega(\mathfrak{U}[\mathcal{C}_P(p), \mathcal{C}_D(d), \mathcal{C}_A(a)]) \\
\quad\quad\quad\quad = \\
\quad\quad\quad tr(\mathcal{A}_P(p) \parallel \mathcal{A}_D(d) \parallel \mathcal{A}_A(a))
\end{array}
$$

Figure 4: Module composition correctness.

correctness of module composition. We now have functions $\mathcal{A}_P$, $\mathcal{A}_D$, and $\mathcal{A}_A$ that map models into LOTOS processes. Since models should compose by conjunction, these processes should compose by parallel composition. We also have a run-time module combinator $\mathfrak{U}[]$ that combines modules from $\mathfrak{P}$, $\mathfrak{D}$, and $\mathfrak{A}$ into a single module whose actions are observable by the Ω function. Figure 4 shows the constraints on the behavior of these entities. Let $p \in \mathsf{Presentation}$, $d \in \mathsf{Dialogue}$, and $a \in \mathsf{Application}$. Then the code generated from these models is correct if, for any observable behavior σ, σ is a legal trace in the parallel composition of the models and vice-versa. 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 Design

We now turn to the design of the run-time synchronization module and model-specific compilers of Figure 1. The correctness conditions of Figure 4 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). This allows us to design a generic infra-structure of inter-model cooperation and to assume this design when crafting model-specific code generation strategies. The design refines the notions of action and synchronization, which form the basis of inter-module communication in Figure 4, into run-time objects that implement these constraints.

4.1 Run-time Control

One concern in designing a system is the implementation of software control[20]. Control can be implemented in many ways. In procedural systems, for example, control is synonymous with location in the code; whereas in concurrent systems, control is distributed and managed by multiple objects concurrently. User-interface software generally implements an event-driven, sequential control scheme, in which a single thread provides a facade of concurrency by dispatching small callback routines when input device activity is sensed. In the interest of providing a single style of control in our systems, we adopt the event-driven, sequential implementation.

The choice of software control implementation influences the design of actions and synchronization. Actions in the presentation module correspond to input device behavior like mouse and keyboard events. Since these events invoke callback procedures, we implement synchronization as a callback. This means that the temporal structure of the inter-model behavior must be implemented in such a way that all legal actions are enabled, all illegal actions are disabled, and after action synchronization, new actions are enabled or disabled. If one model can be made to represent this temporal structure, then functionality provided by the other models can be abstracted into context-independent actions and implemented using method callbacks.

By design, the temporal structure of the dialogue model represents the synchronization needs of the entire program. This makes it natural to treat the dialogue module as the arbiter of system control. In the architecture presented in Figure 1, the **Dialogue** module is a reactive component that computes the enabled/disabled status of actions in response to action synchronizations, and the other modules are collections of code that are invoked when actions synchronize. At run time, every action causes the dialogue module to compute the next state of the system. Based on this next state, actions embedded in other modules are enabled, disabled, or activated as appropriate.

4.2 Action Synchronization

The dialogue module computes the set of enabled actions as a function of the observed actions. This means that actions can be thought of as entities that are enabled, disabled, and activated by an omniscient dialogue agent, and that the model-specific interpretation of said actions can be structured to occur when the action is activated. We now describe the `Action` object, a run-time entity that encapsulates the status (enabled or disabled) of an observable action with an activation procedure that can be specialized by model-specific code generators to implement desired functionality. Figure 5 shows our design as an OMT[20] object model.

The first thing to note about our design is that the class `Action` is abstract. Specifically, it contains an abstract method called `enable()` that must be supplied by a subclass. Subclassing in the OMT notation is denoted by a triangle one point of which

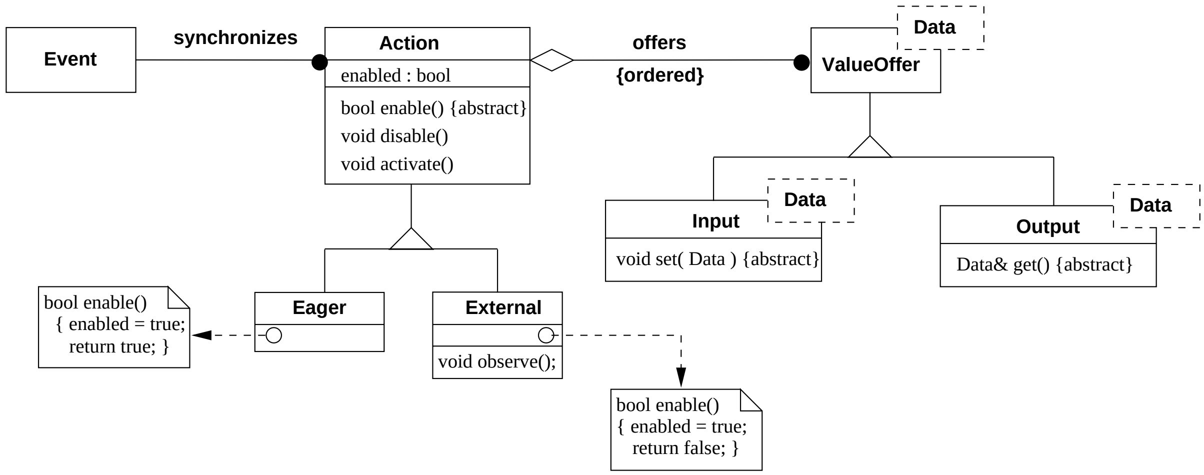

Figure 5: Object model for synchronization mechanism.

is connected to the superclass with one or more lines emanating out to its subclasses. In user-interface software, presentation model actions are associated with presentation module interaction objects, and application model actions are associated with the invocation of methods of objects in the application module. When an action in the presentation model can be offered, the graphical object associated with that action must be enabled and made ready to accept user activity. When the graphical object detects such activity, it must signal the rest of the system than an action synchronization is occurring. When an action in the application model can be offered, a method in the application module must be invoked. The object model of Figure 5 distinguishes between these interpretations of action enabling by subclassing `Action` into those that are `External` and those that are `Eager`. The synchronization requirements of an `Eager` action are met when the action is enabled; whereas `External` actions require both being enabled and observing activity generated by an external entity like a mouse.

The class `Action` has an association called `synchronizes` with the class `Event`. Objects of class `Event` represent process events upon which multiple actions synchronize. `Event` objects encapsulate a unique name with the synchronization requirements of actions from multiple models. Note that while there is an object of class *Action* for every action in any model, there is a unique object of class `Event` for any distinct event.

Recall from Figure 2 that actions are usually accompanied by one or more value inputs or outputs. To accommodate this, we designed the class `Action` to aggregate zero or more objects of the parameterized class `ValueOffer`. Aggregation in OMT is expressed with the diamond operator, and it means that zero or more objects of class `ValueOffer` are parts of every object of class `Action`. The parameter `Data` in Figure 5 names a data type that *parameterizes* class `ValueOffer`. A parameterized class (denoted as a box with a dashed box in the upper right corner) can define local attributes or methods whose signatures depend vary with the parameter. Note that the subclasses `Input` and `Output` use the parameter value to specialize `set` and `get` methods. These abstract methods must be supplied by model-specific code generators, which know how to supply and receive values in model-specific contexts.

Code generators produce code for actions given only the name of the event and zero or more value offers (either inputs or outputs) associated with the synchronization of the event. Table 1 shows the syntax of actions as definable in LOTOS. The only information a code generator has about an action is the event name, whether the value offers are inputs or outputs, the name and type of the variable in which to store the input, and the expression used to compute the output. Note that objects can be constructed from the syntax of actions in the process notation.

4.3 Run-time Execution

`Event` objects internalize synchronization require-

Table 1: Syntax of actions.

$$
\begin{array}{lll}
\textit{action} & ::= & \textit{EventName offer}* \\
\textit{offer} & ::= & \textit{input} \\
 & \mid & \textit{output} \\
\textit{input} & ::= & '?'\ \textit{Variable}\ ':'\ \ \textit{Type}; \\
\textit{output} & ::= & '!'\ \textit{Expr};
\end{array}
$$

ments of multiple actions and issue `activate()` and `get()` and `set()` methods to `Action` and `ValueOffer` objects as callbacks. To make this work, `Event` objects contain a pointer to all of the `Action` objects that synchronize and vice-versa. When an `Action` object is enabled by the dialogue module, the return value (true or false) is recorded in the corresponding `Event` object so that the synchronization requirements can be tabulated. A return value of false indicates that an action is `External`, in which case the `Event` object records that the action is enabled but waits for external confirmation that the action has been chosen by the user. Once all of the synchronization requirements have been met, the `Event` issues the appropriate `activate`, `get`, and `set` methods and then instructs the dialogue module to compute the next state. This process is described in greater detail in [21].

5 Results and Status

Automatic approaches to user-interface software generation have been criticized for their failure to deliver rich and powerful interactive applications[24]. This deficiency has been addressed by incorporating more and more powerful models into the development process. This led to the model composition problem and our present work. We validated our approach on three points: power, correctness, and efficiency.

Power We were able to express the UI's in several case studies using our modeling notations. 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) in a busy airport[21]. The former demonstrates the ability to generate common, highly reusable, tasks for standard graphical user-interfaces. The latter demonstrates the ability to support a complex task using a direct-manipulation interface.

The ATC example testifies to the power of our approach. When flight numbers are keyed in to a text-entry box, an airplane graphic, augmented with the flight number, appears in the airspace. As more planes come into the airspace, the controller keys their flight number in a text-entry box. When the controller decides to change the position of a plane, he does so by dragging the airplane graphic to a new location on the canvas. As soon as he 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 give users information during an operation. In a real deployment, the location of the flights would probably change in response to asynchronous application signals from special hardware monitors. In such a deployment, these signals would be connected to `External` rather than `Eager` actions and would fit into the framework without change. For more details on this case study and the print/save dialogue, see Stirewalt [21].

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 run-time action synchronization. This paper addresses the theoretical issues involved here. In practice, we have found the design to be quite robust. Second is the synthesis of the runtime dialogue component (member of the set $\mathfrak{D}$) from a dialogue model. As we mentioned earlier, the MASTERMIND Dialogue model notation can be thought of as a syntactic sugaring for a subset of Full LOTOS. We implemented a prototype dialogue model code generator whose correctness was validated in Stirewalt[21] (also described in [22]).

Efficiency We measured efficiency empirically by applying our prototype code generator on the ATC example. We generated dialogue 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, which is 0.1 seconds. We believe that more heavyweight, middleware solutions, such as implementing synchronization through object-request brokers, are not competitive with these results.

We have presented an infra-structure of run-time support for multi-model composition. The OMT design model provides guidance to developers of model-specific code generators. We are currently completing a new industrial-strength, dialogue code generator. This new code generator is incorporating state-space reduction technology described in [22] and will improve interaction time that, in the prototype, is a function of the depth of a dialogue expression with constant time interaction. We are also working on adapting the presentation model code generator described in [7] to cooperate work within our infrastructure.

6 Conclusions

We began this work investigating the feasibility of generating user-interface code from multiple declarative models and quickly discovered that generating code for a specific model is easy; whereas integrating the code generated from these models is difficult. Integration is much more complicated than mere linking of 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. A button object described in a presentation model has temporal context defined in a dialogue model and functionality defined in an application model. Designers want to treat presentation, temporal context, 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 parallel composition, we were able to express a correct solution and then refine the correct solution into an efficient design. This is a key difference between our approach and middle-ware solutions that try to implement parallel 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 allowed us to minimize design constraints and was 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, June 1991.

[2] H. Alexander. Structuring dialogues using CSP. In *Formal Methods in Human-Computer Interaction[11]*, pages 273–295. 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] H. J. Bullinger and B. Schackel, editors. *Human Computer Interaction - INTERACT'87.* North Holland, Amsterdam, 1987.

[7] P. Castells, P. Szekely, and E. Salcher. Declarative models of presentation. In *IUI'97: International Conference on Intelligent User Interfaces*, pages 137–144, 1997.

[8] J. Coutaz. PAC, an object-oriented model for dialog design. In *Human Computer Interaction - INTERACT'87 [6]*, pages 431–436. North Holland, Amsterdam, 1987.

[9] W. K. Edwards, S. E. Hudson, R. Rodenstein, T. Rodriguez, and I. E. Smith. Systematic output modification in a 2d user interface toolkit. In *UIST'97: ACM Symposium on User Interface Software Technology*, 1997.

[10] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5), 1988.

[11] M. Harrison and H. Thimbleby, editors. *Formal Methods in Human-Computer Interaction.* Cambridge University Press, 1990.

[12] T. R. Henry, S. E. Hudson, and G. L. Newell. Integrating gesture and snapping into a user interface toolkit. In *Third Annual Symposium on User Interface Software and Technology. Proceedings of the ACM SIGGRAPH Symposium*, pages 112–122, October 1990.

[13] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice/Hall International, Englewood Cliffs, New Jersey, 1985.

[14] 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), August 1988.

[15] B. A. Myers et al. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[16] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *SIGCHI'92: Human Factors in Computing Systems*, May 1992.

[17] R. Neches et al. Knowledgeable development environments using shared design models. In *Intelligent Interfaces Workshop*, pages 63–70, 1993.

[18] 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.

[19] A. Puerta. The Mecano project: Comprehensive and integrated support for model-based user interface development. In *[25]*, pages 19–36. Namur University Press, 1996.

[20] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[21] R. E. K. Stirewalt. *Automatic Generation of Interactive Systems from Declarative Models*. PhD thesis, Georgia Institute of Technology, 1997.

[22] R. E. K. Stirewalt and G. D. Abowd. Composition property analysis: a new strategy for model checking user-interface designs. In *Sixth International SIGSOFT Symposium on the Foundations of Software Engineering (FSE'98)*, 1998. Submitted for publication.

[23] P. Szekely et al. Declarative interface models for user interface construction tools : The MASTERMIND approach. In L. Bass and C. Unger, editors, *Engineering for Human-Computer Interaction*. Chapman & Hall, 1996.

[24] 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*, pages 383–390. Addison Wesley, April 1993.

[25] J. M. Vanderdonckt, editor. *Computer Aided Design of User Interfaces*. Namur University Press, Namur, 1996.

[26] Visual Edge Software Ltd., Cupertino, CA. *Extending and Customizing UIMX*, 1993.

[27] P. Zave. A compositional approach to multiparadigm programming. *IEEE Computer*, September 1989.

[28] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):371–411, 1993.